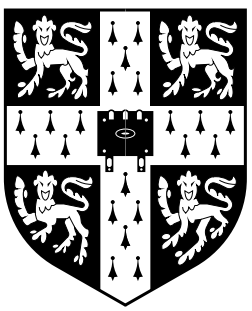
A Method of Program Refinement

Jim Grundy

Fitzwilliam College



A dissertation submitted for the degree of
Doctor of Philosophy in the University of Cambridge

November 1993

COPYRIGHT © Jim Grundy 1993
All rights reserved.

This work is copyright and may not be reproduced, stored nor distributed in whole or in part for commercial purposes. Permission is hereby granted to reproduce, store and distribute this work for non-profit educational and research purposes, provided that the source is acknowledged and the copyright message is retained. Enquiries regarding use for profit should be directed to the author.

Second edition 1993

Typeset by the author using the LaTeX document preparation system.
Printed in England at the University of Cambridge, Computer Laboratory.

To Sandi
and to my parents

Foreword

This foreword introduces the second edition of my dissertation. This edition contains numerous minor corrections and improvements over the first. Most of these improvements were suggested by my examiners, Cliff Jones and Larry Paulson. I would like to thank them both, firstly for agreeing to be my examiners, and then for reading my dissertation so thoroughly and giving me so many helpful comments on its contents.

This second edition will appear as Technical Report 318 of the University of Cambridge, Computer Laboratory; and it is to this report that any citations should be made.

Jim Grundy
19 November 1993

Contents

Foreword **v**

Preface **xi**

1 Introduction **1**
1.1 Motivation 1
1.2 Background 1
1.3 Overview 7
1.4 How to Read this Dissertation 9

2 Window Inference **11**
2.1 Introduction 11
2.2 Example 12
2.3 Windows and Theorems 14
2.4 Reusing Rules 18
2.5 Lemmas 21
2.6 Conjectures 24
2.7 Implementation 27
2.8 Related Work 27
2.9 Applications 28
2.10 Conclusions 29

3 Predicative Programming—A Survey **31**
3.1 Introduction 31
3.2 The Basic Concepts 32
3.3 The Relational Model 37
3.4 The Partial Model 41
3.5 The Total Model 44
3.6 Comparison 48
3.7 More-Elaborate Models 52
3.8 Conclusions 53

4 A Three-Valued Logic for Refinement **55**
4.1 Introduction 55
4.2 A Three-Valued Logic 55
4.3 Three-Valued Predicates as Specifications 57
4.4 Refining with Three-Valued Predicates 60
4.5 Programs as Three-Valued Predicates 63
4.6 Window on Three-Valued Booleans 64
4.7 Nonstandard Predicates 65
4.8 Conclusions 66

5 A Method of Refinement **69**
5.1 Introduction 69
5.2 Program Refinement 70
5.3 skip Statements 71
5.4 Assignment Statements 72
5.5 Conditional Statements 73
5.6 Nondeterministic Choices 73
5.7 Sequential Compositions 74
5.8 Local Variable Blocks 79
5.9 Labelled Blocks 80
5.10 Example 85
5.11 Verification 93
5.12 Conclusions 93

6 A Refinement Case Study **97**
6.1 Introduction 97
6.2 The Hardware 97
6.3 The System Software 99
6.4 A Simple Editor 100
6.5 Handling Errors 103
6.6 The Screen 104
6.7 Data Abstraction 106
6.8 Comments on the Specification 107
6.9 The Refinement 107
6.10 Comments on the Refinement 115

7 Conclusions **119**
7.1 Future Work 121

Appendixes

A Definitions **123**
A.1 Pairs 123
A.2 Lists 123
A.3 Arrays 125

B Refinement of the Line Editor **127**
B.1 Introduction 127
B.2 The Refinement of *INIT* 127
B.3 The Refinement of *EDIT* 134
B.4 The Refinement of *PAINT* 172
B.5 Comments 191

References **193**

Index **203**

Preface

A method of specifying the desired behaviour of a computer program, and of refining such specifications into imperative programs is proposed. The refinement method has been designed with the intention of being amenable to tool support, and of being applicable to real-world refinement problems.

Part of the refinement method proposed involves the use of a style of transformational reasoning called 'window inference'. Window inference is particularly powerful because it allows the information inherent in the context of a subexpression to be used in its transformation. If the notion of transformational reasoning is generalised to include transformations that preserve relationships weaker than equality, then program refinement can be regarded as a special case of transformational reasoning. A generalisation of window inference is described that allows non–equivalence preserving transformations. Window inference was originally proposed independently from, and as an alternative to, traditional styles of reasoning. A correspondence between the generalised version of window inference and natural deduction is described. This correspondence forms the basis of a window inference tool that has been built on top of the HOL theorem proving system.

This dissertation adopts a uniform treatment of specifications and programs as predicates. A survey of the existing approaches to the treatment of programs as predicates is presented. A new approach is then developed based on using predicates of a three-valued logic. This new approach can distinguish more easily between specifications of terminating and nonterminating behaviour than can the existing approaches.

A method of program refinement is then described by combining the unified treatment of specifications and programs as three-valued predicates with the window inference style of transformational reasoning. The result is a simple method of refinement that is well suited to the provision of tool support.

The method of refinement includes a technique for developing recursive programs. The proof of such developments is usually complicated because little can be assumed about the form and termination properties of a partially developed program. These difficulties are side-stepped by using a simplified meaning for recursion that compels the development of terminating programs. Once the development of a program is complete, the simplified meaning for recursion is refined into the true meaning.

The dissertation concludes with a case study which presents the specification and

development of a simple line-editor. The case study demonstrates the applicability of the refinement method to real-world problems. The line editor is a nontrivial example that contains features characteristic of large developments, including complex data structures and the use of data abstraction. Examination of the case study shows that window inference offers a convenient way of structuring large developments.

Acknowledgements

I would like to thank the Australian Defence Science and Technology Organisation for their financial support. The Committee of Vice-Chancellors and Principals of the Universities of the United Kingdom were also kind enough to help support my studies though an Overseas Research Student Award. In addition, Fitzwilliam College and the Computer Laboratory both helped send me to various conferences.

I would like to thank my supervisor, Mike Gordon, both for his initial encouragement to come to Cambridge, and for his guidance and supervision over the past three years. I am indebted to him for giving me complete freedom to pursue my research in any direction, while somehow still being able to give me useful advice on whatever I chose to look at. I would also like to thank him for gently prodding me towards finishing my dissertation.

The members of the Cambridge Hardware Verification Group, both past and present, have helped make my time in Cambridge both an enjoyable and an enlightening one. I would particularly like to thank those members of the group with whom I have had the pleasure of sharing an office: Brian, Jacob, Joakim, John, Laurent, Mats, Monica, Richard, Sara and Victor.

The following people have been kind enough to proof read parts of this dissertation, or the papers on which it is based: Richard Boulton, Juanito Camilleri, David Carrington, Francisco Corrella, Katherine Eastaughffe, Andrew Gordon, Mike Gordon, Brian Graham, Andrew Gravell, John Harrison, Ian Hayes, Roger Jones, Mats Larsson, Tom Melham, Andy Moran, Monica Nesi, Maris Ozols, Peter Robinson, Emil Sekerinski, John Staples, Nigel Ward and Joakim von Wright. Where I have been wise enough to follow their advice, the comments of these people have improved this dissertation greatly. Any remaining flaws are, of course, my own doing. Tom Melham should be mentioned explicitly, not only for being a most diligent proof reader, but also for teaching me much of what little I know about writing style and presentation.

The following people were brave enough to use my window inference interface to the HOL system: Andrew Gordon, Mats Larsson, Laurent Thèry and Joakim von Wright. Through their use of the tool, they helped me to find many of its bugs; they also suggested many improvements. Mats deserved a medal for bravery when he chose to use my tool as part of his own work. He and Katarina also deserve a medal for the warmth of their hospitality and their patience when trying to teach me to ski.

I would like to thank Roger Needham for keeping the department running; and Chris, Graham, Martyn and Piete for keeping the machines running. I would also like to thank Angela for helping with mail and faxes, and Lewis and Paola in the library for helping to track down those difficult to find references.

Finally, I would like to thank Sandi for saying 'yes'.

Fitzwilliam College J.G.
Cambridge

5 October 1993

Chapter 1

Introduction

This dissertation proposes a method of specifying the desired behaviour of a computer program, and of refining specifications into implementations in an imperative programming language. One of the central ideas of the method is to regard program refinement as a special case of transformational reasoning. To this end, a general technique for transformational reasoning is developed by generalising an existing goal-oriented style of reasoning called 'window inference' [RS93]. The other central idea is to give specifications and programs a uniform treatment as predicates in a three-valued logic. The combination of these ideas yields a method of program refinement that is simple, yet flexible, and is well suited to the provision of tool support.

1.1 Motivation

Ever since the 1960s and the pioneering work of Floyd [Flo66] and Hoare [Hoa69] on assigning meaning to programs, it has been possible to construct a proof that a program is correct. Since then, much work has been devoted to improving methods for describing the meaning of a computer program. However, even today, only a handful of large program developments are proved correct. It would seem likely that much of the difficulty of constructing proofs of correctness may not be theoretical, but methodological. To put it another way; perhaps we do not need to prove different theorems, but to prove theorems differently. The goal of this dissertation is to develop a new method of developing correct programs that is applicable to large problems.

1.2 Background

The design of the method of programming described in this dissertation began by performing several semiformal program derivation examples. The idea was to concentrate initially on finding a natural style of programming, with less emphasis placed on its formality. Step two was to formalise both the style of reasoning and the logic

employed. The object of the exercise was to devise a formal, yet natural, method of writing correct programs. The resulting method was not directly inspired by any particular body of previous work. However, the writing of many authors was influential in its development, and is discussed where relevant throughout the dissertation.

Before describing any new thoughts on the problem of developing correct computer programs, it is appropriate to consider that work which has gone before. Therefore, the remainder of this section is dedicated to a brief review of the previous work in the area.

Verification

The first approach to the development of correct software, known as *verification*, was to write a program and then try to prove that it meets its specification. Examples of post development correctness arguments can be found as early as 1966 with Naur's work on 'general snapshots' [Nau66]. The difficulty of performing such proofs quickly led to attempts at their automation. Examples of such program verification systems include the following: m-EVES [Gru91b, ORA90], MALPAS [RTP90] and GVE [You87]. Typically, verification tools take a specification and a program, and calculate the *verification condition*; the theorem that would establish that the program meets the specification. The calculation of the verification condition is usually based on a set of Hoare-logic style rules [Hoa69] that capture the meaning of the programming language.

Both the specification and its implementation are typically expressed in a common language, sometimes referred to as the *interface language* of a verification tool. MALPAS-IL and m-Verdi are examples of interface languages used with the MALPAS and m-EVES systems respectively. Interface languages invariably allow programs to be annotated with assertions in the style of Naur's snapshots. Including assertions in a program can simplify the task of generating its verification condition. It is a characteristic feature of interface languages that the specification and implementation components of the language are distinct and can not be mixed.

The final step of a verification is to prove the verification condition using whatever theorem proving support is supplied by the tool. This task is usually complicated by the sheer size and complexity of the verification conditions generated, even for the smallest of programs.

Program Calculation

Almost from the beginning, it was realised by Dijkstra [Dij68] that a more productive approach would be to develop a program and its proof together. This idea was further developed in the book *A Discipline of Programming* [Dij76]. The method of program development espoused in this book, and in some later work [DF88], is known as *program calculation*. Program calculation is characterised by a mix of

formal and semiformal reasoning. Such development methods are referred to as being *rigorous*, as opposed to being completely formal. The degree of informality in program calculation makes providing tool support for it difficult. Indeed, the advocates of program calculation reject the benefits of mechanised help, preferring instead the freedom offered by pen and paper.

The programming language developed for use with program calculation is called the *guarded-command language*. It is a simple imperative programming language that contains an assignment statement, a loop construct, and a nondeterministic conditional statement. Given a set of command and guard pairs (a *guard* is a boolean expression), the nondeterministic conditional selects a command with a true guard and executes it. Hence the name 'guarded-command language'. The simplicity of this little language makes it a common choice for use with formal development methods.

A second innovation to come from Dijkstra's work on program calculation is the *weakest precondition* style of program semantics. Suppose we have a predicate Q and a command C, the expression $\mathrm{wp}(C, Q)$ denotes a predicate that describes the set of all initial states from which executing C will lead to termination in a state that satisfies the postcondition Q. This is called the weakest precondition of C with respect to Q. Dijkstra gives the guarded-command language meaning by describing how to calculate the weakest precondition of each construct with respect to any post-condition. Weakest preconditions are now a popular style of semantics, particularly for nondeterministic programming languages where their use is significantly simpler than the more traditional approach based on power domains.

The Vienna Development Method

The design and implementation of any nontrivial program is a complex task. This complexity can best be attacked by decomposing the design process into a number of simple steps. This style of program development was first proposed by Nicklaus Wirth [Wir71], and has since been known as *step-wise refinement*. The concept of step-wise refinement has influenced almost every modern program development method.

One of the earliest approaches to couple formal methods with step-wise refinement was the Vienna Development Method (VDM) [Jon90]. VDM encompasses the entire program development process from specification to implementation. Operations are specified in the now common precondition, postrelation style. After specification, abstract data-types are replaced by concrete data-types in a process called *data reification*. The next stage of the development, called *operational decomposition*, is the step-wise derivation of an implementation. Each step of the decomposition involves introducing a programming language construct. There is a small collection of decomposition laws that are used to justify these steps (one per program construct). It is not expected that users of VDM will want to develop their own decomposition rules. Indeed, the semantics of the programming language and the proofs of the

decomposition laws are not included in the usual expositions of VDM. VDM was developed as a practical approach to the programming problems of the day. As such, the emphasis of VDM is on the early—more tractable—phases of the development, with many applications ending after the product has been specified. For similarly pragmatic reasons VDM adopts a rigorous, rather than formal, approach to software engineering.

During the operational decomposition phase of VDM, a development will pass through stages where it is partly implemented and partly specified. This requires the use of a language that can capture specifications and implementations, and all the stages in between. Such languages are now known as *wide-spectrum* languages. Wide-spectrum languages are distinct from the interface languages used with program verification systems in that while they are both capable of expressing specifications and implementations, interface languages can not mix the two concepts to represent the intermediate stages of a development. VDM contains one of the first wide-spectrum languages, a technique that is now almost standard in formal and semiformal program development.

Jones's 'Logic of Partial Functions' [BCJ84] is another innovation associated with VDM. In the Logic of Partial Functions, each type is extended with an extra value that denotes nontermination. Similar approaches to describing nonterminating behaviour had been tried before, most notably by Scott with his 'Logic of Computable Functions' [SS71]. However, Scott's approach was somewhat different. While the Logic of Partial Functions is a true three-valued logic, The Logic of Computable Functions could be more accurately described as a two-valued logic for reasoning about three-valued terms. VDM was the first application of three-valued logic in a formal program development setting. While much of the more recent work on program development has adopted weakest preconditions as its semantic basis, three valued logic remains a promising approach. Chapter 4 of this dissertation takes a fresh look at the application of three-valued logic to program development. (Actually the work described in chapter 4 is more in keeping with Scott's approach than Jones's.)

The RAISE Method

While industrial experience with the use of VDM has proved most encouraging, there are some difficulties with the method, particularly when applied to large-scale developments. The RAISE (Rigourous Approach to Industrial Software Engineering) development method [HPP92, NHWG89, Pre87] takes VDM as a starting point, and aims to improve on it. The emphasis of the RAISE method is on practicality and industrial applicability. For this reason, the developers of RAISE reject strict adherence to formality as too restrictive and difficult, favouring instead a rigorous semiformal approach.

A key feature of the RAISE approach is the provision of tools to assist the de-

velopment process. Much of the additional effort required when adopting a rigorous approach to programming is involved with routine bookkeeping tasks. Good tool support can greatly reduce the burden of these mundane tasks. The RAISE tool set includes the following components:

- syntax directed editors and browsers for the RAISE specification and development languages;
- type checkers for the specification and development languages;
- automatic cross-referencing tools;
- version control tools, including undo and playback facilities;
- a proof editor that can record both formal proof steps and informal natural language justifications;
- translators from RAISE implementations to imperative languages including Modula-2 and Ada.

The developers of RAISE have avoided more elaborate theorem proving and formal program transformation tools because they feel that the state of the art in these tools falls short of being practical for industrial application.

Refinement Calculi

Not all approaches to the development of trusted software reject strict formality. The combination of formality and step-wise refinement characterises a number of approaches known as *program refinement* methods. The earliest work on program refinement was that of Ralph Back [Bac81, Bac88]. This work was later reinvented and extended both by Carroll Morgan [MR87, Mor88b] and by Joe Morris [Mor87]. Each of these methods takes an existing programming language (Dijkstra's guarded-command language), and extends it by adding a new 'specification' statement that can be used to describe an arbitrary requirement. The addition of a specification statement yields a wide-spectrum language capable of capturing all stages of a development. In the work of both Morgan and Morris, the specification statement takes the form of a precondition, postrelation pair. In Back's work, specification is accomplished using a generalised form of the assignment statement in which a single relation captures the requirements. Each of the statements in the wide-spectrum language is given meaning using weakest preconditions. Refinement laws are then proposed for strengthening and decomposing specification statements into other, executable, language constructs. These laws are proved correct with respect to the weakest-precondition semantics. The resulting system is known as a *refinement calculus*.

If, as is asserted by the developers of the RAISE method, the bookkeeping tasks associated with using a rigorous approach to software engineering can benefit from automation, then the same is doubly true of a strictly formal approach. Although work on tool support for the various refinement calculi is now in progress [GNU92, Vic90, Naf92, vWS91, vWHLL93, vW94] (some of which exploits ideas developed in this thesis and published earlier [Gru91c, Gru92]), none of the original papers on the refinement calculi [Bac81, MR87, Mor87] addresses this concern. This suggests that the ease with which tool support could be provided for these calculi was not a consideration when they were being designed.

Predicative Programming

An alternative approach to program refinement has been proposed by Mili [Mil83], Hoare [Hoa84] and Hehner[Heh84]. In this approach, the semantics of a programming language is expressed in the specification language. The result is a wide-spectrum language in which program statements can be mixed freely with other, nonexecutable, expressions. Mili, Hoare, Hehner, and the authors of more recent work on this approach [Gra92, Nei90, Sek92] have all used either a single predicate or relation as their specification language. As a result, this style of program refinement is often known as *predicative programming*.

Describing the semantics of a programming language in the specification language avoids using two formalisms: one for specification and another, meta-level formalism, for reasoning about programs, specifications and their meanings. This makes proofs from first principles easier than when using a refinement calculus. Predicative programming is therefore less dependent on using a preproved set of refinement laws. Providing mechanical assistance that supports this extra flexibility will be even more challenging (though potentially more rewarding) than providing support for the more rigid style of development used with refinement calculi.

Predicative Programming is a most promising approach to program refinement, and has been as source of inspiration for the work described in this dissertation. Chapter 3 contains a survey of the various approaches to predicative programming.

Transformational Programming

A body of work closely related to program refinement has evolved from quite distinct origins. In 1977 Burstall and Darlington published a paper describing how a clear, but inefficient, program could be transformed into an equivalent more efficient, but perhaps less clear, one [BD77]. This marked the beginning of work on a style of development known as *transformational programming*. Burstall and Darlington's work was based on functional programs, which they regarded as being better suited to transformational manipulation than imperative ones.

The transformation approach has since been developed beyond use as an opti-

misation technique by the Munich Project CIP (Computer-aided, Intuition-guided Programming) [BMPP89, BW91, CIP85, CIP87]. Development with the CIP system begins with a specification. The specification is then repeatedly refined to reduce the degree of nondeterminism it allows. The result of this process is much like a functional program. Equivalence transformations, in the style of Burstall and Darlington, can then be used to transform the implementation into a more efficient form. The final stage of the CIP method allows the transformation of a functional style expression into an imperative form.

Most work on formal development methods uses a *model-oriented* style of specification. In model-oriented specifications, a new object is described by constructing a model of it in terms of existing, more primitive, ones. For example, a list can be described as a set of number and element pairs, where the number represents the position of the element in the list. Transformational programming systems, however, typically adopt an *algebraic* style of specification. In algebraic specifications, a new object is described in terms of a number of constants and functions, and by a set of axioms that describe the interrelationships between them. For example, a list may be described algebraically using the functions head, tail and cons, and the constant nil. The following axioms give the description.

$$\vdash \mathsf{tail}(\mathsf{cons}\ e\ l) = l \qquad \vdash \mathsf{head}(\mathsf{cons}\ e\ l) = e \qquad \vdash \mathsf{cons}\ e\ l \neq \mathsf{nil}$$

Model-oriented specifications are frequently regarded as offering a more abstract form of description than their algebraic counterparts. However, algebraic specifications are closely related to functional programs, which makes them a better choice for use with transformational programming. The program refinement method developed in this dissertation is based on the model-oriented approach.

Apart from the different specification styles, one of the primary differences between program refinement and program transformation is their treatment of nondeterminism. In program refinement the nondeterminism in a specification is retained for as long as possible to allow greater freedom at all stages of the development. In the transformational approach, nondeterminism is eliminated as soon as possible to allow the application of traditional transformational programming techniques.

The ideas developed in the CIP project are pursued further by the PROSPECTRA (PROgram SPECification and TRAnsformation) project [KB86, KB91]. The PROSPECTRA method is designed to produce programs in an annotated subset of Ada. Tool support for transformational programming is considered an important part of both the CIP and PROSPECTRA projects. For example, the CIP system contains tools for editing and recording a development, for selecting and applying program transformation rules, and checking the applicability conditions of those rules.

1.3 Overview

This dissertation comprises seven chapters and two appendices:

Chapter 1: *Introduction*

Chapter 2: *Window Inference*

A method of transformational reasoning called 'window inference' is described. The version of window inference presented is a generalisation of the original [RS93], to allow transformations that preserve relations weaker than equality.

Chapter 2 also describes a correspondence between window inference and natural deduction. This correspondence forms the basis of a window inference interface to the HOL theorem prover [GM93].

The material presented in this chapter first appeared as a paper in *The Proceedings of the 1991 International Tutorial and Workshop on the HOL Theorem Proving System and its Applications* [Gru91c], and later—in a condensed form—in *The Proceedings of the 5th Refinement Workshop* [Gru92].

Chapter 3: *Predicative Programming—A Survey*

Several methods of specifying the behaviour of a program using a predicate have been proposed. This chapter presents a survey of these methods and analyses their strengths and weaknesses. In particular, the methods are examined for their ability to describe both terminating and nonterminating behaviour, the ease with which they can capture the semantics of an imperative programming language, and the simplicity of their refinement relation.

This survey first appeared in *The Proceedings of the International Conference on Formal Methods in Programming and Their Applications* [Gru93a].

Chapter 4: *A Three-Valued Logic for Refinement*

Chapter 4 presents a new approach to the problem of describing the behaviour of a computer with a predicate, by using a three-valued logic. The new approach can distinguish between requirements for terminating and nonterminating behaviour more easily than the approaches surveyed in chapter 3. The approach also provides a simple semantics for an imperative programming language, and has a simple refinement relation.

The material presented in this chapter appeared first in *The Proceedings of the International Conference on Formal Methods in Programming and Their Applications* [Gru93b].

Chapter 5: *A Method of Refinement*

In chapter 5, the ideas presented in chapters 2 and 4 are combined to yield a refinement method. Window inference is used to transform specifications, phrased as three-valued predicates, into implementations in an imperative programming language. The method includes a number of rules for frequently repeated refinements, and a technique for developing recursive programs. The

difficulties of proving the correctness of the technique are avoided by using a simplified meaning for recursion. The chapter contains a proof that once a development is complete, the simplified meaning can be replaced by the true one.

An earlier version of the refinement method, which guaranteed only the partial correctness of the resulting implementations, was first described in *The Proceedings of the 5th Refinement Workshop* [Gru92].

Chapter 6: *A Refinement Case Study*
Chapter 6 presents a case study which includes the specification of a simple line editor and an overview of its refinement. A more complete presentation of the development can be found in Appendix B. Although small, the line editor is a nontrivial example that contains features characteristic of real developments.

Chapter 7: *Conclusion*
The conclusion presents a summary of the material presented in the preceeding chapters, including a discussion of the strengths of the refinement method developed therein. It also examines the possibilities for further work arising from this dissertation.

Appendix A: *Definitions*
This appendix contains the definitions used in the refinement case study.

Appendix B: *Refinement of the Line editor*
Appendix B presents the details of the refinement case study.

1.4 How to Read this Dissertation

This dissertation is designed to be read in its entirety in the order presented. However, an understanding of the core ideas may be gained by reading just the following portions: chapters 1 and 2, section 3.2, and chapters 4, 5 and 7. After reading this much, readers seeking more background on the decision to adopt a three-valued logic should return to the remaining sections of chapter 3, while those who would like to see the refinement method applied to a nontrivial problem should read chapter 6, and Appendices A and B.

Those readers who would like to assess the scope of this work by first reading the introduction and conclusion, are advised that the individual chapters contain their own introductions and conclusions that present significant information. Such readers are advised to consider reading at least the following portions of the dissertation: chapter 1, sections 2.1, 2.7–2.10, 3.1, 3.8, 4.1, 4.8, 5.1, 5.12, 6.1, 6.8 and 6.10, and chapter 7.

Chapter 2

Window Inference

2.1 Introduction

This dissertation describes a method of refining programs from their specifications. The view that program refinement is just an instance of the general problem of transformational reasoning lies at the heart of the method proposed. This chapter describes a general method of transformational reasoning called 'window inference' and its implementation as an interface to the HOL theorem prover. The examples in this chapter are pedagogic in nature and are designed to illustrate the various features of window inference. They are *not* intended to be indicative to the reasoning ability of the system. Later parts of the thesis contain more significant examples of window inference, culminating in Appendix B which contains a program derivation case-study. Earlier versions of the material presented in this chapter can be found in *The proceedings of the 1991 International Workshop on the HOL Theorem Proving System and its Applications* [Gru91c] and, in a condensed form, in *The Proceedings of the Fifth Refinement Workshop* [Gru92].

Window inference is a style of reasoning that allows users to transform an expression by restricting attention to a subexpression and transforming it. The remainder of the surrounding expression is left unchanged. While transforming a subexpression it is possible to make assumptions based on its context. For example, suppose we wish to transform the expression $A \wedge B$; this may be done by transforming the subexpression A under the assumption that B is true. This assumption is valid because if B were false, the entire expression would be false regardless of the value of A. Note that only one part of an expression may be transformed at a time. This prevents the transformation of expressions like $A \wedge A$ into true by using each A to transform the other.

Using contextual information in the transformation of subexpressions is an aid to effective reasoning. It does, however, make the proof tree more complex. This chapter describes the implementation of a tool that supports window inference and shields its users from the complexities of exploiting contextual information.

The principles of window inference were first described by Staples and Robinson [RS93], and implemented in their proof checker Demo2 [TRS90]. Window inference, as described by Staples and Robinson, was designed to allow users to transform expressions while preserving an equivalence relation. The system was conceived as a method of performing goal-directed proofs. This chapter extends Staples and Robinson's work by relaxing the constraint that the system should preserve equivalence relations. The window inference system described here can be tailored to preserve any preorder.[1] It is this generalisation that allows window inference to form the basis of a method for refining specifications into implementations.

The window inference system described here also extends the original proposed by Staples and Robinson by introducing a feature called 'conjectures'. Conjectures allow users to make suppositions which they may establish later if they prove to be useful. Staples and Robinson have independently extended their system with a similar facility.

Staples and Robinson describe window inference independently from, and as an alternative to, traditional styles of reasoning like natural deduction. Here, window inference is described in terms of natural deduction. This description serves to reassure the reader who is more familiar with traditional styles of deduction that window inference is a valid style of reasoning. It can also serve as the basis for the implementation of a window inference interface to a natural deduction style theorem prover like HOL [GM93] or Isabelle [Pau93].

2.2 Example

Within a window inference system, reasoning is conducted with a stack of windows. Each window has a *focus*, F, which is the expression to be transformed; a set of formulae, Γ, called the *assumptions*, which can be assumed in the context of the window; and a *relation*, R, that must relate the focus and the expression to which it is transformed. All relations used with the system must be preorders. Using a notation derived from that of Staples and Robinson, a window is written as follows:

$$\begin{array}{rl} ! \, \Gamma & \text{— There may be several such assumptions} \\ R \star F & \end{array}$$

The focus of a window can be an expression of any type.

Users may either transform the focus of the window on top of the stack, push a subwindow onto the stack to concentrate on a subexpression of the focus (called 'opening' a window), or pop the top window from the stack and use the information it held to transform the window underneath it (called 'closing' a window).

[1] A preorder is a reflexive, transitive relation.

We now present a brief example which illustrates the window-inference style of reasoning. Consider the following expression:

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

This expression is neither provable nor refutable, for this reason it falls outside the scope of most proof methods. Using window inference we can transform this expression to another form which is logically equivalent to the original. We begin with a window which has the expression as its focus, and equality as the relation to be preserved.

$$= \star \boxed{(B \vee C \supset A \vee C)} \supset (A \wedge B) \vee (A \supset B)$$

We use the box around the left-hand side of the implication to indicate that we wish to open a subwindow there.

$$\begin{array}{l} \quad ! \neg(A \wedge B) \\ \quad ! A \\ \quad ! \neg B \\ = \star B \vee C \supset A \vee C \end{array}$$

In the context of the subwindow, we may assume that the right-hand side of the implication is false; because if the right-hand side of the implication where true, then the entire implication would be true, regardless of the left-hand side. The assumption that the right-hand side of the implication is false can be decomposed into the following simple assumptions: $\neg(A \wedge B)$, A and $\neg B$. Such decompositions are performed automatically by the window inference interface to the HOL system that is described later in this chapter.

Using the assumption that A is true, we can simplify the focus to true (T).

$$\begin{array}{l} \quad ! A \\ \quad ! \neg B \\ \quad ! \neg(A \wedge B) \\ = \star \mathsf{T} \end{array}$$

This is as far as we can go with this subwindow, so we close it and return to its parent where the term we originally opened the subwindow on is now replaced by T.

$$= \star \mathsf{T} \supset \boxed{(A \wedge B)} \vee (A \supset B)$$

We open another subwindow, this time on the subexpression $A \wedge B$.

$$\begin{array}{l} \quad ! A \\ \quad ! \neg B \\ = \star A \wedge B \end{array}$$

In the context of this subwindow we may assume that A is true and B is false. Using these assumptions, we may simplify the focus to false (F).

$$\begin{array}{r} ! \; A \\ ! \; \neg B \\ = \star \; \mathsf{F} \end{array}$$

We now close this subwindow and return to its parent where the term $A \wedge B$ is now replaced by F.

$$= \star \; \mathsf{T} \supset \mathsf{F} \vee (A \supset B)$$

The focus of this window may now be simplified as follows.

$$= \star \; A \supset B$$

This completes our example, which proves that the original expression is equivalent to the much simpler form $A \supset B$.

2.3 Windows and Theorems

In this section we describe how reasoning with window inference is related to reasoning with natural deduction. This relationship forms the basis of the window-inference interface to the HOL system.

Each window *holds* a theorem which relates the current focus to the original focus of the window. Examples of such theorems appear throughout this chapter beside the windows that hold them. For the sake of simplicity, the hypotheses of such theorems will be shown as the assumptions of the window that holds it. It is, however, only necessary for the hypotheses to contain those assumptions that have actually been used to transform of the focus.

Creating a Window Stack

To begin reasoning with a window inference system, the user creates a window stack. Creating a window stack to transform an expression F_0 under the assumptions Γ, while preserving the relation R, results in the stack containing the single window below:

$$\begin{array}{ll} ! \; \Gamma & \text{theorem:} \\ R \star F_0 & \Gamma \vdash F_0 \; R \; F_0 \end{array}$$

The window initially holds the theorem that F_0 is related to itself. This follows from the requirement that R is a preorder, and hence reflexive.

Transforming a Window

Consider the window:

$!\,\Gamma$	theorem:
$R \star F_n$	$\Gamma \vdash F_n\ R\ F_0$

A transformation of the focus of a window from F_n to F_{n+1} must be justified by a theorem of the following form:

$$\Gamma \vdash F_{n+1}\ R\ F_n$$

We combine this theorem with the one held by the window using the fact that R is a preorder, and therefore transitive. The resulting theorem shows that the new focus, F_{n+1}, is related to the original focus, F_0.

$$\frac{\Gamma \vdash F_{n+1}\ R\ F_n \qquad \Gamma \vdash F_n\ R\ F_0}{\Gamma \vdash F_{n+1}\ R\ F_0}$$

This inference means that the focus can be transformed, and that doing so results in the following window:

$!\,\Gamma$	theorem:
$R \star F_{n+1}$	$\Gamma \vdash F_{n+1}\ R\ F_0$

When implemented as an interface to the HOL theorem prover, we can use any of the existing tools in the HOL system to derive the theorem required to justify a transformation. For example, the rewriting conversions of HOL are immediately available for use with window inference. Moreover, because the HOL system is built on top of the general purpose programming language ML [GMW79], users can write their own functions to automate the proofs of transformations they use frequently.

Subwindows

The opening and closing of a subwindow is controlled by special inference rules called *window rules*. Opening a subwindow pushes a new window onto the stack. The components of the subwindow are extracted from the relevant window rule. Closing the subwindow pops the top window from the stack and uses the theorem it held and the window rule to infer how to transform the window below.

Consider the following window (the box indicates the position on which we wish to open a subwindow):

$!\,\Gamma$	theorem:
$\supset \star A \supset \boxed{B}$	$\Gamma \vdash (A \supset B) \supset F_0$

The window rule below allows us to open a subwindow on the expression B.

$$\frac{X, \Gamma \vdash Y' \supset Y}{\Gamma \vdash (X \supset Y') \supset (X \supset Y)}$$

The resulting subwindow is as follows:

$$\begin{array}{ll} \quad ! \; \Gamma & \text{theorem:} \\ \quad ! \; A & A, \Gamma \vdash B \supset B \\ \supset \star \; B & \end{array}$$

We can then transform the focus (preserving implication) from B to B' under the assumption that A is true.

$$\begin{array}{ll} \quad ! \; \Gamma & \text{theorem:} \\ \quad ! \; A & A, \Gamma \vdash B' \supset B \\ \supset \star \; B' & \end{array}$$

Using the theorems held by the subwindow and the parent window, an appropriate instance of the window rule, and the transitivity of the relation ($\supset$); the following inference can be built.

$$\frac{\dfrac{A, \Gamma \vdash B' \supset B}{\Gamma \vdash (A \supset B') \supset (A \supset B)} \qquad \Gamma \vdash (A \supset B) \supset F_0}{\Gamma \vdash (A \supset B') \supset F_0}$$

This inference justifies closing the subwindow, which is popped from the stack. The transformed parent window is then as follows.

$$\begin{array}{ll} \quad ! \; \Gamma & \text{theorem:} \\ \supset \star \; A \supset B' & \Gamma \vdash (A \supset B') \supset F_0 \end{array}$$

Similarly, we could open a subwindow on A using this window rule:[2]

$$\frac{\neg Y, \Gamma \vdash X' \subset X}{\Gamma \vdash (X' \supset Y) \supset (X \supset Y)}$$

Note that the subwindow preserves the relation '$\subset$'—implication in the reverse direction. This new relation is required as the form of a window inference rule dictates the the new focus be on the left-hand side of the relation and the original focus on the right-hand side.

The general form of a window rule is as follows:[3]

$$\frac{\gamma, \Gamma \vdash f' \; r \; f}{\Gamma \vdash F[f'] \; R \; F[f]}$$

where

[2]The reverse implication relation $\subset$ is defined as follows: $\vdash \forall x \; y. \, (x \subset y) = (y \supset x)$.

[3]The notation $E[e]$ denotes the substitution of e for some distinguished free occurrence of a subexpression within E.

- $F[f]$ is the original focus of the parent window;
- f is the original focus of the subwindow;
- f' is some transformed focus of the subwindow;
- $F[f']$ is the correspondingly transformed focus of the parent window;
- R is the relation preserved by the parent window;
- r is the relation preserved by the subwindow;
- Γ is the set of assumptions which hold in the parent window;
- γ is the set of assumptions which, in addition to Γ, hold in the subwindow.

The only restriction placed on window rules is that they must take this form and, of course, be valid rules of inference.

The window-inference interface to the HOL system allows users to select where to open a subwindow by giving a path that describes the position of the desired subterm. The system then chooses the window rule(s) required to open a subwindow on the selected term, and pushes the resulting subwindow onto the stack. Users of Laurent Théry's graphic front end to the window-inference interface [Thé93] can open a subwindow simply by selecting the desired term with the mouse.

Example

Suppose we wish to simplify the predicate $(A \supset B) \wedge A$ by transforming it to another equivalent predicate. We begin by creating a window stack containing only the window below.

$$= \star (\boxed{A \supset B}) \wedge A$$

theorem:
$\vdash ((A \supset B) \wedge A) = ((A \supset B) \wedge A)$

We decide to open a subwindow on the expression $A \supset B$. The system chooses an appropriate window rule, in this case:

$$\frac{Y \vdash X' = X}{\vdash (X' \wedge Y) = (X \wedge Y)}$$

Using this rule with appropriate instantiations, a new window is pushed onto the stack. Only the top window of the stack is displayed.

$$\begin{array}{l} ! \; A \\ = \star \; A \supset B \end{array}$$

theorem:
$A \vdash (A \supset B) = (A \supset B)$

The focus of this window can be simplified to B using the assumption A:

$$\begin{array}{ll} \quad !\ A & \text{theorem:} \\ =\star\ B & A \vdash B = (A \supset B) \end{array}$$

The window is then closed. Its removal from the stack reveals the window below which is transformed as follows:

$$\begin{array}{ll} =\star\ B \wedge A & \text{theorem:} \\ & \vdash (B \wedge A) = ((A \supset B) \wedge A) \end{array}$$

The theorem held by the top window of the stack states that the transformed focus, $B \wedge A$, is equivalent to the original focus, $(A \supset B) \wedge A$.

2.4 Reusing Rules

To build a window inference system it is *not* necessary to compile a library of window rules that cover every position at which a user may wish to open a window and preserve every relation the user may wish to work with. Techniques for substituting and composing rules can be used to implement a system that, with only three basic window rules, allows users to open a window on any subexpression and substitute equal expressions for each other. This basic system can then be extended to use contextual information and preserve more general relations by adding more window rules as they are required.

Composing Rules

Suppose a user wants to open a subwindow on the subexpression B in the focus of the following window:

$$\begin{array}{l} !\ \Gamma \\ =\star\ (A \wedge \boxed{B}) \wedge C \end{array}$$

It is not necessary to have a window rule for this specific case. A rule can be built by composing the rule for opening on the right-hand side of a conjunction with the rule for opening on the left-hand side of a conjunction, as follows:

$$\dfrac{\dfrac{X, Z, \Gamma \vdash Y' = Y}{Z, \Gamma \vdash (X \wedge Y') = (X \wedge Y)}}{\Gamma \vdash ((X \wedge Y') \wedge Z) = ((X \wedge Y) \wedge Z)}$$

Using a rule built in this way is equivalent to opening a window at $A \wedge B$ and then opening a window at B. In this case, opening a window at B would result in the window below:

$$\begin{array}{l} !\ \Gamma \\ !\ C \\ !\ A \\ =\star\ B \end{array}$$

Each window rule is stored with information about the position of the focus of the subwindow within that of the parent. When asked to open a window at a position for which no specific window rule is stored, the system builds a substitute by composing window rules that descend the parse tree of the focus to the desired position.

Stronger Rules

A relation S is *stronger* than another R if for any x and y, $x\ S\ y$ implies $x\ R\ y$. Note that $=$ will be stronger than any relation used with the system because of the requirement that all such relations should be preorders, and hence reflexive.

Suppose that the following is the current window:

$$\begin{array}{ll} !\ \Gamma & \text{theorem:} \\ R \star F[\boxed{f}] & \Gamma \vdash F[f]\ R\ F_0 \end{array}$$

Suppose also that the user wishes to open a window on f, but the system does not contain a window rule for opening at f that preserves R directly. However, it is still possible to open a window at f if there is a rule for opening at f that preserves a relation S that is stronger than R. An appropriate window rule can be built by composing the rule preserving S with the fact that S is stronger than R as follows:

$$\frac{\dfrac{\gamma, \Gamma \vdash f'\ s\ f}{\Gamma \vdash F[f']\ S\ F[f]} \qquad \vdash (F[f']\ S\ F[f]) \supset (F[f']\ R\ F[f])}{\Gamma \vdash F[f']\ R\ F[f]}$$

Using this rule, the window can be opened as shown below:

$$\begin{array}{l} !\ \Gamma \\ !\ \gamma \\ s \star f \end{array}$$

The system stores information about the relative strength of relations together with inference rules for transforming theorems about stronger relations into theorems about weaker relations. When asked to open a window at a given position preserving a relation for which no specific window rule is stored, the system builds an appropriate substitute by composing a window rule preserving a stronger relation with an inference rule to weaken the relation.

Basic Rules

Expressions in the HOL system can be described by the following grammar:

$$\begin{array}{lll} E ::= v & & \text{— variable} \\ \quad \mid\ c & & \text{— constant} \\ \quad \mid\ E_f\ E_a & & \text{— application} \\ \quad \mid\ \lambda v.\, E & & \text{— abstraction} \end{array}$$

It can be seen from the grammar that there are only three positions within an expression at which a subwindow can be opened. These appear below, along with the window rules that justify opening at that position.

- on the operator of an application:

$$\frac{\Gamma \vdash E'_f = E_f}{\Gamma \vdash (E'_f\ E_a) = (E_f\ E_a)}$$

- on the operand of an application:

$$\frac{\Gamma \vdash E'_a = E_a}{\Gamma \vdash (E_f\ E'_a) = (E_f\ E_a)}$$

- on the body of an abstraction:[4]

$$\frac{\Gamma \vdash E' = E[v'/v]}{\Gamma \vdash (\lambda v'.\, E') = (\lambda v.\, E)}[v' \text{ not free in } \Gamma \text{ or } \lambda v.\, E]$$

  Note that when opening a subwindow on the body of an abstraction, the bound variable of the abstraction may need to be renamed to one that does not occur free in the assumptions of the window. The renaming is necessary because the variables in the assumptions of a window and the variables inside the abstraction are not in the same scope.[5]

These three window rules can be composed to reach any subexpression of a focus. Also, each of the rules preserves equality, which is as strong as any reflexive relation. Therefore, when used in conjunction with the techniques presented in earlier in this section, these rules are sufficient to implement a basic window inference system. More rules would be added to such a system to allow the use of contextual information, and the direct preservation of weaker, more general, relations.

In addition to the three basic rules, the window inference interface to the HOL system comes with thirty-seven window rules. These rules allow, among other things, users to exploit the contextual information contained in expressions involving logical connectives, conditional expressions, and applications of λ-abstractions. The rules also allow transformations that preserve equality or logical implication. Some users of the window inference interface have added their own rules to preserve other relations, including $\leq$ and $\geq$ on the natural numbers and refinement relations between programs.

[4]The notation $E[e/v]$ denotes substitution of e for the variable v in the expression E, while renaming bound variables to avoid capture.

[5]The current version of the window inference interface to the HOL system does not perform any renaming. It simply filters out from the subwindow those assumptions that contain the bound variable. While not the most general solution, this approach seems to work well in practice.

Choosing Rules

The techniques just described for combining and substituting window rules mean that whenever we wish to open a subwindow, we have to make a choice of which window rule(s) to use; or rather, the system has to make a choice for us. For example, consider opening a subwindow on the expression A in the following window.

$$\begin{array}{l} \quad ! \; \Gamma \\ \supset \star \; \boxed{A} \wedge B \end{array}$$

With the basic set of window rules supplied with the window-inference interface to the HOL system, the interface must make a choice between the following rules:[6]

$$\frac{B, \Gamma \vdash A' \supset A}{\Gamma \vdash (A' \wedge B) \supset (A \wedge B)}$$

$$\frac{\dfrac{B, \Gamma \vdash A' = A}{\Gamma \vdash (A' \wedge B) = (A \wedge B)} \qquad \vdash ((A' \wedge B) = (A \wedge B)) \supset ((A' \wedge B) \supset (A \wedge B))}{\Gamma \vdash (A' \wedge B) \supset (A \wedge B)}$$

$$\frac{\dfrac{\dfrac{\Gamma \vdash A' = A}{\Gamma \vdash (\wedge \; A') = (\wedge \; A)}}{\Gamma \vdash (A' \wedge B) = (A \wedge B)} \qquad \vdash ((A' \wedge B) = (A \wedge B)) \supset ((A' \wedge B) \supset (A \wedge B))}{\Gamma \vdash (A' \wedge B) \supset (A \wedge B)}$$

When choosing between rules, the interface uses the following heuristics in decreasing order of priority:

- The interface will choose to use an 'atomic' rule rather than compose several 'smaller' rules together. The motivation for this decision is that the atomic rule is more likely to be specifically designed for the situation at hand, while smaller rules are more likely to be generic in nature.
- The interface will choose a rule that preserves a weak (more general) relation in the child window in preference to a rule that preserves a stronger one.
- The interface will choose whichever of two rules that would result in the child window having more assumptions.

Experience with the interface shows that these heuristics perform the task of choosing appropriate window rules well.

[6] Note that we regard the infix notation for the conjunction $A \wedge B$ as syntactic sugar for the following standard prefix function application: $\wedge \; A \; B$. The expression $\wedge \; A$ denotes the partial application of a conjunction to one argument, A.

2.5 Lemmas

Window inference lends itself to the natural management of lemmas. When transforming a focus it is possible to use any theorem with hypotheses that are a subset of the assumptions of the window. Such theorems are said to be *applicable* to the window. The system maintains a set of theorems called $\mathcal{L}$. The conclusions of the applicable theorems in $\mathcal{L}$ are displayed along with the assumptions of the window as an additional prompt for the user. These extra components are called *lemmas*. To distinguish lemmas from the assumptions, they will be prefixed with '¡'. This visual distinction is artificial as the two may be used interchangeably. The term *context* will be used to refer to both assumptions and lemmas.

Suppose $\mathcal{L}$ is the set $\{(A \vdash B)\}$ and we want to simplify the expression $A \wedge B$. We begin by creating the following window.

$$= \star\ A \wedge \boxed{B}$$

theorem:
$\vdash (A \wedge B) = (A \wedge B)$
$\mathcal{L} = \{(A \vdash B)\}$

By opening a window on B, we enter a context where the theorem $A \vdash B$ is applicable.

$$\begin{array}{r} ! \ A \\ ¡ \ B \\ = \star\ B \end{array}$$

theorem:
$A \vdash B = B$
$\mathcal{L} = \{(A \vdash B)\}$

Using the lemma, we may transform the focus as follows:

$$\begin{array}{r} ! \ A \\ ¡ \ B \\ = \star\ \mathsf{T} \end{array}$$

theorem:
$A \vdash \mathsf{T} = B$
$\mathcal{L} = \{(A \vdash B)\}$

Closing the window, we return to the parent window below:

$$= \star\ A \wedge \mathsf{T}$$

theorem:
$\vdash (A \wedge \mathsf{T}) = (A \wedge B)$
$\mathcal{L} = \{(A \vdash B)\}$

The focus of this window may be simplified as follows:

$$= \star\ A$$

theorem:
$\vdash A = (A \wedge B)$
$\mathcal{L} = \{(A \vdash B)\}$

This series of inferences establishes the theorem $\vdash A = (A \wedge B)$.

Creating New Lemmas

If the user wishes to add a lemma L to the context, a theorem with conclusion L and hypotheses that are a subset of the context must be added to $\mathcal{L}$. One way to obtain the required theorem is to open the following window, which is used to create a backward proof of L in the current context:

$!\ \Gamma$ — assumptions of the parent window
$¡\ \Lambda$ — lemmas of the parent window
$\supset \star L$

After transforming the focus to true the user will have proved the theorem $\Gamma \vdash \mathsf{T} \supset L$, which can be simplified to $\Gamma \vdash L$. This theorem can then be added to $\mathcal{L}$ and the child window closed. The result is that L becomes a lemma in the original window.

Windows on the Context

Often a user will want to make use of a fact that follows indirectly from the context of a window. This is supported by allowing the user to derive new lemmas by opening windows on terms in the context. When a user opens a window on a term in the context, that term becomes the focus of the child window. The context of the parent window is available in the child. The user can then transform the focus of the child window to derive a new fact. Upon closing the child window, this fact is added to the set of relevant theorems, $\mathcal{L}$. Any theorem added in this way is applicable in the context of the parent window.

For example consider the following window:

$!\ A$	theorem:
$!\ \boxed{A \supset B}$	$A \supset B, A \vdash B = F_0$
$= \star B$	$\mathcal{L} = \{\}$

We could open a window on the assumption $A \supset B$.

$!\ A$	theorem:
$!\ A \supset B$	$A \supset B, A \vdash (A \supset B) \subset (A \supset B)$
$\subset \star A \supset B$	$\mathcal{L} = \{\}$

Note that we are now preserving the relation '$\subset$'—implication in the reverse direction. That is, we are trying to find something that follows from the focus. Using the assumption A, this window may be transformed to:

$!\ A$	theorem:
$!\ A \supset B$	$A \supset B, A \vdash B \subset (A \supset B)$
$\subset \star B$	$\mathcal{L} = \{\}$

We now close the child window. Note that the right hand side of the $\subset$ in the theorem held by such a child window always follows from the context and therefore the theorem may be simplified as follows:

$$\dfrac{\dfrac{A \supset B, A \vdash B \subset (A \supset B)}{A \supset B, A \vdash (A \supset B) \supset B}}{A \supset B, A \vdash B}$$

This theorem is added to $\mathcal{L}$. Returning to the parent window, there is a new lemma that will help us complete the proof:

$$\begin{array}{l} \;!\; A \\ \;!\; A \supset B \\ \;¡\; B \\ = \star\, B \end{array} \qquad \begin{array}{l} \text{theorem:} \\ A \supset B, A \vdash B = F_0 \\ \mathcal{L} = \{(A \supset B, A \vdash B)\} \end{array}$$

This technique could be described as a form of resolution by hand. The implementation of window inference for the HOL system could be extended to allow automatic resolution with selected assumptions and lemmas. However, even if such a facility were to be provided, it would still be useful to open windows on the context to derive new facts; either because the user wishes to maintain a total control over the reasoning, or because the derivation of the desired fact requires more complex reasoning than can be automated.

2.6 Conjectures

Window inference provides a natural environment for working with suppositions. The supposition that were the premises Γ to hold, then so would C will be written as '$\Gamma \stackrel{?}{\vdash} C$'. Each window in the system has associated with it a set of suppositions $\mathcal{C}$. Normally the set is the same for each window; but as we shall see later, it is sometimes convenient to remove a supposition from a window. When transforming a focus, it is possible to use any supposition with premises that are a subset of the context of the window. Such suppositions are said to be *applicable* in the context of the window. The notation for describing a window is extended so that the conclusions of the applicable suppositions in $\mathcal{C}$ are written along with the assumptions and lemmas of the window. These extra components are referred to as *conjectures*. To distinguish conjectures from assumptions and lemmas, the conjectures are prefixed with '?'. Conjectures are also considered part of the context of a window.

If a conjecture is used to transform the focus of a window, then it will become a hypothesis of the theorem held by the window. Such a conjecture is said to be *used*. Used conjectures are prefixed with '$\underline{?}$'. It is not permissible to close a window that uses a conjecture that is not present in the parent window. This is because the theorem held by the child window is used to transform the parent. If the theorem is

not applicable in the parent window, then the transformation can not take place. This does not mean that such a window may never be closed, only that before closing the window the offending conjecture must be proved. Conjectures that must be proved before the current window is closed are prefixed with '$\not?$'. Once a conjecture is proved, it is removed from the hypotheses of the theorem held by the window and the theorems in $\mathcal{L}$. The supposition responsible for the conjecture is removed from $\mathcal{C}$ and the theorem that establishes its truth is added to $\mathcal{L}$. That is, the conjecture becomes a lemma. Another option, not supported by the current implementation, would be for the user to review the premises of the conjecture and, if appropriate, delete or discharge some, thereby making the conjecture available in the parent window.

Note that a supposition that has no premises will be applicable in every context and the system will never force the user to prove such a supposition. However, using a supposition and not proving it will result in an additional hypothesis in the theorem resulting from the window inference.

Suppose we wanted to simplify the focus of the window below. Let us also suppose that the assumptions of this window are not available in its parent.

$$\begin{array}{l} ! \; A \supset B \\ ! \; \neg A \supset B \\ ! \; B \supset C \\ = \star \; C \end{array} \qquad \begin{array}{l} \text{theorem:} \\ B \supset C, \neg A \supset B, A \supset B \vdash C = F \\ \mathcal{L} = \{\} \\ \mathcal{C} = \{\} \end{array}$$

If B were true we could infer that the focus were true. Furthermore, it seems clear that B is true, so we decide to make it a conjecture under the current assumptions.

$$\begin{array}{l} ! \; A \supset B \\ ! \; \neg A \supset B \\ ! \; \boxed{B \supset C} \\ ? \; B \\ = \star \; C \end{array} \qquad \begin{array}{l} \text{theorem:} \\ B \supset C, \neg A \supset B, A \supset B \vdash C = F \\ \mathcal{L} = \{\} \\ \mathcal{C} = \{(B \supset C, \neg A \supset B, A \supset B \not\vdash B)\} \end{array}$$

We open a window on the assumption $B \supset C$.

$$\begin{array}{l} ! \; A \supset B \\ ! \; \neg A \supset B \\ ! \; B \supset C \\ ? \; B \\ \subset \star \; B \supset C \end{array} \qquad \begin{array}{l} \text{theorem:} \\ B \supset C, \ldots, A \supset B \vdash (B \supset C) \subset (B \supset C) \\ \mathcal{L} = \{\} \\ \mathcal{C} = \{(B \supset C, \neg A \supset B, A \supset B \not\vdash B)\} \end{array}$$

The focus may be simplified using the conjecture: (Note that the prefix of the conjecture changes to '$\underline{?}$' to indicate its use, and that the used conjecture becomes a hypothesis of the theorem held by the window.)

$$
\begin{array}{l}
\quad ! \; A \supset B \\
\quad ! \; \neg A \supset B \\
\quad ! \; B \supset C \\
\quad \underline{?} \; B \\
\subset \star \; C
\end{array}
$$

theorem:
$B, B \supset C, \ldots, A \supset B \vdash C \subset (B \supset C)$
$\mathcal{L} = \{\}$
$\mathcal{C} = \{(B \supset C, \neg A \supset B, A \supset B \mathrel{?\!\!-} B)\}$

When we close the window, a new lemma is created by simplifying the theorem held by the child window and adding it to $\mathcal{L}$. Note that the prefix of the conjecture reverts to '?' as it has not been used in this window.

$$
\begin{array}{l}
\quad ! \; A \supset B \\
\quad ! \; \neg A \supset B \\
\quad ! \; B \supset C \\
\quad ? \; B \\
\quad ¡ \; C \\
= \star \; C
\end{array}
$$

theorem:
$B \supset C, \neg A \supset B, A \supset B \vdash C = F$
$\mathcal{L} = \{(B, B \supset C, \neg A \supset B, A \supset B \vdash C)\}$
$\mathcal{C} = \{(B \supset C, \neg A \supset B, A \supset B \mathrel{?\!\!-} B)\}$

Using the new lemma, the focus may be transformed to true. Note that using the lemma entails using the conjecture since the conjecture was used in the derivation of the lemma. Note also that since the conjecture is not available in the parent window, we must prove it before closing this window.

$$
\begin{array}{l}
\quad ! \; A \supset B \\
\quad ! \; \neg A \supset B \\
\quad ! \; B \supset C \\
\quad \underline{\underline{?}} \; \boxed{B} \\
\quad ¡ \; C \\
= \star \; \mathsf{T}
\end{array}
$$

theorem:
$B, B \supset C, \neg A \supset B, A \supset B \vdash \mathsf{T} = F$
$\mathcal{L} = \{(B, B \supset C, \neg A \supset B, A \supset B \vdash C)\}$
$\mathcal{C} = \{(B \supset C, \neg A \supset B, A \supset B \mathrel{?\!\!-} B)\}$

We therefore open a window to establish the truth of B. This is done in the same way as creating a new lemma (see section 2.5). We remove the supposition that B is true under the current assumptions from $\mathcal{C}$ in the child window since it can not usefully contribute to its own proof.

$$
\begin{array}{l}
\quad ! \; A \supset B \\
\quad ! \; \neg A \supset B \\
\quad ! \; B \supset C \\
\supset \star \; B
\end{array}
$$

theorem:
$B \supset C, \neg A \supset B, A \supset B \vdash B \supset B$
$\mathcal{L} = \{(B, B \supset C, \neg A \supset B, A \supset B \vdash C)\}$
$\mathcal{C} = \{\}$

We now transform B to true. The details of the transformation, which proceeds by a case-analysis on A, are omitted.

$$
\begin{array}{l}
\quad ! \; A \supset B \\
\quad ! \; \neg A \supset B \\
\quad ! \; B \supset C \\
\supset \star \; \mathsf{T}
\end{array}
$$

theorem:
$B \supset C, \neg A \supset B, A \supset B \vdash \mathsf{T} \supset B$
$\mathcal{L} = \{(B, B \supset C, \neg A \supset B, A \supset B \vdash C)\}$
$\mathcal{C} = \{\}$

Closing the window, the theorem below is added to $\mathcal{L}$ and the corresponding conjecture removed from $\mathcal{C}$.

$$B \supset C, \neg A \supset B, A \supset B \vdash B$$

Using this theorem, B can be removed from the hypotheses of the theorem held by the parent window and the theorems in $\mathcal{L}$:

$$\begin{array}{l} \quad ! \; A \supset B \\ \quad ! \; \neg A \supset B \\ \quad ! \; B \supset C \\ \quad ¡ \; B \\ \quad ¡ \; C \\ = \star \, \mathsf{T} \end{array}$$

theorem:
$B, B \supset C, \neg A \supset B, A \supset B \vdash \mathsf{T} = F$
$\mathcal{L} = \left\{ \begin{array}{l} (B \supset C, \neg A \supset B, A \supset B \vdash C), \\ (B \supset C, \neg A \supset B, A \supset B \vdash B) \end{array} \right\}$
$\mathcal{C} = \{\}$

Windows may be opened to derive new lemmas from conjectures in the same way as described for lemmas and assumptions in section 2.5. The theorem added to $\mathcal{L}$ when such a window is closed will have the conjecture as one of its hypotheses. The conclusion of such a theorem is available as a lemma in the parent window because the conjecture is available there. Using such a lemma entails using the conjecture from which it was derived.

2.7 Implementation

The window inference system as described in this chapter has been implemented as an interface to the HOL theorem prover. The implementation is available free of charge to noncommercial users, and is distributed as a library with the HOL system. The library is accompanied by a complete reference manual [Gru91a] which forms part of the HOL system's documentation [SRI91].

Implementing window inference within a mature theorem prover brings some advantages over the direct implementation approach used by Staples and Robinson with their tool, Demo2. The user gains immediate access to the existing tools in that theorem prover for rewriting, checking tautologies, etc., as well as the ability to move between window inference and the other styles of reasoning supported by the host system. Implementing window inference in an LCF-style [Pau87] system like HOL, where theorems are represented by a secure data-type, brings the further advantage that inconsistent results cannot be obtained through errors in the interface.

2.8 Related Work

The concepts of window inference were first described by John Staples and Peter Robinson [RS93]. The inspiration for building a window inference interface for HOL

came from using their proof checker, Demo2 [TRS90]. The main features of window inference as described by Staples and Robinson are the ability to transform an expression by focusing attention on a subexpression and the ability to make use of contextual information while concentrating on a subexpression.

Francisco Corrella [Cor93] and Leonard Monk [Mon88] have investigated the general problem of using contextual information in a proof. The ability to make limited use of contextual information when performing automatic simplifications can be found in several theorem provers.

The ability to focus attention on a subexpression may also be found in several theorem provers, e.g. Matt Kaufmann's PC-NQTHM interface [Kau88] to the Boyer-Moore theorem prover [BM79, BM88]. An as yet unreleased version of PC-NQTHM also allows the user access to some of the information that follows from the context of the subexpression. For example, using PC-NQTHM to simplify the expression $A \wedge B \wedge C$, the user may 'dive' to the subexpression B and in that context may assume A (but not C).

What distinguishes the work described in this chapter from the previous work is the combination of these features with the ability to preserve relations other than equivalence. This feature is not particularly useful for ordinary theorem proving applications, which may explain why it has been ignored in the past. Its value lies in the fact that it enables a new kind application for theorem provers as refinement systems.

2.9 Applications

The following chapters develop a method of program refinement based on window inference. However, window inference has many other potential applications. This section presents a brief survey of the work of others who have taken the ideas of this chapter as were they first presented [Gru91c] and applied them to their own problems.

While at the University of Cambridge, Andrew Gordon performed some brief but encouraging experiments using window inference to transform hardware designs phrased in the ELLA and Silage hardware description languages [Gor93]. Gordon was using the HOL system to prove the correctness of transformations to hardware designs phrased in these languages. He found that window inference offered a natural user interface for performing such correctness preserving transformations.

Mats Larsson's Licentiate thesis at the University of Linköping presents a transformation method for designing digital systems [Lar93]. The method allows a stepwise refinement of specifications into implementations. The design process is formalised using the HOL system. Window inference is a key component of Larsson's method and the examples of actual design transformations contained in his thesis have been performed using the window-inference interface to the HOL system described in this

chapter.

Alf Smith of the Defence Research Agency has examined the application of window inference to reasoning about specifications phrased in the Z specification language [Smi92]. Smith presents window inference as part of a suite of techniques which could be used in combination to make reasoning about Z specifications more tractable. In particular, Smith advocates the construction of a proof tool for Z containing an implementation of window inference (based on the techniques described here) that would allow users to make the same kinds of reasoning steps they are used to making with pen and paper.

In the section entitled 'Future Work' of the original paper on window inference in HOL [Gru91c], it was suggested that window inference could form the basis of a simple graphic user interface to the HOL system. While in Cambridge, Laurent Théry of INRIA Sophia Antipolis constructed a graphic front-end for the window-inference interface [Thé93]. The front-end was constructed using the Centaur tool [BKT94, TBK92]. Users of the graphic front-end may open subwindows on terms in the focus or context of a window by selecting the desired subterm with the mouse. A variety of transformations, like rewriting, can be applied to a window by selecting them from a menu. Users can write their own transformations and have them included in the menu. Any parameters of a transformation can be entered either via the keyboard or by selecting terms with the mouse. For example, a user can rewrite the focus of a window with selected elements of the context by selecting 'rewrite' from the menu, and then highlighting the desired elements of the context.

The later chapters of this dissertation describe a refinement method based on a combination of window inference and a novel formalisation of a simple programming language. However, window inference can also be used with more traditional approaches to program refinement. The refinement calculus is a theory of program transformations developed by Ralph Back [Bac81], and which has been formalised in HOL by Back and Joakim von Wright [BvW90]. In some recent and encouraging work, von Wright has used window inference to perform refinement on programs according to the laws of this calculus [vW94, vWHLL93]. This work exploits the fact that the window inference system described here can be tailored to preserve any preorder. In this case, window inference was used to preserve the refinement relation between predicate transformers.

2.10 Conclusions

This chapter described a general method of transformational reasoning, called window inference. Window inference has the following strengths:

- It supports users who wish to solve problems by concentrating on one subproblem at a time. This is important when the overall problem is complex.

- It allows users to make arbitrarily fine manipulations of an expression. This is important when the form of the solution is as important as the existence of the solution itself, as is the case when deriving programs from specifications.
- It allows the user to make use of the information implicit in the context of the subproblem under examination.
- It allows the use of suppositions during a proof development. A supposition must be proved before using any results that depend on it.
- The window inference system described here can be tailored to support the preservation of any preorder when transforming an expression. This is essential for the application of window inference to refinement.

A correspondence between window inference and natural deduction has been outlined. This correspondence serves both to reassure the reader that window inference is a valid and practical style of reasoning, and as a guide to implementing a window-inference interface to a natural-deduction based theorem prover, like HOL. A window-inference interface to the HOL system has already been built based on these ideas [Gru91a].

The window inference system described here was inspired by the desire to build a program refinement system. Goal-directed reasoning has been popular for reasoning about the development of correct implementations of specifications by verification. It is, however, unsuited to the task of reasoning about development by refinement. The following chapters demonstrate that window inference is a natural style of proof for reasoning about developing correct implementations by refinement.

Window inference is also well suited to goal-directed reasoning. It may prove advantageous to use window inference instead of, or in combination with, tactic based reasoning for goal-directed proof in the following circumstances:

- when a proof may be progressed by simplifying a subproblem, perhaps a non-boolean expression, which would never be independently treated as a goal by a tactic based system;
- when contextual information plays an important part in the progress of a proof;
- when part of a proof can be developed more easily by forward proof than backward proof.

To this end, the window-inference interface to the HOL system contains tactics which allow window inference to be combined with tactical proof.

Chapter 3

Predicative Programming—A Survey

3.1 Introduction

The main goal of this dissertation is to develop a method of program refinement by combining a unified treatment of programs and specifications with the window-inference method of transformational reasoning. Traditional approaches to program refinement create a unified specification and programming language by adding a specification statement to an existing programming language. The resulting language is then given a semantics, usually in terms of Dijkstra's weakest preconditions [Dij76]. This dissertation uses an alternative approach based on the work of Mili [Mil83], Hoare [Hoa84] and Hehner [Heh84]. The idea is to use a specification language that is expressive enough to capture the semantics of the target programming language. The constructs of the programming language can then be mixed freely with expressions from the specification language. A program is simply a specification composed entirely of program constructs.

The original intent of this work was to use predicates as the specification language. A prototype refinement tool was developed based on this approach and presented at *The 5th Refinement Workshop* [Gru92]. Unfortunately, refinements developed with this tool guaranteed only partial correctness.[1] This survey is the result of attempts to find a stronger predicative formalism that would support refinements guaranteeing total correctness.

The idea of using predicates to specify behaviour is a compelling one. The specification languages HP-SL [BR91] and Z [Hay87] are based on this idea. In addition, predicates of higher-order logic are often used for the specification and verification of hardware [Gor85, HD85]. Predicates have proved popular for specification because of the clarity of the resulting specifications, and because predicates and the predicate calculus are well understood logical concepts.

[1] Partial correctness is correctness under the assumption of termination.

Many of us believe that we share a common intuition about what a specification phrased as a predicate means. It may therefore be surprising to learn that there are several ways to interpret a predicate as a specification. Under these various interpretations, the same predicate can describe different behaviours. This chapter is a survey of the existing approaches to the specification of behaviour and programming language semantics using predicates. It highlights the difficulties with these approaches, particularly as they relate to describing termination. These different approaches are often given the same name. This chapter gives them new names so that we may distinguish between them. The following chapter proposes an alternative approach which is subsequently used to form the basis of a refinement method that guarantees total correctness. If they wish, readers may bypass this chapter and proceed straight to the next after reading section 3.2, which explains some basic notation and terminology.

The material presented in this chapter was originally presented as a paper at *The International Conference on Formal Methods in Programming and Their Applications* [Gru93a].

3.2 The Basic Concepts

Before the various interpretations of predicates as specifications can be described, some basic concepts and terminology must be introduced.

state: The term *state* is used to refer to the state of a computer. The state is a vector of program variables, for example (w, x, y, z), which we refer to as σ. The set of all states is denoted by 'Σ'.

initial state: The *initial state* of a computer is the state before a computation begins. We denote the value of program variable x in the initial state by $`x$, and the entire initial state by $`\sigma$.

final state: The *final state* of a computer is the state that it is left in after a computation finishes. We denote the value of program variable x in the final state by x', and the entire final state by σ'.

nontermination: When a computation finishes it is said to have *terminated*. The term *nontermination* is used to describe a computation that never finishes.

outcome: The *outcome* of a computation is either the final state in which it terminates, or nontermination. The symbol $\bot$ is introduced to denote nontermination. The set of all outcomes, $\Sigma_\bot$, is the disjoint union of the set of all states and nontermination:

$$\Sigma_\bot \stackrel{\text{def}}{=} \Sigma + \{\bot\}$$

behaviour: For any computation, the *behaviour* of a computer can be characterised by the initial state in which the computation was begun and the outcome of the computation. Behaviours are therefore elements of $\Sigma \times \Sigma_\perp$.

specification: A *specification* is a description of the desired behaviours of a computation. A specification is a function from behaviours to booleans, that is of type $(\Sigma \times \Sigma_\perp) \to BOOL$. We say that a behaviour b *meets* a specification S, if S applied to b is true.

We will sometimes give informal specifications as English descriptions of desired behaviour. For example, the following text can be interpreted as a specification:

> The computation should terminate in a final state in which z' is equal to $\grave{}x \times \grave{}y$.

nondeterminism: If for any initial state only one behaviour meets a specification, then the specification is said to be *deterministic*. If, however, several behaviours could meet a specification, then that specification is said to be *nondeterministic*. The more possible behaviours that satisfy a specification, the 'less deterministic' the specification is said to be.

If we are attempting to prove a property of some nondeterministic system, we should assume that it behaves *demonically*. If a specification is nondeterministic and only some of the possible behaviours are ones that we want, then a demonic interpretation of the specification always chooses one of the behaviours that we *do not* want.

The opposite of demonic nondeterminism is *angelic* nondeterminism. If a specification is nondeterministic and only some of the possible behaviours are ones that we want, then an angelic interpretation of the specification always chooses one of the behaviours that we *do* want.

refinement: We say that a specification T is a *refinement* of another specification S, if any behaviour that meets T also meets S. The relationship 'S is refined by T' is written $S \sqsubseteq T$.

$$\models^{\text{def}} (S \sqsubseteq T) = (\forall b.\, (T\ b) \supset (S\ b)) \qquad [\sqsubseteq\text{\_DEF}]$$

From this definition we can see that refinement is the process of reducing nondeterminism in a specification.

Consider the following informal specifications S and T:

S: The computation should terminate in a state where x' is odd.

T: The computation should terminate in a state where x' is a prime number greater than 2.

The specification S is refined by T. We know this, because all prime numbers greater than 2 are odd.

Basic Specifications

This section presents four 'basic' specifications that, for any initial state, represent the extreme values of the set of possible outcomes.

chaos

For some initial state, a specification allows *chaotic* (i.e. completely arbitrary) behaviour if the set of allowable outcomes is the set of all outcomes.

$$\models^{\text{def}} \mathsf{chaos}(i, o) = \mathsf{T} \qquad \text{[chaos\_DEF]}$$

The specification chaos is refined by any specification.

chance

For some initial state, a specification allows *chance* (i.e. arbitrary but terminating) behaviour if the set of allowable outcomes is precisely the set of all states.

$$\models^{\text{def}} \mathsf{chance}(i, o) = (o \neq \bot) \qquad \text{[chance\_DEF]}$$

abort

For some initial state, a specification requires behaviour that never terminates, or *aborts*, if the set of allowable outcomes contains only nontermination.

$$\models^{\text{def}} \mathsf{abort}(i, o) = (o = \bot) \qquad \text{[abort\_DEF]}$$

miracle

For some initial state, a specification can never be satisfied (or requires *miraculous* behaviour) if the set of allowable outcomes is empty.

$$\models^{\text{def}} \mathsf{miracle}(i, o) = \mathsf{F} \qquad \text{[miracle\_DEF]}$$

(In some work, the term 'magic' is used for the specification we call miracle.)

A miracle is a refinement of every specification. Note that it is not possible to implement miracle.

In some work the term 'abort' is used to describe what has just been called chaos. This is invariably done in the context of an interpretation that does not distinguish between nonterminating behaviour and chaotic behaviour. We keep these concepts separate to allow discussion of various interpretations.

Predicates

Predicates are boolean-valued expressions which may contain free variables, possibly from the initial and final states $`\sigma$ and $\sigma'$. This allows predicates to describe a relationship between these states. This relationship can be made explicit by regarding the predicate $P$ as a shorthand for the function $\lambda(`\sigma, \sigma').P$ which is of type $(\Sigma \times \Sigma) \to BOOL$. Whenever it is not ambiguous, we will use predicates to stand for both their boolean values and functions from pairs of states to boolean values. For example, the predicate F stands for both the truth value F, and the constant function $\lambda(`\sigma, \sigma').\mathsf{F}$.

The interpretation of a predicate as a specification is a function from the domain of predicates to the domain of specifications, that is, a function of type:

$$\underbrace{((\Sigma \times \Sigma) \to BOOL)}_{\text{Predicates}} \to \underbrace{((\Sigma \times \Sigma_\perp) \to BOOL)}_{\text{Specifications}}$$

There are many possible interpretations of predicates as specifications, but only a few preserve any of our intuitions about what a predicate should mean as a specification. These few can be distinguished by the different ways in which nontermination is introduced into the interpretation.

The most obvious interpretation is the one that does not allow the description of nonterminating behaviour. Under this interpretation, any initial state i and outcome o will meet the specification described by P, if o is not $\perp$, and i and o satisfy P. Unfortunately since programs do not always terminate, this interpretation is unsuitable for describing programming language semantics. This chapter is concerned only with those interpretations that can describe nonterminating behaviour.

A Simple Language

This section describes a small imperative programming language. Later sections give this language a semantics using the various predicative models. Once each programming language construct has been given a meaning in terms of a predicate, the constructs may be mixed freely with other predicate notations. The result is that predicates can be treated as a wide-spectrum language where we can both phrase specifications and develop implementations.

Although any predicate can be interpreted as a specification, only a restricted category of expressions are considered executable. The restriction is syntactic, which means that executable specifications are easily recognised by both humans and machines. We do not give a complete definition of which expressions are executable. For example, the decision as to whether an expression like '$\#l$', denoting the length of the list l, is executable is left to the reader. One might reasonably expect that expressions like '$1 + 2$' are executable, while expressions like '$\int_a^b f\ x\ dx$' are not. Note that in order to be considered executable, all variables in an expression must be drawn from the initial state.

We now describe the constructs of our programming language and the conditions under which they are executable.

skip

The skip construct does nothing; it specifies behaviour that terminates with the same final state as the initial state. The skip construct is always executable.

$v' := e$

The parallel assignment construct $x' := e$ assigns a vector of expressions e to a vector of variables x', and leaves the other variables of the final state with the values they held initially. The assignment $x' := e$ is executable if all the expressions in e are executable and no variable occurs on the left-hand side more than once.

if b **then** P **else** Q

The conditional construct behaves according to P if the condition b is true. Otherwise, it behaves in accordance with Q. The conditional is executable if b, P and Q are executable. Note that although b is a boolean-valued expression it should not be interpreted as a specification.

$P \mid Q$

The choice construct makes an arbitrary choice between P and Q, and then behaves according to that specification. The choice $P \mid Q$ is executable if both P and Q are executable.

$P; Q$

The sequential composition $P; Q$ behaves according to the specification P and then (if this behaviour terminates) according to Q. The composition $P; Q$ is executable if both P and Q are executable.

var $x \,.\, P$

The var construct is used to introduce a local variable. The new variable x exists only within P. The block **var** $x \,.\, P$ is executable if P is executable. Note that within the scope of the var construct, the state space Σ should be extended to include the new variable.

label $l \,.\, P$

The label construct is used for creating recursively defined blocks. Within the body of the block, P, any instance of the label l should be interpreted as a recursive call to the start of the block. For example, suppose x is a natural number; the following program decrements x until it reaches 0:

$$\textbf{label}\ l.\ \textbf{if}\ \grave{}x = 0\ \textbf{then skip else}\ (x' := \grave{}x - 1;\ l(\grave{}\sigma, \sigma'))$$

The block **label** $l \,.\, P$ is executable if assuming l is executable (when applied to the current initial and final states), then P is executable.

A Note on Nondeterministic Specifications

It is sometimes assumed that the demonic choice between any specification and one requiring nontermination should always choose the one requiring nontermination. Suppose, however, we have a specification that requires nontermination, for example abort, and a potential implementation is proposed that allows a nondeterministic choice between some terminating behaviour and nontermination, for example skip | abort. Clearly, in this case the demonic choice is to choose the specification that allows terminating behaviour.

In general, a demonic choice between two specifications should choose the least refined of the two. However, not every set of specifications has a least refined element. We do know that the specification chaos is less refined that any other. Therefore the demonic choice between any specification and one that permits chaotic behaviour should choose the specification that permits chaotic behaviour. We also know that the specification miracle is more refined than any other. Therefore the demonic choice between a specification that requires miraculous behaviour and any other specification should choose the other specification.

3.3 The Relational Model

This section begins the examination of the various interpretations of predicates as specifications. We call the first interpretation we examine the *relational* model. The relational model is used by the HP-SL [BR91] and Z [Hay87] specification languages. Mili's work on program development also uses the relational model [Mil83, MXYQ86, MDG86]. More recently Gravell [Gra92] and Sekerinski [Sek92] have developed refinement methods based on this model. Nguyen has proposed a semantics for Dijkstra's guarded-command language using the relational model [Ngu91].

Description

Consider a predicate P. The following is an interpretation of P as a specification using the relational model.

> Given any initial state, if there exists a final state satisfying P, then terminate in such a state. If there is no final state that satisfies P, any behaviour (including nontermination) is acceptable.

Let us denote the interpretation of the predicate P as a specification using the relational model by '$[\![P]\!]_r$'. When formalised, this interpretation is equivalent to the definition below:

$$\stackrel{\text{def}}{=} [\![P]\!]_r(i,o) = ((\exists \sigma.\, P(i,\sigma)) \supset (o \neq \bot) \wedge P(i,o)) \qquad [[\![\_]\!]_r\text{\_DEF}]$$

A refinement relation between predicates in the relational model ($\sqsubseteq_r$) follows immediately from their interpretation.

$$\models^{\text{def}} (P \sqsubseteq_r Q) = (\llbracket P \rrbracket_r \sqsubseteq \llbracket Q \rrbracket_r) \qquad [\sqsubseteq_r \text{\_DEF}]$$

A more direct definition of refinement would be easier to work with. Consider the choices available to us when attempting to satisfy the specification described by P. We have greatest choice when, for a particular initial state, there is no final state such that P holds—we can terminate in any state, or even fail to terminate. We can therefore refine a specification by increasing the number of initial states for which some final state will satisfy the predicate. When, for a particular initial state, there are several final states that satisfy P, we may terminate in any of those final states. Therefore, we can also refine a specification by decreasing the number of final states that will satisfy the predicate for any given initial state. Accordingly, the refinement relation between predicates in the relational model can be expressed as follows:

$$\begin{array}{l} \vdash (P \sqsubseteq_r Q) = \\ \quad \forall \grave{}\sigma.\, (\exists \sigma'.\, P(\grave{}\sigma, \sigma')) \supset \\ \qquad (\exists \sigma'.\, Q(\grave{}\sigma, \sigma')) \wedge (\forall \sigma'.\, Q(\grave{}\sigma, \sigma') \supset P(\grave{}\sigma, \sigma')) \end{array} \qquad [\sqsubseteq_r \text{\_THM}]$$

That is, for any initial state, predicates are ordered by the implication ordering; except for those predicates that are false. Those predicates are placed at the bottom of the ordering.

Basic Specifications

Let us examine how the basic specifications of section 3.2 are treated in the relational model:

chaos
: The specification chaos is represented by the predicate F.

$$\vdash \mathsf{chaos} = \llbracket \mathsf{F} \rrbracket_r$$

chance
: The specification chance is represented by the predicate T.

$$\vdash \mathsf{chance} = \llbracket \mathsf{T} \rrbracket_r$$

abort
: There is no way to describe abort using the relational model.

$$\vdash \neg \exists P.\, \mathsf{abort} = \llbracket P \rrbracket_r$$

miracle
: There is no way to describe miracle using the relational model.

$$\vdash \neg \exists P.\, \mathsf{miracle} = \llbracket P \rrbracket_r$$

The Simple Language

We now examine each of the programming language constructs described in section 3.2 to see how they are represented in the relational model:

skip
: The skip command is easily expressed by requiring that the final state is equal to the initial state.

$$\mathsf{skip} \stackrel{\text{def}}{=} (\sigma' = \grave{}\sigma)$$

$x' := e$
: Assignment can be expressed by requiring that the final state is equal to the initial state with the appropriate substitutions.

$$(x' := e) \stackrel{\text{def}}{=} (\sigma' = \grave{}\sigma)[e/\grave{}x]$$

This semantics extends to parallel assignment by regarding x' and e as vectors.

if b then P else Q
: The conditional construct is expressed as a choice between two mutually exclusive alternatives.

$$(\mathsf{if}\ b\ \mathsf{then}\ P\ \mathsf{else}\ Q) \stackrel{\text{def}}{=} (b \wedge P) \vee (\neg b \wedge Q)$$

The meaning for this construct could be given using a conditional expression, but this is the form traditionally used with the relational model.

$P \mid Q$
: One might think that the nondeterministic choice between P and Q can be expressed by $P \vee Q$. Unfortunately, for any initial state if either P or Q would allow chaotic behaviour, this semantics would angelically choose the other specification. The meaning below forces a demonic interpretation of the choice between P and Q.

$$P \mid Q \stackrel{\text{def}}{=} (\exists \sigma'.\, P) \wedge (\exists \sigma'.\, Q) \wedge (P \vee Q)$$

$P; Q$
: At first sight, it might appear that the composition of two specifications, P and Q, can be expressed as relational composition $\exists \sigma.\, P[\sigma/\sigma'] \wedge Q[\sigma/\grave{}\sigma]$. Unfortunately, this semantics requires the interpretation of P to angelically choose a final state from which Q will not allow chaotic behaviour. The meaning below gives a demonic interpretation to the composition by explicitly allowing chaotic

behaviour whenever P may terminate in a state from which Q allows chaotic behaviour.

$$P;\ Q \stackrel{\text{def}}{=} (\forall \sigma.\ P[\sigma/\sigma'] \supset (\exists \sigma'.\ Q[\sigma/\grave{}\sigma])) \wedge (\exists \sigma.\ P[\sigma/\sigma'] \wedge Q[\sigma/\grave{}\sigma])$$

$\mathsf{var}\, x.\, P$

To introduce a local variable, we must provide a quantified scope for its initial and final values. At first it might appear that the statement $\mathsf{var}\, x.\, P$ should be defined as $\exists \grave{}x\ x'.\ P$. Unfortunately, this semantics is angelic. If there are some initial values of x such that P describes chaotic behaviour, then this meaning will (if possible) angelically avoid choosing one of those values. The correct meaning for $\mathsf{var}\, x.\, P$ must demonically allow chaotic behaviour whenever any initial value of x could result in chaotic behaviour.

$$(\mathsf{var}\, x.\, P) \stackrel{\text{def}}{=} (\forall \grave{}x.\ \exists \sigma'\ x'.\ P) \wedge (\exists \grave{}x\ x'.\ P)$$

$\mathsf{label}\ l.\, P$

It is not always possible to give a meaning for a recursively defined block. However, if the body of the block represents a monotonic function of the label[2] (as is the case if the block is executable), then we can express the meaning as the least fixed-point of that function.[3]

$$\mathrm{mono}(\lambda l.\, P) \stackrel{\text{def}}{\Rightarrow} (\mathsf{label}\ l.\, P) = \mathrm{fix}(\lambda l.\, P)$$

Note that both 'mono' and 'fix' are ordered with respect to $\sqsubseteq_r$.

A Note on Nontermination in the Relational Model

A possible misconception about the relational model is that the specification F represents abort, the program that fails to terminate. Suppose that F does specify the program that fails to terminate. The definition of refinement in section 3.2 tells us that one specification refines another if every behaviour that meets the first specification also meets the second one. The definition of refinement used with the relational model allows us to refine F into T (or chance), the program that terminates in an arbitrary state. Behaviours which meet the specification chance do not meet the

[2] The function f is said to be *monotonic* with respect to the relation $\sqsubseteq$ if the following property holds: $\forall x\ y.\ (x \sqsubseteq y) \supset \big((f\ x) \sqsubseteq (f\ y)\big)$.

[3] A *fixed point* of a function, f, is a value, X, such that $f\ X = X$. The *least fixed-point* of a function, denoted '$\mathrm{fix} f$', is the least of all such fixed points with respect to the relation $\sqsubseteq$. The Tarski-Knaster theorem states that any monotonic function on a domain where any monotonically increasing chain has a least upper-bound, has a least fixed-point.

specification abort, because such behaviours terminate while abort requires nontermination. Therefore, we know that F does not represent abort; it represents chaos, the specification that allows completely arbitrary behaviour. Refining F into T simply reduced the nondeterminism by eliminating the possibility of nontermination.

3.4 The Partial Model

The next model to be examined will be called the *partial* model. The partial model can establish only the partial correctness of one specification with respect to another. Despite this failing, the partial model has been extensively used for the specification and verification of hardware [Gor85, HD85] (where the problem analogous to avoiding nontermination is avoiding short-circuits). The partial model has also been applied to software. David Park combined the partial model with a separate characterisation of termination to come up with a method of describing the semantics of programming languages [Par79]. Gordon has formalised Hoare logic [Hoa69] using the partial model [Gor88]. The partial model was used in the prototype refinement tool built during the earlier stages of the research described in this dissertation [Gru92].

Description

Consider a predicate P. The following is an interpretation of P as a specification using the partial model.

> For any initial state, either choose a final state satisfying P and terminate in that state, or fail to terminate. If there is no final state that satisfies P, then nontermination is the only acceptable behaviour.

The interpretation of the predicate P as a specification according to the partial model is written '$[\![P]\!]_p$'. This interpretation is formalised as follows:

$$\vdash^{\text{def}} [\![P]\!]_p(i, o) = P(i, o) \vee (o = \bot) \qquad [[\![\_]\!]_p\text{\_DEF}]$$

Under this interpretation, termination in a state that does not satisfy the predicate is the only incorrect behaviour. Nontermination always meets the specification.

The definition of refinement between predicates in the partial model ($\sqsubseteq_p$) follows directly from their interpretation.

$$\vdash^{\text{def}} (P \sqsubseteq_p Q) = ([\![P]\!]_p \sqsubseteq [\![Q]\!]_p) \qquad [\sqsubseteq_p\text{\_DEF}]$$

A direct definition of refinement would be of more practical use. Refinement is the process of reducing nondeterminism. A specification is refined by reducing the number of outcomes that satisfy the specification for any given initial state. Refinement of predicates in the partial model can therefore be expressed as follows:

$$\vdash (P \sqsubseteq_p Q) = (\forall `\sigma\ \sigma'.\ Q(`\sigma, \sigma') \supset P(`\sigma, \sigma')) \qquad [\sqsubseteq_p\text{\_THM}]$$

That is, the predicates are ordered by implication; with false at the top of the order, and true at the bottom.

Basic Specifications

Here we consider the treatment of the basic specifications of section 3.2 in the partial model:

chaos
: The predicate T describes chaos. Termination in any final state satisfies T and, under the partial model, nontermination also satisfies T.

$$\vdash \mathsf{chaos} = [\![\mathsf{T}]\!]_p$$

chance
: The specification chance cannot be represented in the partial model.

$$\vdash \neg\exists P.\, \mathsf{chance} = [\![P]\!]_p$$

abort
: The specification abort requires nontermination, which is described by the predicate F in the partial model.

$$\vdash \mathsf{abort} = [\![\mathsf{F}]\!]_p$$

miracle
: The specification miracle can not be represented in the partial model.

$$\vdash \neg\exists P.\, \mathsf{miracle} = [\![P]\!]_p$$

However, nontermination has a miracle like property in the partial model, because abort is a refinement of every specification that can be represented in this model.

The Simple Language

We now examine the definition of the programming language constructs, described in section 3.2 under the partial model.

skip
: Strictly speaking, skip cannot be expressed in this model because skip is required to terminate. If we ignore termination, we can capture the rest of the meaning by requiring that the final and initial states are the same.

$$\mathsf{skip} \stackrel{\text{def}}{=} (\sigma' = \grave{}\sigma)$$

$x' := e$

Assignment requires termination and therefore can not be properly expressed in the partial model. If we ignore termination, we can capture the rest of the meaning as follows:

$$(x' := e) \stackrel{\text{def}}{=} (\sigma' = \grave{}\sigma)[e/\grave{}x]$$

if b then P else Q

The conditional construct can be naturally expressed in the partial model by using implication:

$$(\textsf{if } b \textsf{ then } P \textsf{ else } Q) \stackrel{\text{def}}{=} ((b \supset P) \wedge (\neg b \supset Q))$$

Note that this definition is equivalent to the one used in the relational model. They are expressed differently because of the different conventions associated with the models.

$P \mid Q$

Choice is simply expressed as disjunction:

$$P \mid Q \stackrel{\text{def}}{=} P \vee Q$$

Since nontermination is always acceptable in the partial model, there is no need to explicitly allow for it as was done when giving the relational semantics of choice.

$P; Q$

Sequential composition is expressed in the partial model as follows:

$$P; Q \stackrel{\text{def}}{=} (\exists \sigma.\, P[\sigma/\sigma'] \wedge Q[\sigma/\grave{}\sigma])$$

As with choice, the potential for nontermination does not complicate the definition of composition in the partial model.

var $x \,.\, P$

A local variable is introduced simply by providing an existentially quantified scope for the initial and final values.

$$(\textsf{var } x.\, P) \stackrel{\text{def}}{=} (\exists \grave{}x\ x'.\, P)$$

$\mathsf{label}\ l.\ P$

It is not always possible to capture the meaning of a recursive block. However, if the body of the block is a monotonic function of the label then the meaning can be expressed as a fixed point of that function. It is usual to associate the meaning of a recursive function with its least-fixed point. However, in the partial model, the meaning of a recursive function is more accurately captured by the greatest fixed-point.

Consider the following simple recursive block: $\mathsf{label}\ l.\ l$. The body of this block is the identity function. The least-fixed point of the identity function is T, the predicate that denotes chaotic behaviour. However, no invocation of this block can ever terminate, so F—the predicate that denotes nonterminating behaviour—is a more accurate meaning. The predicate F is the greatest fixed-point of the identity function.

Accordingly we define the meaning of a recursive block in the partial model as follows (we use the notation '$\overline{\mathrm{fix}}\, f$' to denote the greatest fixed-point of the function f with respect to some ordering).

$$\mathrm{mono}(\lambda l.\ P) \stackrel{\mathrm{def}}{\vdash} (\mathsf{label}\ l.\ P) = \overline{\mathrm{fix}}(\lambda l.\ P)$$

Note that both 'mono' and '$\overline{\mathrm{fix}}$' are ordered with respect to $\sqsubseteq_p$.

A Note on Nondeterminism in the Partial Model

Note that it is sometimes thought that the meaning assigned to nondeterministic choice by the partial model is angelic because it is possible to prove the following equivalence:

$$\vdash P \mid \mathsf{abort} = P$$

This is not the case. Under the partial model nontermination always satisfies a specification. Therefore, explicitly allowing a choice between satisfying a specification or failing to terminate does not allow any more possible behaviours than the specification itself did.

3.5 The Total Model

The next model to be examined we call the *total* model. The total model has been used by Hoare [Hoa84, HH87] and Hehner [Heh84, HGM86] to specify behaviour and program semantics. This model is more commonly known as the 'predicate' model. We will not use this name since we wish to use the term 'predicative' to describe all the models.

Description

As with the previous models, we consider the interpretation of a predicate P as a specification, this time under the total model.

> For any initial state, if every final state would satisfy P, then we assume that the specifier cares so little about the behaviour that not only is termination in any final state acceptable, but so is nontermination. If, however, not every final state would satisfy P, then we must choose some state that does and terminate in it.

The interpretation of the predicate P as a specification according to the total model is written '$[\![P]\!]_t$'. When formalised, this interpretation is equivalent to the definition below:

$$\models^{\text{def}} [\![P]\!]_t(i,o) = ((\neg\forall\sigma.\, P(i,\sigma)) \supset ((o \neq \bot) \wedge P(i,o))) \qquad [[\![\_]\!]_t\text{\_DEF}]$$

As before, the definition of refinement in this model ($\sqsubseteq_t$) follows directly from the interpretation of predicates as specifications.

$$\models^{\text{def}} (P \sqsubseteq_t Q) = ([\![P]\!]_t \sqsubseteq [\![Q]\!]_t) \qquad [\sqsubseteq_t \text{\_DEF}]$$

Refinement can be seen as the process of reducing nondeterminism. A specification is refined by reducing the number of outcomes that satisfy the specification for any given initial state. Therefore, refinement in the total model may also be expressed as follows:

$$\vdash (P \sqsubseteq_t Q) = (\forall `\sigma\, \sigma'.\, Q(`\sigma, \sigma') \supset P(`\sigma, \sigma')) \qquad [\sqsubseteq_t \text{\_THM}]$$

That is, the predicates are ordered by implication; with false at the top of the order, and true at the bottom.

Basic Specifications

We now examine the basic specifications of section 3.2 using the total model.

chaos
: The predicate T represents chaos.

$$\vdash \mathsf{chaos} = [\![\mathsf{T}]\!]_t$$

chance
: It is not possible to represent chance in this model. This is because T, the only predicate that allows termination in any final state, also allows nontermination.

$$\vdash \neg\exists P.\, \mathsf{chance} = [\![P]\!]_t$$

abort
: It is not possible to represent abort in this model. This is because T, the only predicate that allows nontermination, also allows termination in any final state.

$$\vdash \neg\exists P.\, \mathsf{abort} = [\![P]\!]_t$$

miracle
: The predicate F represents the miracle in the total model, it is impossible to satisfy.

$$\vdash \mathsf{miracle} = [\![\mathsf{F}]\!]_t$$

The Simple Language

We now examine each of the programming constructs described in section 3.2 to see how they are defined in the total model:

skip
: The skip command is easily expressed by requiring that the final state is equal to the initial state.

$$\mathsf{skip} \stackrel{\text{def}}{=} (\sigma' = \grave{}\sigma)$$

$x' := e$
: Parallel assignment can be expressed by stating that the final state is equal to the initial state with the appropriate substitutions.

$$(x' := e) \stackrel{\text{def}}{=} (\sigma' = \grave{}\sigma)[e/\grave{}x]$$

if b then P else Q
: The conditional construct can be naturally expressed in the total model by using implication.

$$(\mathsf{if}\ b\ \mathsf{then}\ P\ \mathsf{else}\ Q) \stackrel{\text{def}}{=} ((b \supset P) \wedge (\neg b \supset Q))$$

$P \mid Q$
: Choice is simply expressed as disjunction:

$$P \mid Q \stackrel{\text{def}}{=} P \vee Q$$

This semantics for choice is demonic because T represents chaotic behaviour in the total model.

Note that this semantics has one curious feature. Consider the following meanings for the predicates P and Q:

P: even x'
This requires behaviour that terminates in a state where x is even.

Q: odd x'
This requires behaviour that terminates in a state where x is odd.

Since we know that all numbers are either even or odd, the nondeterministic choice between P and Q is equivalent to the predicate T (which is equivalent to chaos in the total model). This means that the nondeterministic choice between two specifications both of which require terminating behaviour may allow nontermination.

$P; Q$
One might think that the sequential composition of two specifications should be represented in the total model as follows: $\exists \sigma.\, P[\sigma/\sigma'] \wedge Q[\sigma/\grave{}\sigma]$. Suppose, however, for some initial state, P specifies chaotic, i.e. potentially nonterminating, behaviour. Suppose also that for all initial states, Q specifies some terminating behaviour. This meaning for composition incorrectly requires that the composition of these two specification always terminates. The following meaning for composition avoids this problem:

$$P; Q \stackrel{\text{def}}{=} ((\neg \forall \sigma'.\, P) \supset (\exists \sigma.\, P[\sigma/\sigma'] \wedge Q[\sigma/\grave{}\sigma]))$$

At first sight this meaning might be thought to allow the same kind of undesirable behaviour that was explicitly banned in the relational model. Suppose P allowed termination in some set of intermediate states, and suppose that for only some of those intermediate states does there exists a final state that satisfies Q. This meaning for composition requires the interpretation of P to choose one of those intermediate states for which there is a final state that satisfies Q. In the relational model this was an angelic choice because it avoided potentially chaotic behaviour. In the total model, however, this choice is demonic because it avoids a requirement for miraculous behaviour.

Note that this meaning for sequential composition has a flaw in that the sequential composition of two specifications, both of which require termination, may allow nonterminating behaviour. This problem means that sequential composition is not always associative in the total model. The example below illustrates the problem:

$$\begin{array}{ll} & (x' \neq \grave{}x; x' \neq \grave{}x); x' = 0 \\ = & \mathsf{T}; x' = 0 \\ = & \mathsf{T} \end{array} \qquad \begin{array}{ll} & x' \neq \grave{}x; (x' \neq \grave{}x; x' = 0) \\ = & x' \neq \grave{}x; x' = 0 \\ = & x' = 0 \end{array}$$

$\mathsf{var}\, x \,.\, P$

As in the partial model, we introduce a local variable by simply providing an existentially quantified scope for the initial and final values.

$$(\mathsf{var}\, x \,.\, P) \stackrel{\text{def}}{=} (\exists \grave{x}\ x' \,.\, P)$$

Note that in the total model it is possible for a predicate that describes terminating behaviour to allow nontermination when it is enclosed in a var construct. For example, $\mathsf{var}\, x \,.\, x' = \grave{x}$ allows nonterminating behaviour, where $x' = \grave{x}$ does not.

$\mathsf{label}\ l \,.\, P$

It is not always possible to give a meaning for a recursively defined block. However, if the body of the block represents a monotonic function of the label (as is the case if the block is executable), then we can express the meaning as the least fixed-point of that function.

$$\mathrm{mono}(\lambda l \,.\, P) \stackrel{\text{def}}{\vdash} (\mathsf{label}\ l \,.\, P) = \mathrm{fix}(\lambda l \,.\, P)$$

Note that both 'mono' and 'fix' are ordered with respect to $\sqsubseteq_t$.

3.6 Comparison

This section contains a comparison of the various models. In particular, it highlights the relative difficulty of using the various models for specifying behaviour and proving properties of programs. We shall also highlight areas where the models are unusual or run counter to our expectations.

The Refinement Ordering

The table below shows the definition of the refinement ordering for each of the models.

	$P \sqsubseteq Q$
Relational	$\forall \grave{\sigma} . (\exists \sigma' . P(\grave{\sigma}, \sigma')) \supset ((\exists \sigma' . Q(\grave{\sigma}, \sigma')) \wedge (\forall \sigma' . Q(\grave{\sigma}, \sigma') \supset P(\grave{\sigma}, \sigma')))$
Partial	$\forall \grave{\sigma}\ \sigma' . Q(\grave{\sigma}, \sigma') \supset P(\grave{\sigma}, \sigma')$
Total	$\forall \grave{\sigma}\ \sigma' . Q(\grave{\sigma}, \sigma') \supset P(\grave{\sigma}, \sigma')$

If we wish to use a predicative model as the basis of a refinement technique, and in particular a refinement technique where we are not restricted to using preproved refinement laws, then the simpler the definition of our refinement relation, the simpler our proof obligations will be. From this perspective the simplicity of the refinement relation as defined in the partial and total models gives them a clear advantage over

the relational model. If we wish to make a refinement in the relational model that is not an instance of some preproved class of refinements, then we will have to perform an existence proof.

In addition to the simplicity of the partial and total models, these models support the intuitive association between correctness and logical implication.

The Basic Specifications

The table below gives the predicates used to describe each of the basic specifications in the various models. Note again that we use abort to mean nontermination while some authors use 'abort' to stand for the concept we have called chaos.

	miracle	abort	chance	chaos
Relational			T	F
Partial		F		T
Total	F			T

The predicate F is used to describe miracle in the total model. This supports an intuitive association between the predicate which is impossible to satisfy and the specification which is impossible to meet. Note that while the partial model can not describe miracle, the specification abort (again represented by F) fills an analogous role in that model. The relational model cannot describe miracle. The inability to describe miracles is not a disadvantage in itself, since miracles cannot be implemented. However, various authors have argued that the presence of miracles in a calculus leads to a simpler and more powerful formalism [Mor88a, Mor88b, Nel89].

The partial model describes abort with the predicate F. Neither of the other models can describe behaviour that is required not to terminate. The assumption of users of the relational and total models is that nobody ever requires nontermination. This being the case, abort becomes just another kind of undesirable behaviour and its role can be subsumed by that of chaos.

The relational model describes chance with the predicate T. Neither of the other models can describe arbitrary but terminating behaviour. In the partial model this is not possible because every specification is satisfied by nontermination. Users of the total model make the assumption that if a specifier does not care in which state a program terminates, then the specifier also does not care whether the program terminates at all.

All three models can describe chaos. By describing chaos with the predicate T, the partial and total models support the intuitive association between the specification which is trivial to meet and the predicate which is trivial to satisfy.

The Simple Language

In this section we examine those programming language constructs that are treated differently by the different models.

The Choice Construct

The table below sets out the meaning for the nondeterministic choice construct in each model.

	$P \mid Q$
Relational	$(\exists \sigma' . P) \wedge (\exists \sigma' . Q) \wedge (P \vee Q)$
Partial	$P \vee Q$
Total	$P \vee Q$

Because chaos is represented by F in the relational model we must explicitly ensure that nondeterministic choice behaves demonically. In the other models, where chaos is represented by T, demonic behaviour is the default. This results in a simpler meaning for choice in the partial and total models, which in turn should result in simpler proof obligations for programs involving choice. The partial and total models support the intuitive association held by many people between choice and logical or.

Unfortunately, in the total model, the nondeterministic choice between two specifications each requiring termination may sometimes allow nontermination. This problem was illustrated in section 3.5.

Sequential Composition

The table below gives the meaning assigned to sequential composition in each of the models.

	$P; Q$
Relational	$(\forall \sigma . P[\sigma/\sigma'] \supset (\exists \sigma' . Q[\sigma/\grave{}\sigma])) \wedge (\exists \sigma . P[\sigma/\sigma'] \wedge Q[\sigma/\grave{}\sigma])$
Partial	$\exists \sigma . P[\sigma/\sigma'] \wedge Q[\sigma/\grave{}\sigma]$
Total	$(\neg \forall \sigma' . P) \supset (\exists \sigma . P[\sigma/\sigma'] \wedge Q[\sigma/\grave{}\sigma])$

Again the meaning in the relational model is complicated by the need to explicitly ensure that the construct behaves demonically. The meaning in the total model is slightly complicated by the fact that if a specification allows termination in any state, then it also allows nontermination. Also, this meaning is not associative, as we expect it should be (this problem was noted in section 3.5).

The meaning of composition used in the partial model is the simplest, and supports the intuitive association between sequential composition and composition of relations. As with the other constructs, a simpler meaning for sequential composition leads to simpler proof obligations when proving properties of programs.

Local Variables

The table below sets out the meaning for a local variable declaration in each of the models.

	$\mathsf{var}\ x.P$
Relational	$(\forall `x.\exists \sigma'\ x'.P) \wedge (\exists `x\ x'.P)$
Partial	$\exists `x\ x'.P$
Total	$\exists `x\ x'.P$

The meaning in the relational model must explicitly ensure that the construct behaves demonically. For example, suppose the for some initial values of x the predicate P is always false. The expression $\exists `x\ x'.P$ avoids, if possible, choosing such values for $`x$. In the relational model where F corresponds to chaos, this would represent an angelic choice. In the other models where F represents $\mathsf{miracle}$, this is the demonic choice.

Note that in the total model, as examined in section 3.5, enclosing a predicate that describes a terminating behaviour in a var construct may result in a predicate that describes potentially nonterminating behaviour.

General Recursion

In the table below, we set out the meaning given to recursive blocks in each of the models. These meanings are valid only if the bodies of the blocks are monotonic functions. This is the case if the blocks are executable, but may not be the case otherwise. Hence, the meanings given below are not a useful starting place for developing refinement techniques for general recursion. (Such a technique is outlined in chapter 5.)

	$\mathsf{label}\ l.P$
Relational	$\mathrm{fix}(\lambda l.P)$
Partial	$\overline{\mathrm{fix}}(\lambda l.P)$
Total	$\mathrm{fix}(\lambda l.P)$

Both the relational and total models define recursion as a least fixed-point. The Partial model uses the unusual construction of a greatest fixed-point because nontermination is the most defined specification that can be represented in this model. We could also have modelled recursion in the partial model with a least fixed-point. This would equate infinite recursion with chaotic behaviour as is done in the other models. However, by using a greatest fixed-point we can be more accurate and equate infinite recursions with nonterminating behaviour.

Extremes

In order to highlight some curiosities, it is illustrative to examine what happens as we approach extreme specifications. Suppose that we have two integer-valued constants **min** and **max**, such that **min** is less than **max**.

Approaching the Impossible

Consider the following predicate:

$$\mathbf{min} \leq x' \leq \mathbf{max}$$

For any initial state this predicate describes a set of final states. In all three models this predicate represents a specification that allows termination in any state where the value of x lies between **min** and **max**. In the partial model, nontermination also meets the specification. If we move **min** and **max** closer together, we reduce the number of final states in which termination is acceptable. If we continue this process until the set described contains only one value for x then we must choose that value. Something curious happens if we move **min** and **max** further so that they pass each other and the predicate describes an empty set. In the total model no behaviour meets the specification. Similarly, in the partial model no terminating behaviour meets the specification. In the relational model, however, any behaviour including nontermination will now meet the specification. This sudden change represents a 'discontinuity' in the meaning of predicates in the relational model.

Approaching the Trivial

Consider the following predicate which is similar to the one used above:

$$\neg(\mathbf{min} \leq x' \leq \mathbf{max})$$

For any initial state this predicate describes all those final states where the value of x does *not* lie between **min** and **max**. Under all three interpretations, termination in any of those states is acceptable. Again, the partial model also permits nontermination. If we move **min** and **max** closer together then we increase the number of final states in which termination is acceptable. This process continues smoothly up to and including the point at which there is only one unacceptable state. As before, something curious happens if we move **min** and **max** further so that they overlap and the specification describes the universal set of final states. In both the relational and partial models the meaning is simply that termination in any state is now acceptable. As always, the partial model also permits nontermination. In the total model, however, this last step alters the meaning of the specification so that for the first time nontermination is also an acceptable behaviour. This sudden change represents a discontinuity in the meaning of predicates in the total model.

There are no discontinuities in the partial model because nontermination remains possible throughout the entire range of specifications.

3.7 More-Elaborate Models

The models examined in this chapter so far have been very simple. This simplicity is a large aspect of their appeal. However, each of these simple models is, in some way, less than perfect. This section gives a brief outline of the other, more complex, predicative models that have been devised to more accurately describe the behaviour of a computer.

The Extended Total Model

In more recent work [HH86], Hoare extends the total model by regarding predicates as functions on states extended with a distinguished element $\perp$, denoting nontermination. This avoids the problem with nondeterminism in the total model highlighted in section 3.5. For example, if P is again the predicate 'even x'' which describes behaviour that terminates in a state where x is even, and Q is the predicate 'odd x'' describing behaviour that terminates in a state where x is odd, then $P \vee Q$ describes any terminating behaviour. The predicate T still describes chaos in the extended total model, but $P \vee Q$ is no longer equal to T since the value of x is neither even nor odd in the state $\perp$.

While this extension corrects the defects in the naive version of the total model, it does so at the cost of added complexity. For example, the extended total model makes use of a three-valued logic. It is this logic that allows predicates like 'even $x' \vee$ odd x'' to be false for undefined values of x'. Also, part of the appeal of the predicative models is that the state is just a distinguished set of variables, and that no further differentiation is made between logic variables and program variables. This uniformity breaks down in the extended total modal where the state may be either a collection of values or $\perp$. Program variables in the extended total model are perhaps best thought of as functions from the state to their value. These functions are undefined if the state is $\perp$.

The Timed Model

Hehner [Heh89, Heh90] proposes a different approach based on an extension to the partial model. Hehner regards proofs of termination of being of questionable value. Instead he extends the state with a distinguished variable which represents the current time. Specifications can then be extended with requirements on the duration of a computation. The treatment of the timing variable slightly complicates the expression of programming language semantics in this model; however the primary reason it was not adopted for use in this dissertation is that its presentation contains no formal semantics for the recursive construct used in its simple programming language.

3.8 Conclusions

In this chapter we have examined the various interpretations of predicates as specifications. (Readers interested in a comparison of these interpretations with approaches based on pairs of predicates are referred to [Heh88].) In doing so we have shown there is no commonly held understanding of what a predicate means as a specification of behaviour. Some aspects of these models are simpler and support more of our intuitions about programs than others. We can see, however, that each of these models has its failings.

The relational model chooses to describe those primitive specifications that are of practical importance, namely chance and chaos. Perhaps this explains why the relational model has proved such a popular basis for specification languages. It was also the relational model that gave the sample programming language the most complex semantics and had the most complex refinement relation. This may explain why many formal developments begun with a specification phrased in the relational model are completed only after re-expressing the specification in another formalism.

One of the difficulties of specifying large systems is avoiding an inconsistent specification. However, the consequences of making this error are much more serious in the relational model where any behaviour will vacuously satisfy an inconsistent specification. In the total model, no behaviour will satisfy an inconsistent specification. Similarly, in the partial model no terminating behaviour will satisfy an inconsistent specification.

The partial model gave the sample programming language the simplest semantics. Unfortunately this model is unable to support proofs of total correctness. This may explain the unpopularity of this model as the basis of a refinement technique. It may also explain why this model is considered well suited to program verification [Gor88] where the separation of the partial correctness proof and the termination proof is a welcome division of the task of proving total correctness.

The total model preserves much of the simplicity of the partial model while supporting proofs of total correctness, but this model is not without its difficulties. Its basic assumption that if a specifier does not care in which state a computation terminates, then they also do not care if the computation terminates at all is not universally shared [Par85]. Certainly the consequences of this assumption can be counter to our intuitions; in particular—as illustrated in section 3.5—the nonassociativity of sequential composition, and the fact the nondeterministic choice between two terminating behaviours can allow nontermination are most unwelcome.

The various difficulties with the predicative models examined in this survey have meant that while predicates have proved popular for use in specification and even for verification, they have been almost universally abandoned in favour of specifications phrased as pairs of predicates for work in program refinement. The next chapter proposes an interpretation of predicates in a three-valued logic that works around these problems while maintaining much of the appeal of the predicative styles.

Chapter 4

A Three-Valued Logic for Refinement

4.1 Introduction

This chapter introduces a simple three-valued logic. Indeed, it might be more accurate to say that it introduces a new type of three-valued booleans and some functions on that type, which are then reasoned about in ordinary two-valued logic. The predicates of this three-valued logic can be used to describe potentially nonterminating behaviour more clearly than was possible using the various interpretations of two-valued predicates discussed in the previous chapter. They can also be used to give a semantics to the simple imperative programming language introduced in the previous chapter that is as simple as any of those possible using two-valued predicates.

Unlike the extended total model described in section 3.7, which implicitly used an 'internal' three-valued logic; the work presented in this chapter uses three-valued logic explicitly. The resulting simplifications mean that, like the simpler predicate models, no real distinction between logical variables and program variables need be made. We also avoid the need to extend all functions to correctly handle undefined arguments; only the logical connectives need be extended.

The material presented in this chapter was originally presented as a paper at *The International Conference on Formal Methods in Programming and Their Applications* [Gru93b].

4.2 A Three-Valued Logic

We wish to define a three-valued logic for reasoning about computations. Various systems of three-valued logics were proposed in the 1920s and 1930s, the most notable being those of Bochvar [Boch38], Kleene [Kle38] and Łukasiewicz [Luk20b, Luk20a]. More recently, logics in which every type—not just the booleans—is extended with an additional value have been developed for reasoning about partial functions. The most

well known of these are Jones's 'Logic of Partial Functions' [BCJ84] and Scott's 'Logic of Computable Functions' [SS71]. Scott's logic could perhaps be more accurately described as a two-valued logic for reasoning about types extended with an extra element.

This section introduces a three-valued logic for reasoning about programs—only the boolean type is extended with a distinguished value. The three-valued logic presented here is similar to Bochvar's internal logic. It is also similar to Scott's logic in that it is really a two-valued logic for reasoning about three-valued booleans.

Let us begin by defining the three-valued booleans $BOOL_3$. The elements of $BOOL_3$ denote the truth values of the logic: false, true and undefined.

$$BOOL_3 \stackrel{\text{def}}{=} \mathsf{F}_3 \mid \mathsf{T}_3 \mid \bot_3$$

We define an injection $\lfloor \_ \rfloor$ from $BOOL$ into $BOOL_3$.

$$\models^{\text{def}} (\lfloor \mathsf{T} \rfloor = \mathsf{T}_3) \wedge (\lfloor \mathsf{F} \rfloor = \mathsf{F}_3) \qquad [\lfloor \_ \rfloor \text{\_DEF}]$$

We also partially define a function $\lceil \_ \rceil$ mapping values from $BOOL_3$ into $BOOL$.

$$\models^{\text{def}} (\lceil \mathsf{T}_3 \rceil = \mathsf{T}) \wedge (\lceil \mathsf{F}_3 \rceil = \mathsf{F}) \qquad [\lceil \_ \rceil \text{\_DEF}]$$

A 'strict' version of each logical connective is defined as follows. If any argument of the connective is $\bot_3$, then the result is $\bot_3$; otherwise, the result is as it would have been in two valued logic. We denote such a translated connective by underlining the symbol that denotes it in two valued logic. For example, here are the truth tables for the three-valued versions of disjunction, implication, equality, and negation:

$A \underline{\vee} B$		B		
		$\bot_3$	T_3	F_3
A	$\bot_3$	$\bot_3$	$\bot_3$	$\bot_3$
	T_3	$\bot_3$	T_3	T_3
	F_3	$\bot_3$	T_3	F_3

$A \underline{\supseteq} B$		B		
		$\bot_3$	T_3	F_3
A	$\bot_3$	$\bot_3$	$\bot_3$	$\bot_3$
	T_3	$\bot_3$	T_3	F_3
	F_3	$\bot_3$	T_3	T_3

$A \underline{\equiv} B$		B		
		$\bot_3$	T_3	F_3
A	$\bot_3$	$\bot_3$	$\bot_3$	$\bot_3$
	T_3	$\bot_3$	T_3	F_3
	F_3	$\bot_3$	F_3	T_3

A	$\underline{\neg} A$
$\bot_3$	$\bot_3$
T_3	F_3
F_3	T_3

We retain the traditional 'strong' meaning for the undecorated equality symbol, namely that it is true if and only if the arguments are identical.

$A = B$		B		
		$\bot_3$	T_3	F_3
A	$\bot_3$	T	F	F
	T_3	F	T	F
	F_3	F	F	T

We also translate quantifiers into our three-valued logic. If for any value of the bound variable the predicate is $\bot_3$, then the result is $\bot_3$; otherwise, the result is as it would have been in two-valued logic.[1]

$$\models^{\text{def}} (\underline{\exists} v.\, P) = ((\exists v.\, P = \bot_3) \Rightarrow \bot_3 \mid \lfloor \exists v.\, \lceil P \rceil \rfloor) \qquad [\underline{\exists}\text{\_DEF}]$$

[1] The notation $B \Rightarrow X \mid Y$ denotes the conditional expression which equals X if B is true, and Y otherwise.

$$\models^{\mathsf{def}}(\underline{\forall} v.\, P) = ((\exists v.\, P = \bot_3) \Rightarrow \bot_3 \mid \lfloor \forall v.\, \lceil P \rceil \rfloor) \qquad [\underline{\forall}\text{-DEF}]$$

In two-valued logic, if a bound variable is of finite type, we can think of existential and universal quantification as forms of distributed disjunction and conjunction. These intuitions remain valid in our three-valued logic.

We now define two connectives unique to our three-valued logic. The first of these, $\supset\!\!\!)$, is a kind of conditional that is useful for phrasing specifications. It takes two arguments, a boolean and a three-valued boolean, and returns a three-valued boolean. If the left-hand side is true, the result is the value of the second argument. Otherwise, the result is undefined ($\bot_3$).

$A \supset\!\!\!) B$		B		
		$\bot_3$	T_3	F_3
A	T	$\bot_3$	T_3	F_3
	F	$\bot_3$	$\bot_3$	$\bot_3$

An example of the use of $\supset\!\!\!)$ can be found in section 4.3.

The second conective, $\between$, is a form of conjunction. It behaves like the normal three-valued conjunction except that when using $\between$ the conjunction of false and anything (including $\bot_3$) is false, the normal three-valued conjunction is strict.

$A \between B$		B		
		$\bot_3$	T_3	F_3
A	$\bot_3$	$\bot_3$	$\bot_3$	F_3
	T_3	$\bot_3$	T_3	F_3
	F_3	F_3	F_3	F_3

The $\between$ operator is useful for defining the semantics of sequential composition. An example of its use can be found in section 4.5.

4.3 Three-Valued Predicates as Specifications

Three-valued predicates are three-valued boolean expressions which may contain free variables, possibly from the initial and final states $`\sigma$ and $\sigma'$. This allows three-valued predicates to describe a relationship between these states. This relationship can be made explicit by regarding the three-valued predicate $P$ as a shorthand for the function $\lambda(`\sigma, \sigma').\, P$, which is of type $(\Sigma \times \Sigma) \to BOOL_3$. We call objects of this type *descriptions*. Whenever it is not ambiguous, we will use three-valued predicates to stand for both their values and their corresponding descriptions. For example, the predicate F_3 stands for both the value F_3 and the description $\lambda(`\sigma, \sigma').\, \mathsf{F}_3$.

The type of specifications, $(\Sigma \times \Sigma_\perp) \to BOOL$, is similar to the type of descriptions. Let us now consider an interpretation function, $[\![\_]\!]$, from descriptions to specifications:

$$[\![\_]\!] : \underbrace{((\Sigma \times \Sigma) \to BOOL_3)}_{\text{Descriptions}} \to \underbrace{((\Sigma \times \Sigma_\perp) \to BOOL)}_{\text{Specifications}}$$

The following is an informal interpretation of a description D as a specification:

> Given any initial state, choose a final state such that D is not F_3. (If there is no such state, then the specification requires a miracle.) If D is T_3, then terminate in that final state; otherwise, if D is $\perp_3$, either terminate in that final state, or fail to terminate.

When formalised, this interpretation is equivalent to the following definition:

$$\models^{\text{def}} [\![D]\!](i, o) = ((o \neq \perp) \Rightarrow (D(i, o) \neq \mathsf{F}_3) \,\mathbf{|}\, (\exists \sigma.\, D(i, \sigma) = \perp_3)) \qquad [[\![\_]\!]\text{\_DEF}]$$

Whenever it is not ambiguous, we identify descriptions with the specifications they denote. For example, consider the description below:[2]

$$\lfloor (\grave{}y > 0) \supset (z' = \grave{}x \div \grave{}y) \rfloor$$

We regard this description as denoting the informal specification:

> Given an initial state where $\grave{}y > 0$ holds, the computer should terminate in a state where $z' = \grave{}x \div \grave{}y$. If $\grave{}y > 0$ does not hold in the initial state, termination in any final state is acceptable.

As another example, consider the similar description:

$$(\grave{}y > 0) \supset\!\!\!\supset \lfloor z' = \grave{}x \div \grave{}y \rfloor$$

We regard this description as denoting the informal specification:

> Given an initial state where $\grave{}y > 0$ holds, the computer should terminate in a final state where $z' = \grave{}x \div \grave{}y$. If $\grave{}y > 0$ does not hold in the initial state, any behaviour (terminating or nonterminating) is acceptable.

[2] This dissertation follows the classical approach to dealing with potentially undefined terms like $x \div 0$. That is, we assume that such a term denotes a value of the appropriate type, and that it always denotes the same value.

Basic Specifications

We can now examine how the basic specifications from section 3.2 are modelled by three-valued predicates:

chaos
: The three-valued predicate $\bot_3$ represents completely arbitrary (potentially nonterminating) behaviour.

$$\vdash \mathsf{chaos} = [\![\bot_3]\!]$$

chance
: The three-valued predicate T_3 represents arbitrary but terminating behaviour.

$$\vdash \mathsf{chance} = [\![\mathsf{T}_3]\!]$$

abort
: It is not possible to describe abort, the specification that requires nontermination, with this interpretation of three-valued predicates. In practice this is not a great loss, since specifications that require nontermination are rarely of much use.

$$\vdash \neg\exists D.\, \mathsf{abort} = [\![D]\!]$$

miracle
: The three-valued predicate F_3 represents a miracle (the specification that is impossible to meet).

$$\vdash \mathsf{miracle} = [\![\mathsf{F}_3]\!]$$

This interpretation supports some pleasing intuitions. The predicate that is impossible to satisfy, F_3, represents the specification that is impossible to meet, miracle. The predicate that is trivial to satisfy, T_3, represents the specification that is met by any terminating behaviour, chance. The predicate $\bot_3$, traditionally associated with undefinedness, corresponds to the specification that admits completely arbitrary behaviour, chaos.

Redundancy

This interpretation of descriptions as specifications contains some redundancy. This redundancy can be illustrated by representing descriptions as sets of tuples. Suppose that we wish to represent the description D as a set. If D maps the pair $(`\sigma, \sigma')$ to $X$, then the tuple $((`\sigma, \sigma'), X)$ occurs in the set. Let us assume that we wish to

specify the behaviour of a simple machine with just two states s_1 and s_2. Consider the description represented by the set below:

$$\{((s_1, s_1), \bot_3), ((s_1, s_2), \mathsf{T}_3), ((s_2, s_1), \mathsf{T}_3), ((s_2, s_2), \mathsf{F}_3)\}$$

If we interpret this as a specification, then a computation commenced in the state s_1 may yield any of the following outcomes: termination in s_1, termination in s_2, or nontermination. Computations commenced in s_2 are required to terminate in s_1. The description below models the same specification:

$$\{((s_1, s_1), \bot_3), ((s_1, s_2), \bot_3), ((s_2, s_1), \mathsf{T}_3), ((s_2, s_2), \mathsf{F}_3)\}$$

While not desirable, this redundancy does not prove to be inconvenient.

4.4 Refining with Three-Valued Predicates

We wish to refine descriptions into programs in a language whose semantics is expressed with descriptions. It is easy to infer the refinement relation for descriptions.

$$\models^{\text{def}} (D \sqsubseteq_D D') = (\llbracket D \rrbracket \sqsubseteq \llbracket D' \rrbracket) \qquad [\sqsubseteq_D \text{\_DEF}]$$

Each step in the refinement process requires a proof that the new description is a refinement of its predecessor. As noted in the previous chapter, it is important that the refinement relation be as simple as possible in order to reduce the complexity of these proofs. Although $\sqsubseteq_D$ is the relationship we must preserve, its definition is more complex than we would like; though it appears simple, the complexity lies in the definition of $\llbracket \_ \rrbracket$. To avoid this complexity, we will now define a simpler refinement relation and shows why it is acceptable to preserve this relationship instead.

A Simple Refinement Relation

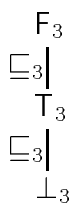
We recall from our definition of refinement in section 3.2 that refinement can be viewed as a process of reducing nondeterminism. We now restate the informal interpretation of a three-valued predicate P as a specification in order to introduce a simplified concept of refinement between three-valued predicates:

Given an initial state $`\sigma$:

1. Choose a state σ' such that P is not F_3.
2. If P is T_3, terminate in the state σ'.
   If P is $\bot_3$, either terminate in σ' or fail to terminate.

Let us consider refining the description P; in particular, refining the value of P at the point $(`\sigma, \sigma')$, leaving all other points unchanged. A value of F_3 offers the least choice in how to fulfil the specification as it would eliminate the possibility of choosing σ' at step 1. Failing that, a value of T_3 offers less alternatives than $\bot_3$, because while both T_3 and $\bot_3$ allow termination in σ', $\bot_3$ also allows nontermination. Therefore, our refinement ordering over the three-valued booleans $\sqsubseteq_3$ is that F_3 is better than T_3, which is better than $\bot_3$. The following truth table gives a complete definition of the refinement relation which is summarised by the accompanying diagram:

$A \sqsubseteq_3 B$		B		
		$\bot_3$	T_3	F_3
A	$\bot_3$	T	T	T
	T_3	F	T	T
	F_3	F	F	T



The ordering is just the natural lifting of the implication ordering over booleans. Implication and the refinement ordering are related as follows:

$$\vdash (B' \supset B) = (\lfloor B \rfloor \sqsubseteq_3 \lfloor B' \rfloor) \qquad [\supset\sqsubseteq_3\_\mathsf{THM}]$$

Our simplified refinement relation for descriptions bears the following obviously desirable relationship to the true refinement relation:

$$\vdash (\forall(`\sigma, \sigma').\, D(`\sigma, \sigma') \sqsubseteq_3 D'(`\sigma, \sigma')) \supset (D \sqsubseteq_D D') \qquad [\sqsubseteq_3\sqsubseteq_D\_\mathsf{THM}]$$

The following examples illustrate refinement with three-valued predicates:

- In this first refinement we make the specification more deterministic.

  $$\vdash \lfloor x' > `x \rfloor \sqsubseteq_3 \lfloor x' = `x + 1 \rfloor$$

- In the second refinement we increase the domain of guaranteed termination.

  $$\vdash (`x > 0 \supset \lfloor z' = `x \times `y \rfloor) \sqsubseteq_3 \lfloor z' = `x \times `y \rfloor$$

A Potential Problem

In the previous section we noted that it is possible to refine a description by refining it with respect to the simplified refinement relation $\sqsubseteq_3$.

$$\vdash (\forall(`\sigma, \sigma').\, D(`\sigma, \sigma') \sqsubseteq_3 D'(`\sigma, \sigma')) \supset (D \sqsubseteq_D D')$$

However, the implication does not hold in the other direction. Consider the following descriptions:

$$D_1 = \{((s_1, s_1), \bot_3), ((s_1, s_2), \mathsf{T}_3), ((s_2, s_1), \mathsf{T}_3), ((s_2, s_2), \mathsf{F}_3)\}$$
$$D_2 = \{((s_1, s_1), \mathsf{F}_3), ((s_1, s_2), \bot_3), ((s_2, s_1), \mathsf{T}_3), ((s_2, s_2), \mathsf{F}_3)\}$$

The meanings of these descriptions as specifications are as follows:

$$[\![D_1]\!] = \{(s_1, \bot), (s_1, s_1), (s_1, s_2), (s_2, s_1)\}$$
$$[\![D_2]\!] = \{(s_1, \bot), (s_1, s_2), (s_2, s_1)\}$$

It can be seen that D_2 is a refinement of D_1, and the relationship $D_1 \sqsubseteq_D D_2$ holds as expected. Unfortunately D_1 is not refined by D_2 at all points under the relationship $\sqsubseteq_3$. This suggests that if we were to use $\sqsubseteq_3$ as our refinement relation, we might be prevented from making some valid refinement of a description.

No Problem

Consider another description, D_1'.

$$D_1' = \{((s_1, s_1), \bot_3), ((s_1, s_2), \bot_3), ((s_2, s_1), \mathsf{T}_3), ((s_2, s_2), \mathsf{F}_3)\}$$

This description represents the same specification as that represented by D_1 above; that is $[\![D_1']\!] = [\![D_1]\!]$. Note, however, that D_1' *is* refined by D_2.

It is easy to identify a class of descriptions whose members are refined, according to $\sqsubseteq_3$, by any genuine refinement. We call these the *clear* descriptions.

$$\vdash^{\text{def}} clear\, D = \forall \sigma_0.\, (\exists \sigma_1.\, D(\sigma_0, \sigma_1) = \bot_3) \supset (\forall \sigma_1.\, D(\sigma_0, \sigma_1) \neq \mathsf{T}_3) \qquad [clear\_\mathsf{DEF}]$$

$$\vdash clear\, D = (\forall D'.\, (\forall (\grave{}\sigma, \sigma').\, D(\grave{}\sigma, \sigma') \sqsubseteq_3 D'(\grave{}\sigma, \sigma')) = (D \sqsubseteq_D D')) \quad [clear\_\mathsf{THM}]$$

If we partition the set of descriptions into groups whose members denote the same specification, then each group will contain precisely one clear description.

Fortunately, when interpreted as descriptions, the three-valued predicates which most naturally capture our intended specification correspond to clear descriptions. In fact, any predicate that does not explicitly relate particular final states to the value $\bot_3$ denotes a clear description. For example, if we wish to specify a behaviour which is always required to terminate, then that specification can be captured by a description of the form:

$$\lfloor Rel \rfloor$$

If for some initial states nontermination is allowed, then the specification can be captured by a description of the form (where Pre does not contain variables from σ'):

$$Pre \supset\!\!\!) \lfloor Rel \rfloor$$

Note also that this precondition and postrelation form of specification can be found in different guises in Hoare logic, VDM and the various refinement calculi. Such specifications are generally regarded as providing a natural and sufficiently descriptive method describing behaviour.

The next section gives a semantics to the simple imperative programming language defined in the last chapter. The meaning of each command in the programming language is a clear description. The constructs of the programming language which combine descriptions to form new descriptions produce clear descriptions if their arguments are clear descriptions. A refinement process that begins with a clear description and progresses to an implementation via steps that combine program constructs and clear descriptions, will never encounter the potential problem highlighted earlier.

4.5 Programs as Three-Valued Predicates

This section uses three-valued predicates to give a semantics to the constructs of the programming langauge described in section 3.2. We will then be able treat three-valued predicates as a wide-spectrum language in which we can both phrase specifications and develop implementations.

skip
: The meaning of the skip statement is easily captured with the predicate that states that it terminates with the final state equal to the intial state.

$$\mathsf{skip} \stackrel{\text{def}}{=} \lfloor \grave{}\sigma = \sigma' \rfloor$$

$x' := e$
: The meaning of an assignment statement can be expressed by stating that it terminates with the final state equal to the initial state with the appropriate substitutions.

$$(x' := e) \stackrel{\text{def}}{=} \lfloor \grave{}\sigma = \sigma' \rfloor [e/\grave{}x]$$

This semantics extends to parallel assignment by regarding x' and e as vectors of variables and expressions.

if b then P else Q
: The semantics of the conditional statement is easily captured using a conditional expression.

$$(\mathsf{if}\ b\ \mathsf{then}\ P\ \mathsf{else}\ Q) \stackrel{\text{def}}{=} (b \Rightarrow P \mid Q)$$

$P \mid Q$
: The nondeterministic choice between P and Q can be expressed with three-valued disjunction.

$$P \mid Q \stackrel{\text{def}}{=} P \mathbin{\underline{\vee}} Q$$

$P; Q$

Sequential composition can be expressed using three-valued predicates as follows.

$$(P; Q) \stackrel{\text{def}}{=} (\underline{\exists}\sigma.\, P[\sigma/\sigma'] \between Q[\sigma/\grave{}\sigma])$$

$\mathsf{var}\, x.\, D$

We introduce a local variable x simply by providing an existentially quantified scope for its initial and final values.

$$(\mathsf{var}\, x.\, P) \stackrel{\text{def}}{=} (\underline{\exists}\grave{} x\ x'.\, P)$$

$\mathsf{label}\, l.\, D$

It is not always possible to give a meaning for a recursive block. However, if the body is a monotonic function of the label (as is the case if the block is executable), then we can express the meaning as the least fixed-point of that function.

$$\mathrm{mono}(\lambda l.\, D) \stackrel{\text{def}}{\vdash} (\mathsf{label}\, l.\, D) = \mathrm{fix}(\lambda l.\, D)$$

Note that both 'mono' and 'fix' are ordered with respect to $\sqsubseteq_3$.

4.6 Window on Three-Valued Booleans

Window inference can be used to manipulate terms of type $BOOL_3$ just as it can be used with any other type. However, to get the most out of the system, special rules must be introduced for opening subwindows on the operands of the three-valued connectives. In particular, we will need to introduce new rules if we wish to use window inference to preserve $\sqsupseteq_3$ (the refinement relation between three-valued predicates).

For example, the following window rules allow a three-valued disjunction to be refined by refining either of its arguments. The refinement may proceed under the assumption that the other argument is not $\bot_3$.

$$\frac{B \neq \bot_3 \vdash A' \sqsupseteq_3 A}{\vdash (A' \underline{\vee} B) \sqsupseteq_3 (A \underline{\vee} B)} \qquad \frac{A \neq \bot_3 \vdash B' \sqsupseteq_3 B}{\vdash (A \underline{\vee} B') \sqsupseteq_3 (A \underline{\vee} B)}$$

Similar rules can be given for opening a subwindow on either side of any of the strict versions of the familiar logical connectives. Further window rules are required for opening subwindows that preserve refinement on the $\supset$ and $\between$ operators.

$$\frac{B \neq \bot_3 \vdash A' \subset A}{\vdash (A' \supset B) \sqsupseteq_3 (A \supset B)} \qquad \frac{A \vdash B' \sqsupseteq_3 B}{\vdash (A \supset B') \sqsupseteq_3 (A \supset B)}$$

$$\frac{B \neq \mathsf{F}_3 \vdash A' \sqsupseteq_3 A}{\vdash (A' \between B) \sqsupseteq_3 (A \between B)} \qquad \frac{A \neq \mathsf{F}_3 \vdash B' \sqsupseteq_3 B}{\vdash (A \between B') \sqsupseteq_3 (A \between B)}$$

These rules illustrate the additional difficulty of reasoning about three-valued booleans. While in two-valued logic we may have been able to assume a particular term was true; in three-valued logic is is likely that we will only be able to assume that it is defined, or at best not false. Several example of using window inference to reason about three-valued booleans can be found in the next chapter.

4.7 Nonstandard Predicates

The notation proposed for use with program refinement in both this chapter and the previous one is not quite that of standard logic. Fortunately, the difference is small and our notation can be regarded as an abstract representation of the standard 'concrete' notation. Consider the predicate below:

$$\lfloor x' = \grave{}x + 1 \rfloor ; \lfloor x' = \grave{}x + 1 \rfloor$$

The intended meaning of this partial program is that x should be incremented, and then x should be incremented again. Assuming σ_0 and σ_1 are the initial and final states, our original predicate is actually an abstraction of the following concrete predicate (x_0 is a variable from the initial state, x is a variable from the intermediate state, and x_1 is a variable from the final state):

$$\exists \sigma . \lfloor x = x_0 + 1 \rfloor \between \lfloor x_1 = x + 1 \rfloor$$

Although in keeping with standard programming practice, the program-like notation is unusual because the term contains two instances of x' (and $\grave{}x$) that have the same name, but denote different variables. Similarly, the first x' and the second $\grave{}x$ denote the same variable (namely x), despite having different names. This means that a sequential composition of predicates is not a standard logical term when expressed in a program-like notation. However, the meaning of a sequential composition, as defined earlier in this section, is a standard logical term. We wish to express our programs in a familiar notation, and reason about them using standard logic. We will therefore regard a sequential composition of predicates as an abstract representation of standard logical notation that describes its meaning.

We can now make a slight generalisation to our notation. In addition to writing $\grave{}x$ to denote the value of x before the current statement, we also write $\ddot{}x$ to denote the value of x before that, and $\dddot{}x$ for the value of x before that, and so on. Similarly, while x' denotes the value of x after the current statement, x'' denotes the value of x after that, and so on. Consider the following predicate which uses multiple priming:

$$\lfloor x' = \grave{}x + 1 \rfloor ; \lfloor x' = \ddot{}x + 1 \rfloor$$

The meaning of this specification is that x should be incremented once. The form of the specification suggests that x should be incremented twice, but that the result of the first increment is ignored when calculating the second. Again, assuming σ_0 and σ_1 are the initial and final states, this predicate is an abstract representation of the following standard notation:

$$\underline{\exists}\sigma.\, \lfloor x = x_0 + 1 \rfloor \mathbin{/\!\!\wedge} \lfloor x_1 = x_0 + 1 \rfloor$$

Note that while expressions involving multiple primes, like $\grave{}x$ and x'', can be valuable in the intermediate stages of a refinement, they can never be executable.

The translation between the abstract and concrete representations of a predicate is dependant upon its context. The translation begins in an initial context and builds upon the context as it descends the structure of the term. For example, consider the following predicate:

$$\lfloor x_1 = x_0 + 1 \rfloor$$

If σ_0 and σ_1 are our initial and final states, then concrete predicate would correspond to the following abstract representation:

$$\lfloor x' = \grave{}x + 1 \rfloor$$

However, if this predicate were a subterm in a larger predicate, it might appear differently. In our previous example where this term was the second part of a sequential composition, it had the following abstract representation:

$$\lfloor x' = \grave{}\grave{}x + 1 \rfloor$$

The process of translation recursively descends through the other language constructs in the obvious way. For example, consider the following program fragment:

$$\mathsf{if}\ \grave{}x > 0\ \mathsf{then}\ \lfloor z' = \grave{}x \times \grave{}y \rfloor\ \mathsf{else}\ \lfloor (x', y') = (-\grave{}x, -\grave{}y) \rfloor; \lfloor z' = \grave{}x \times \grave{}y \rfloor$$

If the initial and final states are again σ_0 and σ_1, then this corresponds to the following concrete representation.

$$\mathsf{if}\ x_0 > 0\ \mathsf{then}\ \lfloor z_1 = x_0 \times y_0 \rfloor\ \mathsf{else}\ (\exists\sigma.\, \lfloor (x, y) = (-x_0, -y_0) \rfloor \mathbin{/\!\!\wedge} \lfloor z_1 = x \times y \rfloor)$$

4.8 Conclusions

It is sometimes assumed that nonterminating behaviour is never desirable and that therefore a specification language need not be able to describe it. However, if we wish to be able to mix specifications and program constructs in a unified framework, then the specification language must be able to describe all behaviours permitted by the programming language. Since it is easy to write programs that fail to terminate, the specification language must be capable of describing nonterminating behaviour.

This chapter described a language of three-valued predicates which can be used to describe potentially nonterminating behaviour more elegantly than the approaches based on two-valued predicates surveyed in the previous chapter. In particular, by introducing a third truth-value to explicitly represent nontermination, this model has been able to avoid the complexity of the relational model of predicates as specifications, while also avoiding the errors present in the total model.

The language of three-valued predicates described in this chapter retains the clarity and simplicity of specification that motivates the treatment of programs as predicates. Indeed, if a specification does not call for potential nontermination, it can be expressed as an ordinary two-valued predicate.

This chapter also defined a simple refinement relation between three-valued predicates and gave a semantics to a small imperative programming language that was as simple as any given using two-valued predicates in the previous chapter. These features are important as the complexity of the refinement relation and the semantics of the target language will be directly reflected in the complexity of reasoning about program refinements.

Chapter 5

A Method of Refinement

5.1 Introduction

This chapter combines the unified treatment of specifications and programs as three-valued predicates with the window-inference approach to transformational reasoning. The result is a method of program refinement. A similar method was described in a paper presented at *The Fifth Refinement Workshop* [Gru92]. In that paper, window inference was combined with a unified treatment of specifications and programs as predicates using the partial model described in section 3.4. The result guaranteed only the partial correctness of a derived implementation; the question of how to extend the technique to guarantee total correctness was left open. This question was addressed later in a paper presented at *The International Conference on Formal Methods in Programming and Their Applications* [Gru93b]. That paper introduced the three-valued logic described in the previous chapter, and gave some results demonstrating its applicability to program refinement.

Any program refinement will involve a number of simple transformations. Some sequences of transformations are frequently reused. Among those sequences reused most often are the one that introduce programming language constructs into a refinement. This chapter examines some of those sequences so that thereafter they may be used as single refinement steps. Where appropriate, they are presented alongside the window rules required to justify opening subwindows on the clauses of the constructs they introduce. All of the sequences examined could, to a greater or lesser extent, be automated by a programmable window-inference system like the one developed for the HOL theorem prover.

Section 5.9 presents a method of refining specifications into recursive implementations. Some authors [Dij76, Nel89] have noted that general recursion requires a more elaborate semantics than is required for simple iteration. As a result, much work on program refinement has concentrated on methods of developing iterative programs, or has only treated the development of recursive programs informally. Much of the difficulty in deriving recursive programs lies in the fact that very little can be assumed

about a program during the intermediate stages of its development. For example, it may not be possible to assume that a partly-developed recursive block is a monotonic function. These difficulties can be sidestepped by using an 'approximate' meaning for recursive blocks. The approximation chosen is accurate enough to support a natural style of refinement, and once the refinement process is complete the approximation can be replaced by the true meaning.

5.2 Program Refinement

The process of program refinement is begun by making a specification the focus of a window that can be transformed while preserving refinement. The idea is to transform the specification into an executable form. Some specifications are simple enough to transform directly into a program construct. Others must be decomposed into several simpler specifications combined by program constructs. Subwindows can then be opened on each of the simpler specifications to further pursue their refinement.

As an example, consider a few steps from the refinement of the specification $\lfloor z' = \grave{}x \times \grave{}y \rfloor$ which states that the initial values of x and y should be multiplied together and the result left in z. We begin by creating a window with the specification as its focus, and refinement as the relation it will preserve. (Note that the refinement relation is usually written $\sqsubseteq_3$; however, window inference requires a version oriented in the other direction.)[1]

$$\sqsupseteq_3 \star \lfloor z' = \grave{}x \times \grave{}y \rfloor$$

A conditional statement is introduced to allow separate consideration of the case where $\grave{}x$ is zero.

$$\sqsupseteq_3 \star \mathsf{if}\ \grave{}x = 0\ \mathsf{then}\ \boxed{\lfloor z' = \grave{}x \times \grave{}y \rfloor}\ \mathsf{else}\ \lfloor z' = \grave{}x \times \grave{}y \rfloor$$

A subwindow is then opened on the first clause of the conditional. In the context of the resulting window, we may assume that $\grave{}x$ is zero.

$$\begin{array}{l} \quad ! \ \grave{}x = 0 \\ \sqsupseteq_3 \star \lfloor z' = \grave{}x \times \grave{}y \rfloor \end{array}$$

Using the assumption, the focus may be simplified as follows:

$$\sqsupseteq_3 \star \lfloor z' = 0 \rfloor$$

This focus can be implemented with an assignment statement. The validity of this last step follows from the definition of assignment, and is examined further in section 5.4.

$$\sqsupseteq_3 \star z' := 0$$

[1] This new relation has the obvious definition $\vdash^{\mathsf{def}} \forall P\ Q.\ (P \sqsupseteq_3 Q) = (Q \sqsubseteq_3 P)$

From these few steps, it is possible to see how the combination of window inference and a unified treatment of specifications and programs yields a method of program refinement. A more complete example is presented at the end of the chapter.

The next few sections contain further examples of simple refinements. These examples concern transformations that introduce programming language constructs into a refinement.

5.3 skip Statements

Whenever the focus of a window is a specification that is satisfied by terminating in the same state as the initial state, the focus can be refined into a skip statement. This section illustrates a general procedure for refining such specifications into skip statements. Hereafter, whenever we wish to refine a specification into a skip statement, we will do so in a simple step, rather than repeat the reasoning outlined below.

Suppose P is a specification that would be satisfied by terminating with the same final state as the initial state.

$$\sqsupseteq_3 \star P$$

The specification may strengthened to require behaviour in accordance with skip.

$$\sqsupseteq_3 \star \mathsf{skip} \mathbin{/\!\!\backslash} \boxed{P}$$

We can then open a subwindow on P and simplify it under the assumption that the system behaves according to skip.

$$\begin{array}{l} \quad ! \ \boxed{\mathsf{skip} \neq \mathsf{F}_3} \\ \sqsupseteq_3 \star P \end{array}$$

Opening a subwindow on the assumption and expanding with the definition of skip (given in section 4.5) allows us to deduce that the initial and final states are the same.

$$\begin{array}{l} \subset \star \mathsf{skip} \neq \mathsf{F}_3 \\ \subset \star \lfloor \sigma' = \grave{}\sigma \rfloor \neq \mathsf{F}_3 \\ \subset \star \sigma' = \grave{}\sigma \end{array}$$

We close the subwindow, and return to its parent. The parent window contains a new lemma which states that the initial and final states are the same. Assuming that P does describe a specification that is satisfied by skip, it should now be possible to transform P into T_3.

$$\begin{array}{l} \quad ! \ \mathsf{skip} \neq \mathsf{F}_3 \\ \quad ¡ \ \sigma' = \grave{}\sigma \\ \sqsupseteq_3 \star P \\ \quad \vdots \\ \sqsupseteq_3 \star \mathsf{T}_3 \end{array}$$

Once the transformation of P to T_3 is complete, we close this window and simplify the focus of the parent.

$$\sqsupseteq_3 \star \mathsf{skip} \between \mathsf{T}_3$$
$$\sqsupseteq_3 \star \mathsf{skip}$$

With the exception of the simplification of P into T_3, this process is simple enough to be automated in a programmable window inference system like the one constructed for the HOL theorem prover. Even the simplification could be automated for certain predicates.

5.4 Assignment Statements

Whenever the focus of a window describes a specification that would be satisfied by assigning some value to a variable, then it can be refined into an assignment statement. The procedure for introducing an assignment is essentially the same as that for introducing skip.

Assuming P represents a specification that can be satisfied by an assignment, the refinement proceeds as follows. First, the assignment statement is introduced.

$$\sqsupseteq_3 \star P$$
$$\sqsupseteq_3 \star (x' := e) \between \boxed{P}$$

Once the assignment has been introduced, we open a subwindow on P and simplify it under the assumption that the assignment is made.

$$\begin{array}{l} \quad ! \ \boxed{(x' := e) \neq \mathsf{F}_3} \\ \sqsupseteq_3 \star P \end{array}$$

Opening a subwindow on the new assumption and using the definition of assignment from section 4.5, allows us to deduce that x' has the value e and all the other program variables are unchanged.

$$\begin{array}{l} \subset \star (v' := e) \neq \mathsf{F}_3 \\ \subset \star \lfloor \sigma' = \grave{}\sigma \rfloor [e/\grave{}x] \neq \mathsf{F}_3 \\ \subset \star (\sigma' = \grave{}\sigma)[e/\grave{}x] \end{array}$$

We close the subwindow and return to simplifying P. We may now proceed using the new lemma which states that $(\sigma' = \grave{}\sigma)[e/\grave{}x]$. If P does describe behaviour that would be satisfied by assigning e to x, then it should now be possible to transform P into T_3.

$$\begin{array}{l} \quad ! \ (x' := e) \neq \mathsf{F}_3 \\ \quad ¡ \ (\sigma' = \grave{}\sigma)[e/\grave{}x] \\ \sqsupseteq_3 \star P \\ \quad \vdots \\ \sqsupseteq_3 \star \mathsf{T}_3 \end{array}$$

Once the transformation of P into T_3 is complete, we close this window and simplify the focus of the parent.

$$\sqsupseteq_3 \star (x' := e) \mathbin{/\!\!\backslash} \mathsf{T}_3$$
$$\sqsupseteq_3 \star (x' := e)$$

The same procedure may be used to introduce parallel assignments, in which case x and e denote vectors of variables and expressions.

Like the introduction of skip, the process of introducing an assignment could be mostly automated in a programmable window inference system. Henceforth, whenever the focus can obviously be refined to true under the assumption that a particular assignment is made, we may refine the focus to that assignment in a single step.

5.5 Conditional Statements

Sometimes, it is helpful to split a refinement into two cases, depending on whether some condition b is true or not. In such situations, a conditional statement should be introduced. Any specification, P, can be refined into a conditional statement as follows:

$$\sqsupseteq_3 \star P$$
$$\sqsupseteq_3 \star \mathsf{if}\ b\ \mathsf{then}\ P\ \mathsf{else}\ P$$

The validity of this refinement follows directly from the definition of conditional statements given in section 4.5.

Once introduced, subwindows can be opened on the then and else clauses of a conditional to consider those two cases separately. The following window rule allows a conditional to be refined by opening a subwindow on the then clause. While refining the then clause, the condition may be assumed true.

$$\frac{\Gamma, b \vdash P' \sqsupseteq_3 P}{\Gamma \vdash (\mathsf{if}\ b\ \mathsf{then}\ P'\ \mathsf{else}\ Q) \sqsupseteq_3 (\mathsf{if}\ b\ \mathsf{then}\ P\ \mathsf{else}\ Q)}$$

Similarly, the rule below allows a conditional statement to be refined by opening a subwindow on the else clause. While refining the else clause, the condition may be assumed false.

$$\frac{\Gamma, \neg b \vdash Q' \sqsupseteq_3 Q}{\Gamma \vdash (\mathsf{if}\ b\ \mathsf{then}\ P\ \mathsf{else}\ Q') \sqsupseteq_3 (\mathsf{if}\ b\ \mathsf{then}\ P\ \mathsf{else}\ Q)}$$

The validity of these rules follows from the definition of the conditional statement and the monotonicity of conditional expressions.

5.6 Nondeterministic Choices

Specifications phrased in terms of a three-valued disjunction may be refined into a nondeterministic choice, as follows:

$$\sqsupseteq_3 \star\ P \mathbin{\underline{\vee}} Q$$
$$\sqsupseteq_3 \star\ P \mid Q$$

The refinement is valid because in section 4.5 choice was defined as three-valued disjunction.

Once introduced, subwindows can be opened on either subclause of a nondeterministic choice using the window rules below. Each subclause may be refined under the assumption that the other does not lead to nontermination.

$$\frac{\Gamma, Q \neq \bot_3 \vdash P' \sqsupseteq_3 P}{\Gamma \vdash (P' \mid Q) \sqsupseteq_3 (P \mid Q)} \qquad \frac{\Gamma, P \neq \bot_3 \vdash Q' \sqsupseteq_3 Q}{\Gamma \vdash (P \mid Q') \sqsupseteq_3 (P \mid Q)}$$

The validity of these window rules follows directly from the definition of choice and the monotonicity of three-valued disjunction.

5.7 Sequential Compositions

Any specification phrased as a conjunction of three-valued predicates may be trivially decomposed into a sequential composition. For example, consider the requirement $P \mathbin{\underline{\wedge}} Q$, which may be decomposed as follows:

$$\sqsupseteq_3 \star\ P \mathbin{\underline{\wedge}} Q$$
$$\sqsupseteq_3 \star\ P[\sigma'', \sigma''', \ldots / \sigma', \sigma'', \ldots]; Q[\grave{}\sigma, \grave{}\grave{}\sigma, \ldots / \grave{}\sigma, \grave{}\sigma, \ldots]$$

The resulting composition has the following concrete syntax: $\underline{\exists}\sigma.\, P \mathbin{/\!\!\wedge\!\!\backslash} Q$, where σ is a fresh variable. The relationship between the concrete and abstract forms of a composition was described in section 4.7. The refinement is valid because $/\!\!\wedge\!\!\backslash$ is a stronger connective than $\underline{\wedge}$. Note that although a composition has been introduced, the operations described by P and Q have not been placed in series. The variables in P that referred to states after the conjunction now refer to states after the composition. Similarly, those variables in Q that referred to states before the conjunction now refer to states before the compositions.

Subwindows can be opened to refine either clause of a sequential composition. The window rule below allows a sequential composition to be refined by opening a subwindow on the second clause. While refining the second clause, one can assume that the first clause has been executed.

$$\frac{\Gamma, P[\sigma'/\sigma] \neq \mathsf{F}_3 \vdash Q' \sqsupseteq Q[\sigma'/\sigma]}{\Gamma \vdash (\exists\sigma'.\, P[\sigma'/\sigma] \mathbin{/\!\!\wedge\!\!\backslash} Q') \sqsupseteq_3 (\exists\sigma.\, P \mathbin{/\!\!\wedge\!\!\backslash} Q)}$$

Similarly, the following window rule allows a sequential composition to be refined by opening a subwindow on the first clause. While refining this clause of a composition, one can assume that the second clause will be executed.

$$\frac{\Gamma,\, Q[\sigma'/\sigma] \neq \mathsf{F}_3 \vdash P' \sqsupseteq P[\sigma'/\sigma]}{\Gamma \vdash (\exists \sigma'.\, P' \mathbin{/\!\!\backslash} Q[\sigma'/\sigma]) \sqsupseteq_3 (\exists \sigma.\, P \mathbin{/\!\!\backslash} Q)}$$

The state bound by a sequential composition may need to be renamed when opening a subwindow inside the composition. This renaming does not alter its abstract representation.

The translation between the concrete and abstract representations of a term depends on its context. The context of a subwindow opened on either clause of a sequential composition is different from that of the parent window. This means that some terms in the subwindow may have a different appearance from that which they had in the parent window. For example, consider opening a subwindow on the expression P in the following window.

$$\begin{array}{l} \quad ! \; \Gamma \\ \sqsupseteq_3 \star \boxed{P}; Q \end{array}$$

In this window, a variable drawn from the final state σ' appearing in one of the assumptions refers to the value that variable will hold in the state after behaviour in accordance with $P; Q$. However, in the context of the subexpression P, the same variable appears to be drawn from the state σ''. Therefore, when opening a subwindow on P, some variables in the context of the window undergo an apparent renaming. The variables in the subexpression Q also undergo an apparent renaming. Consider those variables in Q that appear to be drawn from the initial state $`\sigma$. In the context of P, the same variables appear to be drawn from the final state σ'. The subwindow below illustrates these apparent renamings.

$$\begin{array}{l} \quad ! \; \Gamma[\sigma'', \sigma''', \ldots / \sigma', \sigma'', \ldots] \\ \quad ! \; Q[\ldots, `\sigma, \sigma', \sigma'', \sigma''', \ldots / \ldots, ``\sigma, `\sigma, \sigma', \sigma'', \ldots] \neq \mathsf{F}_3 \\ \sqsupseteq_3 \star P \end{array}$$

Similarly, the following windows illustrate the apparent renamings that take place when a subwindow is opened on the second clause of a composition.

$$\begin{array}{l} \quad ! \; \Gamma \\ \sqsupseteq_3 \star P; \boxed{Q} \end{array}$$

$$\begin{array}{l} \quad ! \; \Gamma[\ldots, ```\sigma, ``\sigma / \ldots, ``\sigma, `\sigma] \\ \quad ! \; P[\ldots, ```\sigma, ``\sigma, `\sigma, \sigma', \ldots / \ldots, ``\sigma, `\sigma, \sigma', \sigma'', \ldots] \neq \mathsf{F}_3 \\ \sqsupseteq_3 \star Q \end{array}$$

Finally, many requirements are most easily expressed in terms of a two-valued predicate—that is, with an expression of the form $\lfloor P \rfloor$. Therefore when opening subwindows on sequential compositions, it is common to generate assumptions of the form $\lfloor P \rfloor \neq \mathsf{F}_3$. Such an assumption is equivalent to the more simple assumption P. From now on, we will assume that when opening a subwindow on a composition, any new assumption of the form $\lfloor P \rfloor \neq \mathsf{F}_3$ will automatically be transformed into an assumption of the form P. Such automatic transformations can be easily added to the window inference interface to the HOL system. Indeed, the interface already contains several similar automatic simplifications.

Serialisation

Perhaps the most frequently performed refinement involving a sequential composition is to serialise a parallel operation. Therefore, a procedure is now described for decomposing a conjunction of requirements into a sequential composition of requirements that may be performed in series.

Consider the following specification which requires behaviour that satisfies both the predicates P and Q. The predicate P relates a collection of variables in the initial state $\grave{}s_{p_1}$ to some variables in the final state s'_{p_2}, while Q relates some of the variables in the initial state $\grave{}s_{q_1}$ to some in the final state s'_{q_2}.

$$\sqsupseteq_3 \star P[\grave{}s_{p_1}, s'_{p_2}] \mathbin{\underline{\triangle}} Q[\grave{}s_{q_1}, s'_{q_2}]$$

The refinement is partitioned into two cases, depending upon whether one of P or Q is equal to $\bot_3$ or not.

$$\begin{array}{rl} \sqsupseteq_3 \star (P[\grave{}s_{p_1}, s'_{p_2}] \neq \bot_3) \wedge (Q[\grave{}s_{q_1}, s'_{q_2}] \neq \bot_3) & \\ \quad \Rightarrow & \boxed{P[\grave{}s_{p_1}, s'_{p_2}] \mathbin{\underline{\triangle}} Q[\grave{}s_{q_1}, s'_{q_2}]} \\ \quad \| & P[\grave{}s_{p_1}, s'_{p_2}] \mathbin{\underline{\triangle}} Q[\grave{}s_{q_1}, s'_{q_2}] \end{array}$$

We start with the case where neither P nor Q is $\bot_3$.

$$\begin{array}{r} ! \; P[\grave{}s_{p_1}, s'_{p_2}] \neq \bot_3 \\ ! \; Q[\grave{}s_{q_1}, s'_{q_2}] \neq \bot_3 \\ \sqsupseteq_3 \star P[\grave{}s_{p_1}, s'_{p_2}] \mathbin{\underline{\triangle}} Q[\grave{}s_{q_1}, s'_{q_2}] \end{array}$$

A sequential composition can be introduced in the manner described at the beginning of this section.

$$\sqsupseteq_3 \star P[\grave{}s_{p_1}, s''_{p_2}]; Q[\grave{}s_{q_1}, s'_{q_2}]$$

Some of the variables in the resulting focus have multiple primes, and since such expressions are never considered executable, the next step is to eliminate the multiple priming. This is achieved by strengthening the first clause of the composition so that

it maintains the values of those variables needed as input to the second clause, and strengthening the second clause of the composition to maintain the values of those variables determined by the first clause.

$$\exists_3 \star (P[\grave{}s_{p_1}, s''_{p_2}] \underline{\triangle} \lfloor s'_{q_1} = \grave{}s_{q_1} \rfloor); \boxed{(\lfloor s'_{p_2} = \grave{}s_{p_2} \rfloor \underline{\triangle} Q[\breve{}s_{q_1}, s'_{q_2}])}$$

A window is then opened on the second clause of the composition.

$$\begin{array}{l} \quad ! \; P[\breve{}s_{p_1}, s'_{p_2}] \neq \bot_3 \\ \quad ! \; Q[\breve{}s_{q_1}, s'_{q_2}] \neq \bot_3 \\ \quad ! \; \boxed{(P[\breve{}s_{p_1}, s'_{p_2}] \underline{\triangle} \lfloor \grave{}s_{q_1} = \breve{}s_{q_1} \rfloor) \neq \mathsf{F}_3} \\ \exists_3 \star (\lfloor s'_{p_2} = \grave{}s_{p_2} \rfloor \underline{\triangle} Q[\breve{}s_{q_1}, s'_{q_2}]) \end{array}$$

By opening a subwindow on the assumption that the first clause of the composition has been executed, we can deduce that $\grave{}s_{q_1} = \breve{}s_{q_1}$.

$$\begin{array}{l} \quad \vdots \\ \quad ! \; P[\breve{}s_{p_1}, s'_{p_2}] \neq \bot_3 \\ \subset \star (P[\breve{}s_{p_1}, s'_{p_2}] \underline{\triangle} \lfloor \grave{}s_{q_1} = \breve{}s_{q_1} \rfloor) \neq \mathsf{F}_3 \\ \subset \star \grave{}s_{q_1} = \breve{}s_{q_1} \end{array}$$

Having derived this fact, we close the subwindow. This action creates a new lemma in the parent window.

$$\begin{array}{l} \quad \vdots \\ \quad ¡ \; \grave{}s_{q_1} = \breve{}s_{q_1} \\ \exists_3 \star (\lfloor s'_{p_2} = \grave{}s_{p_2} \rfloor \underline{\triangle} Q[\breve{}s_{q_1}, s'_{q_2}]) \end{array}$$

Using the new lemma, we can remove the multiple priming from s_{q_1}.

$$\exists_3 \star (\lfloor s'_{p_2} = \grave{}s_{p_2} \rfloor \underline{\triangle} Q[\grave{}s_{q_1}, s'_{q_2}])$$

This window can now be closed.

$$\begin{array}{l} \quad ! \; P[\grave{}s_{p_1}, s'_{p_2}] \neq \bot_3 \\ \quad ! \; Q[\grave{}s_{q_1}, s'_{q_2}] \neq \bot_3 \\ \exists_3 \star (P[\grave{}s_{p_1}, s''_{p_2}] \underline{\triangle} \lfloor s'_{q_1} = \grave{}s_{q_1} \rfloor); (\lfloor s'_{p_2} = \grave{}s_{p_2} \rfloor \underline{\triangle} Q[\grave{}s_{q_1}, s'_{q_2}]) \end{array}$$

By repeating the same line of reasoning just used on the second clause of the composition, the multiple priming of s''_{p_2} can also be removed.

$$\exists_3 \star (P[\grave{}s_{p_1}, s'_{p_2}] \underline{\triangle} \lfloor s'_{q_1} = \grave{}s_{q_1} \rfloor); (\lfloor s'_{p_2} = \grave{}s_{p_2} \rfloor \underline{\triangle} Q[\grave{}s_{q_1}, s'_{q_2}])$$

Next, we close this window and return to its parent.

$$\begin{array}{l} \exists_3 \star (P[\grave{}s_{p_1}, s'_{p_2}] \neq \bot_3) \wedge (Q[\grave{}s_{q_1}, s'_{q_2}] \neq \bot_3) \\ \qquad \Rightarrow \quad (P[\grave{}s_{p_1}, s'_{p_2}] \underline{\triangle} \lfloor s'_{q_1} = \grave{}s_{q_1} \rfloor); (\lfloor s'_{p_2} = \grave{}s_{p_2} \rfloor \underline{\triangle} Q[\grave{}s_{q_1}, s'_{q_2}]) \\ \qquad\quad \mathsf{I} \quad \boxed{P[\grave{}s_{p_1}, s'_{p_2}] \underline{\triangle} Q[\grave{}s_{q_1}, s'_{q_2}]} \end{array}$$

We now open a window to examine the case were at least one of P and Q is equal to $\bot_3$.

$$\begin{array}{l} \quad ! \ (P[\grave{}s_{p_1}, s'_{p_2}] \neq \bot_3) \vee (Q[\grave{}s_{q_1}, s'_{q_2}] \neq \bot_3) \\ \sqsupseteq_3 \star \ P[\grave{}s_{p_1}, s'_{p_2}] \mathbin{\underline{\triangle}} Q[\grave{}s_{q_1}, s'_{q_2}] \end{array}$$

Under this assumption, we know from the definition of $\underline{\triangle}$ that the focus is $\bot_3$.

$$\sqsupseteq_3 \star \bot_3$$

We can refine a focus of $\bot_3$ into anything, we choose to refine it into the the same composition arrived at for the case where neither P nor Q was equal to $\bot_3$. This will subsequently allow us to eliminate the conditional altogether.

$$\sqsupseteq_3 \star (P[\grave{}s_{p_1}, s'_{p_2}] \mathbin{\underline{\triangle}} \lfloor s'_{q_1} = \grave{}s_{q_1} \rfloor); (\lfloor s'_{p_2} = \grave{}s_{p_2} \rfloor \mathbin{\underline{\triangle}} Q[\grave{}s_{q_1}, s'_{q_2}])$$

We now close this window.

$$\begin{array}{rl} \sqsupseteq_3 \star (P[\grave{}s_{p_1}, s'_{p_2}] \neq \bot_3) \wedge (Q[\grave{}s_{q_1}, s'_{q_2}] \neq \bot_3) & \\ \quad \Rightarrow & (P[\grave{}s_{p_1}, s'_{p_2}] \mathbin{\underline{\triangle}} \lfloor s'_{q_1} = \grave{}s_{q_1} \rfloor); (\lfloor s'_{p_2} = \grave{}s_{p_2} \rfloor \mathbin{\underline{\triangle}} Q[\grave{}s_{q_1}, s'_{q_2}]) \\ \quad \mathbf{|} & (P[\grave{}s_{p_1}, s'_{p_2}] \mathbin{\underline{\triangle}} \lfloor s'_{q_1} = \grave{}s_{q_1} \rfloor); (\lfloor s'_{p_2} = \grave{}s_{p_2} \rfloor \mathbin{\underline{\triangle}} Q[\grave{}s_{q_1}, s'_{q_2}]) \end{array}$$

Since both branches of the conditional are the same, it can be simplified away.

$$\sqsupseteq_3 \star (P[\grave{}s_{p_1}, s'_{p_2}] \mathbin{\underline{\triangle}} \lfloor s'_{q_1} = \grave{}s_{q_1} \rfloor); (\lfloor s'_{p_2} = \grave{}s_{p_2} \rfloor \mathbin{\underline{\triangle}} Q[\grave{}s_{q_1}, s'_{q_2}])$$

The two operations P and Q have now been serialised.

Each step in the process of serialising a conjunction of requirements is simple enough that it could be automated in a programmable window inference system. From now on, whenever we encounter a specification of the form:

$$P[\grave{}s_{p_1}, s'_{p_2}] \mathbin{\underline{\triangle}} Q[\grave{}s_{q_1}, s'_{q_2}]$$

we may refine it into a sequential composition of the form below in a single step.

$$(P[\grave{}s_{p_1}, s'_{p_2}] \mathbin{\underline{\triangle}} \lfloor s'_{q_1} = \grave{}s_{q_1} \rfloor); (\lfloor s'_{p_2} = \grave{}s_{p_2} \rfloor \mathbin{\underline{\triangle}} Q[\grave{}s_{q_1}, s'_{q_2}])$$

Note that while this procedure may be applied to any conjunction, if the variables set by the first predicate are not disjoint from the variables required as input to the second, then the resulting composition may require that the value of a variable be both modified and kept the same. Such a contradictory requirement can only be satisfied by a miracle. It will therefore be impossible to further refine the resulting composition into an executable form.

Initialisation

Another frequently performed refinement involving sequential composition is the introduction of an initialisation. Suppose we have a predicate Q that we would like to refine. Let us assume that $\grave{}s$ denotes the set of variables from the initial state that occur in Q. Suppose also that the refinement would be easier if some initial conditions were established prior to Q; for example, we might wish to establish an invariant. Assuming those initial conditions are captured by the predicate I, the development could proceed as follows:

$$\sqsupseteq_3 \star Q[\grave{}s]$$

We introduce a conjunction of $\lfloor I \rfloor$ with Q.

$$\sqsupseteq_3 \star \lfloor I \rfloor \mathbin{\underline{\wedge}} Q[\grave{}s]$$

The focus is then serialised using the procedure described earlier in this section.

$$\sqsupseteq_3 \star (\lfloor I \rfloor \mathbin{\underline{\wedge}} \lfloor s' = \grave{}s \rfloor); Q[\grave{}s]$$

A slight simplification completes the development.

$$\sqsupseteq_3 \star \lfloor I \wedge s' = \grave{}s \rfloor; Q[\grave{}s]$$

5.8 Local Variable Blocks

Sometimes it is helpful to introduce a local variable when refining a specification. Any specification can be refined by enclosing it in a block that introduces a local variable, provided the variable is not already used.

$$\sqsupseteq_3 \star P$$

Provided no variant of x is free in P.

$$\sqsupseteq_3 \star \mathsf{var}\, x.\, P$$

The validity of this refinement step follows from the definition of var in section 4.5.

Once introduced, a subwindow can be opened to refine the body of a local variable block. Opening a subwindow on the body of such a block may require the bound variable be renamed to one that does not occur free in the context of the window. The following window rule is used when opening a subwindow on the body of a local variable block.

$$\frac{\Gamma \vdash P' \sqsupseteq_3 P[x'/x]}{\Gamma \vdash (\mathsf{var}\, x'.\, P') \sqsupseteq_3 (\mathsf{var}\, x.\, P)} [\text{no variant of } x' \text{ free in } \Gamma \text{ or } \mathsf{var}\, x.\, P]$$

Note that inside the scope of a local block, the state space σ should be extended to include the new variable.

Initialised Local Blocks

It is common when introducing a new variable to initialise it to hold a particular value. The following general procedure justifies the introduction of an initialisation along with a new variable. The procedure is simple enough to be automated.

Suppose we are refining a predicate P and would like to introduce a new variable x initialised with the value e. Let us assume that $\grave{}s$ is the set of variables from the initial state that occur in P. The refinement would proceed as follows:

$$\sqsupseteq_3 \star P[\grave{}s]$$

First we introduce the new variable, assuming x is not used in P.

$$\sqsupseteq_3 \star \mathsf{var}\, x \,.\, P[\grave{}s]$$

Then we can initialise P with the requirement that x equals e. This is done using the procedure described in the previous section.

$$\sqsupseteq_3 \star \mathsf{var}\, x \,.\, \lfloor x' = e \wedge s' = \grave{}s \rfloor ; P[\grave{}s]$$

The first clause of the composition can then be refined into an assignment using the procedure described in section 5.4.

$$\sqsupseteq_3 \star \mathsf{var}\, x \,.\, x' := e ; P[\grave{}s]$$

5.9 Labelled Blocks

The development of labelled blocks is more interesting than that of the other program constructs. The goal of this section is to support a method of developing recursive blocks with the following properties:

- A specification can be refined by enclosing it in a labelled block.
- A labelled block can be refined by refining its body.
- If the body of a block is refined into some function of the original specification that the block is being used to implement, then the block can be further refined by replacing the original specification with a recursive call to the label.

Refinement to Terminating Code

Unfortunately, the requirements of the refinement style just described are not sufficient to guarantee the correctness of the resulting implementations. Any description could be given a label; then, since the body of the block is the original specification, it could be further refined into a recursive call to the label. This is not, in general, a valid refinement. The problem is caused by attempting to refine a specification into a nonterminating implementation. Moreover, the task of demonstrating the correctness of a procedure for introducing and refining labelled blocks is complicated by the fact that very little can be assumed about the form of a block during the intermediate stages of its development. For example, the body of a partially developed block need not be a monotonic function of its label.

These problems can be avoided by contriving an approximate meaning for labelled blocks. The approximation should support the refinement of recursive blocks in the style desired, while forcing the development of terminating implementations. The approximation should be weaker than the true meaning, so that when a development is complete, the approximate meaning can be replaced by the true meaning.

To this end, each block will now be annotated with two subscripts. The first of these, v, must be a function from the state space to some well-ordered domain like the natural numbers. (For the examples in this dissertation we will assume, without loss of generality, that the domain *is* the natural numbers.) This function is called the *variant* of the block. The second subscript, O, should be the original description that the block was introduced to implement. The approximate meaning associated with such a block is as follows:

$$\begin{array}{l} \models^{\mathrm{def}} (\mathsf{label}_{(v,O)}\, l.\, D)(\sigma_0, \sigma_1) = \\ \quad (\lambda l.\, D) \\ \quad\quad (\lambda(\sigma_a, \sigma_b).\, (v\ \sigma_a < v\ \sigma_0) \supset O(\sigma_a, \sigma_b)) \\ \quad\quad (\sigma_0, \sigma_1) \end{array} \qquad [\mathsf{label}_{(v,O)}\text{\_DEF}]$$

That is, in states where the variant has decreased, a call to the label invokes the original specification O. In all other states, the meaning of the label is not defined.

From this definition, it can been seen that any specification in which the only program variables are drawn from the current initial and final states, and where the label l does not occur free, can be refined as follows:

$$\begin{array}{l} \sqsupseteq_3 \star\, O(`\sigma, \sigma') \\ \qquad \text{Provided } l \text{ is not free in } O(`\sigma, \sigma'). \\ \sqsupseteq_3 \star\, \mathsf{label}_{(v,O)}\, l.\, O(`\sigma, \sigma') \end{array}$$

This refinement holds for any variant v.

The contrived meaning for recursive blocks was chosen specifically so that the following window rule would be valid. The rule states that a recursive block may be refined by refining its body. It also states that while refining the body of a block,

if the value of the variant has decreased since the beginning of the block, then any instance of the original specification can be refined into a call to the label. This allows recursive blocks to be developed in the style required, while ensuring that resulting implementations are terminating.

$$\frac{\gamma, \forall \sigma_0\, \sigma_1.\, ((v\ \sigma_0 < v\ `\sigma) \supset (l'(\sigma_0, \sigma_1) \sqsupseteq_3 O(\sigma_0, \sigma_1))) \vdash D' \sqsupseteq_3 D[l'/l]}{\Gamma \vdash (\mathsf{label}_{(v,O)}\ l'.\ D') \sqsupseteq_3 (\mathsf{label}_{(v,O)}\ l.\ D)}$$

Note that γ denotes those assumptions in Γ that contain no program variables.

As with other window rules used to open a subwindow inside the scope of a bound variable, the bound variable (in this case, the label of the block) may need to be renamed.

At the end of a refinement process begun by giving a description a label, and advanced by repeatedly refining the body of the labelled block, we will have a block of the form $\mathsf{label}_{(v,O)}\ l.\ D$, where D is executable. The product of this process is a theorem of the following form:

$$\vdash (\mathsf{label}_{(v,O)}\ l.\ D)(`\sigma, \sigma') \sqsupseteq_3 O(`\sigma, \sigma')$$

The final step in the refinement is to drop the subscripts from the labelled block by refining it into its true meaning. This step is justified by the theorem below.

Theorem. *The true meaning for a labelled block is stronger than the contrived meaning whenever the block is executable (and therefore monotonic).*

$$\forall \sigma_x\, \sigma_y.\, (\mathsf{label}_{(v,O)}\ l.\ D)(\sigma_x, \sigma_y) \sqsupseteq_3 O(\sigma_x, \sigma_y), \mathrm{mono}(\lambda l.\ D) \vdash$$
$$(\mathsf{label}\ l.\ D)(\sigma_0, \sigma_1) \sqsupseteq_3 (\mathsf{label}_{(v,O)}\ l.\ D)(\sigma_0, \sigma_1)$$

Proof. The proof is a well-founded induction over the value of the variant in the initial state.

Inductive Hypothesis:

$$v\ \sigma_a < v\ \sigma_0, \forall \sigma_x\, \sigma_y.\, (\mathsf{label}_{(v,O)}\ l.\ D)(\sigma_x, \sigma_y) \sqsupseteq_3 O(\sigma_x, \sigma_y), \mathrm{mono}(\lambda l.\ D) \vdash$$
$$\mathrm{fix}(\lambda l.\ D)(\sigma_a, \sigma_b) \sqsupseteq_3 (\mathsf{label}_{(v,O)}\ l.\ D)(\sigma_a, \sigma_b)$$

For the induction, we hypothesise that we have proved that the true meaning is stronger than the contrived meaning for any initial state (σ_a) where the variant is less than it is in σ_0. We then show that our hypothesis also holds for the state σ_0.

$(\mathsf{label}_{(v,O)}\ l.\ D)(\sigma_0, \sigma_1)$

We replace the block with its definition:

$=$ $(\lambda l.\, D)$ [$\mathsf{label}_{(v,O)}$_DEF]
$(\lambda(\sigma_a, \sigma_b).\, (v\ \sigma_a < v\ \sigma_0) \supset O(\sigma_a, \sigma_b))$
(σ_0, σ_1)

We use the assumption that O is refined by the block, together with the fact that $(\lambda l.\, D)$ is monotonic, to refine O into the block:

$\sqsubseteq_3$ $(\lambda l.\, D)$ [D_MONO]
$(\lambda(\sigma_a, \sigma_b).\, (v\ \sigma_a < v\ \sigma_0) \supset (\mathsf{label}_{(v,O)}\ l.\, D)(\sigma_a, \sigma_b))$
(σ_0, σ_1)

In the context of the term '$(\mathsf{label}_{(v,O)}\ l.\, D)(\sigma_a, \sigma_b)$', we can assume that $v\ \sigma_a$ is less than $v\ \sigma_0$. Therefore we can use the inductive hypothesis and the monotonicity of $(\lambda l.\, D)$ to further refine the block into its true meaning:

$\sqsubseteq_3$ $(\lambda l.\, D)$ [INDUCT_HYP]
$(\lambda(\sigma_a, \sigma_b).\, (v\ \sigma_a < v\ \sigma_0) \supset \mathrm{fix}(\lambda l.\, D)(\sigma_a, \sigma_b))$
(σ_0, σ_1)

Since $(\lambda l.\, D)$ is monotonic, the conditional clause can be refined simply to $\mathrm{fix}(\lambda l.\, D)(\sigma_a, \sigma_b)$:

$\sqsubseteq_3$ $(\lambda l.\, D)(\lambda(\sigma_a, \sigma_b).\, \mathrm{fix}(\lambda l.\, D)(\sigma_a, \sigma_b))(\sigma_0, \sigma_1)$ [D_MONO]

The abstraction over (σ_a, σ_b) can be removed through η-conversion:

$=$ $(\lambda l.\, D)(\,\mathrm{fix}(\lambda l.\, D))(\sigma_0, \sigma_1)$ [η_CONV]

Using the definition of a fixed point and the assumption that $(\lambda l.\, D)$ is monotonic, we can remove the application of $(\lambda l.\, D)$ to its fixed point:

$=$ $\mathrm{fix}(\lambda l.\, D)(\sigma_0, \sigma_1)$ [fix_DEF]

We have now proved the following theorem:

$$\forall \sigma_x\ \sigma_y.\, (\mathsf{label}_{(v,O)}\ l.\, D)(\sigma_x, \sigma_y) \sqsupseteq_3 O(\sigma_x, \sigma_y), \mathrm{mono}(\lambda l.\, D) \vdash \mathrm{fix}(\lambda l.\, D)(\sigma_0, \sigma_1) \sqsupseteq_3 (\mathsf{label}_{(v,O)}\ l.\, D)(\sigma_0, \sigma_1)$$

We complete the proof using the definition of $(\mathsf{label}\ l.\, D)$.

$$\forall \sigma_x\ \sigma_y.\, (\mathsf{label}_{(v,O)}\ l.\, D)(\sigma_x, \sigma_y) \sqsupseteq_3 O(\sigma_x, \sigma_y), \mathrm{mono}(\lambda l.\, D) \vdash (\mathsf{label}\ l.\, D)(\sigma_0, \sigma_1) \sqsupseteq_3 (\mathsf{label}_{(v,O)}\ l.\, D)(\sigma_0, \sigma_1)$$

□

Analogous proofs can be performed for nested recursive blocks by using Bekič's theorem [Bek84] and multiple inductions. With a little extra effort, a similar proof can be produced for recursive blocks whose meanings have been defined as a least upper-bound rather than a least fixed-point.

Invariants

Invariants are an important aid for reasoning about recursive programs. We would therefore like to associate an invariant with each labelled block. The meaning of a block extended with an invariant should require that in order to introduce a block, we would be required to prove that the invariant is satisfied. We should also be required to show the invariant was true before being able to introduce a recursive call to the label. In return for these additional restrictions, we would be able to assume that the invariant is true in the context of the body of the block.

Here is another contrived meaning for blocks which supports the use of invariants. The subscripts v and O denote, as before, the variant and original description that the block was used to refine. The new subscript, I, is the invariant of the block and is a function from the program state to booleans.

$$\models^{\mathrm{def}} (\mathsf{label}_{(v,O,I)}\, l.\, D)(\sigma_0, \sigma_1) = \left(\begin{array}{l} I\ \sigma_0 \supset \\ \quad (\lambda l.\, D) \\ \qquad (\lambda(\sigma_a, \sigma_b).\, I\ \sigma_a \wedge (v\ \sigma_a < v\ \sigma_0) \supset O(\sigma_a, \sigma_b)) \\ \qquad (\sigma_0, \sigma_1) \end{array} \right) \qquad [\mathsf{label}_{(v,O,I)}\_\mathsf{DEF}]$$

It can be seen that if the invariant I holds, then a specification O can be refined by giving it a label l; providing that the program variables in O are drawn only from the initial and final states, and the label l does not occur free in O.

$$\sqsupseteq_3 \star\, O(\grave{}\sigma, \sigma')$$

Provided $I\ \grave{}\sigma$, and l is not free in $O(\grave{}\sigma, \sigma')$.

$$\sqsupseteq_3 \star\, \mathsf{label}_{(v,O,I)}\, l.\, O(\grave{}\sigma, \sigma')$$

The refinement holds for any variant v.

The approximate meaning for recursive blocks with invariants was chosen specifically so the following window rule would be valid. The rule states that a recursive block may be refined by refining its body. It also states that while refining the body, the invariant can be assumed to hold at the beginning of the block. Furthermore, in any state where the invariant holds and the value of the variant has decreased, an instance of the original specification can be refined into a call to the label of the block.

$$\frac{\gamma, I\ \grave{}\sigma, \forall \sigma_0\, \sigma_1.\, (I\ \sigma_0 \wedge (v\ \sigma_0 < v\ \grave{}\sigma) \supset (l'(\sigma_0, \sigma_1) \sqsupseteq_3 O(\sigma_0, \sigma_1))) \vdash D' \sqsupseteq_3 D[l'/l]}{\Gamma \vdash (\mathsf{label}_{(v,O,I)}\, l'.\, D') \sqsupseteq_3 (\mathsf{label}_{(v,O,I)}\, l.\, D)}$$

Note that γ denotes those assumptions in Γ that contain no program variables.

At the end of a refinement process begun by giving a description a label with an invariant, and advanced by repeatedly refining the body of the labelled block, we will

have a block of the form $(\mathsf{label}_{(v,O,I)}\ l.\ D)$, where D is executable. The result of this process is a theorem of the form below:

$$I\ \grave{}\sigma \vdash (\mathsf{label}_{(v,O,I)}\ l.\ D)(\grave{}\sigma, \sigma') \sqsupseteq_3 O(\grave{}\sigma, \sigma')$$

The next step is to drop the subscripts by refining the contrived meaning for the block into the true meaning.

The following theorem states that once a block has been refined into an executable form, the contrived meaning can be replaced by the true meaning.

Theorem. *The true meaning for a labelled recursive block is stronger than the contrived meaning whenever the block is executable (and therefore monotonic).*

$$\forall \sigma_x\ \sigma_y.\ I\ \sigma_x \supset ((\mathsf{label}_{(v,O,I)}\ l.\ D)(\sigma_x, \sigma_y) \sqsupseteq_3 O(\sigma_x, \sigma_y)), \mathrm{mono}(\lambda l.\ D) \vdash$$
$$(\mathsf{label}\ l.\ D)(\sigma_0, \sigma_1) \sqsupseteq_3 (\mathsf{label}_{(v,O,I)}\ l.\ D)(\sigma_0, \sigma_1)$$

The proof of this theorem is almost identical to that given for blocks without an invariant, and is therefore omitted.

5.10 Example

The refinement method described in this chapter will now be demonstrated using a simple example. The specification to be refined states that the value of the program variable x is to be multiplied by that of y, and the result left in z. Let us assume that multiplication is not an executable expression. Instead, the desired result will have to be achieved using repeated addition. This is the same example as was presented in [Gru92]; only this time, the refinement proves the total correctness of the implementation.

How to Read a Refinement

Each window in the development has a number. This number reflects the 'depth' of that window. Window numbers do not increase monotonically during a development. Rather, they go up and down as the development explores different areas of the problem. The first window in a development is Window 1. If the current window is Window n, and we open a subwindow, the resulting window is Window $n+1$. When we close that subwindow, we return to Window n.

The easiest way to read a development is to skip over some of the subwindows. For example, if you are currently reading Window n and the next window in the development is Window $n+1$, you should not necessarily read this window next. Instead, skip forward to the next instance of Window n. If you find that the step between one instance of Window n and the next is a simple and logical one, then just keep reading. If, on the other hand, you find that the step was not clear, return to where you were and continue reading at Window $n+1$.

This technique may not prove to be of much use on the first few windows of a development. For example, in skipping from one instance of Window 1 to the next, you may miss out almost the entire development. However, at the lower levels, it should prove to be more helpful.

The Refinement

The development begins by making the specification we wish to implement the focus of a window that preserves refinement. We can then proceed to transform the focus into an implementation. Note that the state space σ contains the variables x, y and z.

Window: 1

$$\sqsupseteq_3 \star \lfloor z' = \grave{}x \times \grave{}y \rfloor$$

The specification states that the system should terminate in a state where the final value of z is equal to the product of the initial values of x and y. The first step of the refinement is to extended the state space with a temporary variable t. The intention is to find a recursive implementation that repeatedly adds x to a running total held in z. The new variable t will be used to keep track of how many times x has been added to the total. The process stops when x has been added y times.

$$\sqsupseteq_3 \star \mathsf{var}\, t. \boxed{\lfloor z' = \grave{}x \times \grave{}y \rfloor}$$

We open a subwindow inside the scope of t.

Window: 2

$$\sqsupseteq_3 \star \lfloor z' = \grave{}x \times \grave{}y \rfloor$$

Before we can introduce a recursive block, we must ensure that its invariant holds. The desired invariant states that z holds the accumulation of t additions of x. It must also state that t has not exceeded y, the required number of additions. A predicate, I_m, is defined to capture these requirements.

$$\models^{\text{def}} \forall \sigma.\, I_m\, \sigma = (z = x \times t) \wedge (t \leq y) \qquad [I_m\text{\_DEF}]$$

We initialise the current focus with a requirement to establish the invariant. (The procedure for making this kind of refinement was described in section 5.7.)

$$\sqsupseteq_3 \star \lfloor I_m\, \sigma' \wedge ((x', y') = (\grave{}x, \grave{}y)) \rfloor; \boxed{\lfloor z' = \grave{}x \times \grave{}y \rfloor}$$

We open a subwindow to concentrate on the main requirement. Later, we will return to implement the initialisation.

Window: 3

$$\begin{array}{l}\vdots\\ !\ I_m\ \grave{}\sigma\\ \sqsupseteq_3 \star \lfloor z' = \grave{}x \times \grave{}y \rfloor\end{array}$$

In the context of this subwindow, we may assume that the invariant has been established. It is therefore possible to introduce the recursive block. The variant that we wish to decrease on each call to the block is the difference between the values of y and t. We define a function v_m that describes the variant as a function of the state:[2]

$$\models^{\text{def}} \forall \sigma.\ v_m\ \sigma = |y - t| \qquad [v_m\text{\_DEF}]$$

The 'original' specification that the block will be introduced to refine is the focus of the current window. The predicate O_m is defined to capture that specification.

$$\models^{\text{def}} \forall \sigma_1\ \sigma_2.\ O_m(\sigma_1, \sigma_2) = (z_2 = x_1 \times y_1) \qquad [O_m\text{\_DEF}]$$

The block may now be introduced as follows:

$$\sqsupseteq_3 \star \mathsf{label}_{(v_m, O_m, I_m)}\ M.\ \boxed{\lfloor z' = \grave{}x \times \grave{}y \rfloor}$$

We continue by opening a subwindow inside the body of the block.

Window: 4

$$\begin{array}{l}!\ I_m\ \grave{}\sigma\\ !\ \forall \sigma_1\ \sigma_2.\ I_m\ \sigma_1 \wedge (v_m\ \sigma_1 < v_m\ \grave{}\sigma) \supset (M(\sigma_1, \sigma_2) \sqsupseteq_3 O_m(\sigma_1, \sigma_2))\\ \sqsupseteq_3 \star \lfloor z' = \grave{}x \times \grave{}y \rfloor\end{array}$$

We may now assume that the invariant holds. We may also assume that in any state where the invariant holds and the value of the variant has decreased, then the label can be used to implement the original specification.

By expanding the assumption I_m with its definition, we derive the following lemmas:

$$\begin{array}{l}¡\ \grave{}z = \grave{}x \times \grave{}t\\ ¡\ \grave{}t \leq \grave{}y\\ \sqsupseteq_3 \star \lfloor z' = \grave{}x \times \grave{}y \rfloor\end{array}$$

We now introduce a conditional statement to partition the requirements into two cases, depending on whether $\grave{}t$ and $\grave{}y$ are equal or not.

$$\sqsupseteq_3 \star \mathsf{if}\ \grave{}t = \grave{}y\ \mathsf{then}\ \boxed{\lfloor z' = \grave{}x \times \grave{}y \rfloor}\ \mathsf{else}\ \lfloor z' = \grave{}x \times \grave{}y \rfloor$$

We open a subwindow to consider the implementation of the **then** clause of the conditional.

[2] The expression '$|x|$' denotes the absolute value of x.

Window: 5

$$
\begin{array}{l}
\vdots \\
\mathsf{i}\ \grave{}z = \grave{}x \times \grave{}t \\
!\ \grave{}t = \grave{}y \\
\sqsupseteq_3 \star \lfloor z' = \grave{}x \times \grave{}y \rfloor
\end{array}
$$

In the context of this subwindow, we may assume that $\grave{}t$ and $\grave{}y$ are equal. From this assumption and the lemma $\grave{}z = \grave{}x \times \grave{}t$, we can see that the requirements of the focus are already satisfied in the initial state. Therefore, we refine the focus into a skip statement using the procedure described in section 5.3.

$$\sqsupseteq_3 \star \mathsf{skip}$$

The implementation of this window is now complete. We therefore close it and return to its parent.

Window: 4

$$
\begin{array}{l}
\vdots \\
!\ \forall \sigma_1\, \sigma_2.\, I_m\, \sigma_1 \wedge (v_m\, \sigma_1 < v_m\, \grave{}\sigma) \supset (M(\sigma_1, \sigma_2) \sqsupseteq_3 O_m(\sigma_1, \sigma_2)) \\
\mathsf{i}\ \grave{}z = \grave{}x \times \grave{}t \\
\mathsf{i}\ \grave{}t \leq \grave{}y \\
\sqsupseteq_3 \star \mathsf{if}\ \grave{}t = \grave{}y\ \mathsf{then\ skip\ else}\ \boxed{\lfloor z' = \grave{}x \times \grave{}y \rfloor}
\end{array}
$$

Now that the derivation of the then clause is complete, we turn our attention of the development of the else clause.

Window: 5

$$
\begin{array}{l}
\vdots \\
!\ \forall \sigma_1\, \sigma_2.\, I_m\, \sigma_1 \wedge (v_m\, \sigma_1 < v_m\, \grave{}\sigma) \supset (M(\sigma_1, \sigma_2) \sqsupseteq_3 O_m(\sigma_1, \sigma_2)) \\
\mathsf{i}\ \grave{}z = \grave{}x \times \grave{}t \\
\mathsf{i}\ \grave{}t \leq \grave{}y \\
!\ \grave{}t \neq \grave{}y \\
\sqsupseteq_3 \star \lfloor z' = \grave{}x \times \grave{}y \rfloor
\end{array}
$$

In the context of this subwindow, we can assume that $\grave{}t$ and $\grave{}y$ are not equal, and consequently that the variant is not equal to zero. We therefore prefix the focus with a requirement to decrease the variant while maintaining the invariant. Again, this is performed using the initialisation transformation described in section 5.7.

$$\sqsupseteq_3 \star \lfloor I_m\, \sigma' \wedge (v_m\, \sigma' < v_m\, \grave{}\sigma) \wedge ((x', y') = (\grave{}x, \grave{}y)) \rfloor; \boxed{\lfloor z' = \grave{}x \times \grave{}y \rfloor}$$

We open a subwindow to concentrate on the original requirement.

Window: 6

$$
\begin{array}{l}
\vdots \\
! \ \boxed{\forall \sigma_1\, \sigma_2.\, I_m\, \sigma_1 \wedge (v_m\, \sigma_1 < v_m\, \grave{}\sigma) \supset (M(\sigma_1, \sigma_2) \sqsupseteq_3 O_m(\sigma_1, \sigma_2))} \\
! \ I_m\, \grave{}\sigma \\
! \ v_m\, \grave{}\sigma < v_m\, \grave{}\grave{}\sigma \\
\sqsupseteq_3 \star \lfloor z' = \grave{}x \times \grave{}y \rfloor
\end{array}
$$

In this subwindow, we may assume that the invariant holds and that the variant has decreased. We know that in such states, a recursive call to the label M will implement the specification O_m. The current focus is an instance of O_m, so we open a subwindow to derive the lemma required to refine the focus into a recursive call to the label.

Window: 7

$$
\begin{array}{l}
\vdots \\
! \ I_m\, \grave{}\sigma \\
! \ v_m\, \grave{}\sigma < v_m\, \grave{}\grave{}\sigma \\
\subset \star \forall \sigma_1\, \sigma_2.\, I_m\, \sigma_1 \wedge (v_m\, \sigma_1 < v_m\, \grave{}\sigma) \supset (M(\sigma_1, \sigma_2) \sqsupseteq_3 O_m(\sigma_1, \sigma_2))
\end{array}
$$

We instantiate the quantified states in the focus to be the current initial and final states. We then simplify the focus further using the assumptions in the context and the definition of O_m.

$$
\begin{array}{l}
\subset \star I_m\, \grave{}\sigma \wedge (v_m\, \grave{}\sigma < v_m\, \grave{}\grave{}\sigma) \supset (M(\grave{}\sigma, \sigma') \sqsupseteq_3 O_m(\grave{}\sigma, \sigma')) \\
\subset \star M(\grave{}\sigma, \sigma') \sqsupseteq_3 O_m(\grave{}\sigma, \sigma') \\
\subset \star M(\grave{}\sigma, \sigma') \sqsupseteq_3 \lfloor z' = \grave{}x \times \grave{}y \rfloor
\end{array}
$$

Having derived the fact we require, we close this window.

Window: 6

$$
\begin{array}{l}
\vdots \\
¡ \ M(\grave{}\sigma, \sigma') \sqsupseteq_3 \lfloor z' = \grave{}x \times \grave{}y \rfloor \\
\sqsupseteq_3 \star \lfloor z' = \grave{}x \times \grave{}y \rfloor
\end{array}
$$

We may now use the lemma just derived to implement the focus with a recursive call to the label M.

$$
\sqsupseteq_3 \star M(\grave{}\sigma, \sigma')
$$

We then close this window and return to its parent.

Window: 5

$$
\begin{array}{l}
\vdots \\
\mathbin{¡}\ `z = `x \times `t \\
\mathbin{¡}\ `t \leq `y \\
!\ `t \neq `y \\
\sqsupseteq_3 \star \boxed{\lfloor I_m\ \sigma' \wedge (v_m\ \sigma' < v_m\ `\sigma) \wedge ((x', y') = (`x, `y))\rfloor}; M(`\sigma, \sigma')
\end{array}
$$

In this window, the requirement to decrease the variant while preserving the invariant remains unimplemented. We therefore open a subwindow to concentrate on implementing this requirement.

Window: 6

$$
\begin{array}{l}
\vdots \\
\mathbin{¡}\ `z = `x \times `t \\
\mathbin{¡}\ `t \leq `y \\
!\ `t \neq `y \\
\sqsupseteq_3 \star \lfloor I_m\ \sigma' \wedge (v_m\ \sigma' < v_m\ `\sigma) \wedge ((x', y') = (`x, `y))\rfloor
\end{array}
$$

The first step is to expand with the definitions of I_m and v_m.

$$\sqsupseteq_3 \star \lfloor (z' = x' \times t') \wedge (t' \leq y') \wedge \boxed{(|t' - y'| < |`t - `y|)} \wedge ((x', y') = (`x, `y))\rfloor$$

We then open a subwindow to refine the requirement to decrease the variant.

Window: 7

$$
\begin{array}{l}
\vdots \\
\mathbin{¡}\ `t \leq `y \\
!\ t' \leq y' \\
!\ ((x', y') = (`x, `y)) \\
\supset \star |t' - y'| < |`t - `y|
\end{array}
$$

From our assumptions we know that $`t \leq `y$ and $t' \leq y'$, so we can remove the absolute value expressions. The focus can then be further simplified using the other assumptions.

$$
\begin{array}{l}
\supset \star t' - y' < `t - `y \\
\supset \star t' - `y < `t - `y \\
\supset \star t' < `t
\end{array}
$$

We have now transformed the requirement to decrease the variant into a requirement to increase the value of t. We can refine this requirement by choosing to increase the value of t by a particular amount—we choose to increase it by one.

$$\supset \star t' = `t + 1$$

We now close this window.

Window: 6

$$
\begin{array}{l}
\vdots \\
\text{¡ } `z = `x \times `t \\
\text{¡ } `t \leq `y \\
\text{! } `t \neq `y \\
\sqsupseteq_3 \star \lfloor (z' = x' \times t') \wedge \boxed{(t' \leq y')} \wedge (t' = `t + 1) \wedge ((x', y') = (`x, `y)) \rfloor
\end{array}
$$

Next, we open a subwindow on the requirement $t' \leq y'$.

Window: 7

$$
\begin{array}{l}
\vdots \\
\text{¡ } `t \leq `y \\
\text{! } `t \neq `y \\
\text{! } t' = `t + 1 \\
\text{! } (x', y') = (`x, `y) \\
\supset \star\, t' \leq y'
\end{array}
$$

Under the assumptions of this window, the focus can be transformed to true.

$$
\begin{array}{l}
\supset \star\, `t + 1 \leq `y \\
\supset \star\, `t < `y \\
\supset \star\, (`t \leq `y) \wedge (`t \neq `y) \\
\supset \star\, \mathsf{T}
\end{array}
$$

This subwindow is then closed.

Window: 6

$$
\begin{array}{l}
\vdots \\
\text{¡ } `z = `x \times `t \\
\text{! } `t \neq `y \\
\sqsupseteq_3 \star \lfloor (z' = x' \times t') \wedge \mathsf{T} \wedge (t' = `t + 1) \wedge ((x', y') = (`x, `y)) \rfloor
\end{array}
$$

We simplify the focus of this window, and then open a subwindow on the requirement to establish $z' = x' \times t'$.

$$
\sqsupseteq_3 \star \lfloor \boxed{(z' = x' \times t')} \wedge (t' = `t + 1) \wedge ((x', y') = (`x, `y)) \rfloor
$$

Window: 7

$$
\begin{array}{l}
\vdots \\
\text{¡ } `z = `x \times `t \\
\text{! } t' = `t + 1 \\
\text{! } (x', y') = (`x, `y) \\
\supset \star\, z' = x' \times t'
\end{array}
$$

Using the assumptions of this window, the focus may be simplified as follows:

$$\supset \star z' = \grave{}x \times (\grave{}t + 1)$$
$$\supset \star z' = \grave{}x \times \grave{}t + \grave{}x$$
$$\supset \star z' = \grave{}z + \grave{}x$$

We then close the window.

Window: 6

$$\vdots$$
$$\sqsupseteq_3 \star \lfloor (z' = \grave{}z + \grave{}x) \wedge (t' = \grave{}t + 1) \wedge ((x', y') = (\grave{}x, \grave{}y)) \rfloor$$

The focus of this window can now be implemented with an assignment statement using the procedure described in section 5.4.

$$\sqsupseteq_3 \star (z', t') := (\grave{}z + \grave{}x, \grave{}t + 1)$$

The implementation of this window is now complete, so we close it and return to its parent.

Window: 5

$$\vdots$$
$$\sqsupseteq_3 \star (z', t') := (\grave{}z + \grave{}x, \grave{}t + 1); M(\grave{}\sigma, \sigma')$$

This window is also complete, so it too is closed.

Window: 4

$$\vdots$$
$$\sqsupseteq_3 \star \mathsf{if}\ \grave{}t = \grave{}y\ \mathsf{then\ skip\ else}\ (z', t') := (\grave{}z + \grave{}x, \grave{}t + 1); M(\grave{}\sigma, \sigma')$$

This window can also be closed.

Window: 3

$$\vdots$$
$$\sqsupseteq_3 \star \mathsf{label}_{(v_m, O_m, I_m)}\, M.$$
$$\mathsf{if}\ \grave{}t = \grave{}y\ \mathsf{then\ skip\ else}\ (z', t') := (\grave{}z + \grave{}x, \grave{}t + 1); M(\grave{}\sigma, \sigma')$$

The development of the recursive block is now complete. We may therefore drop the subscripts from the label by replacing the approximate meaning for the block by its true meaning. The correctness of this step follows from the theorem proved in section 5.9.

$$\sqsupseteq_3 \star \mathsf{label}\ M.\ \mathsf{if}\ \grave{}t = \grave{}y\ \mathsf{then\ skip\ else}\ (z', t') := (\grave{}z + \grave{}x, \grave{}t + 1); M(\grave{}\sigma, \sigma')$$

We now close this window.

Window: 2

$$\sqsupseteq_3 \star \lfloor I_m\, \sigma' \wedge ((x', y') = (\grave{}x, \grave{}y))\rfloor;$$
$$\quad \mathsf{label}\ M.\ \mathsf{if}\ \grave{}t = \grave{}y\ \mathsf{then\ skip\ else}\ (z', t') := (\grave{}z + \grave{}x, \grave{}t + 1);\ M(\grave{}\sigma, \sigma')$$

The initialisation to establish the invariant remains unimplemented. First we expand with the expression $I\ \grave{}\sigma$.

$$\sqsupseteq_3 \star \lfloor (z' = x' \times t') \wedge (t' \leq y') \wedge ((x', y') = (\grave{}x, \grave{}y))\rfloor;$$
$$\quad \mathsf{label}\ M.\ \mathsf{if}\ \grave{}t = \grave{}y\ \mathsf{then\ skip\ else}\ (z', t') := (\grave{}z + \grave{}x, \grave{}t + 1);\ M(\grave{}\sigma, \sigma')$$

It is then easy to see that the invariant can be established by assigning the value zero to both z and t. The assignment statement is introduced using the procedure described in section 5.4.

$$\sqsupseteq_3 \star (z, t) := (0, 0);$$
$$\quad \mathsf{label}\ M.\ \mathsf{if}\ \grave{}t = \grave{}y\ \mathsf{then\ skip\ else}\ (z', t') := (\grave{}z + \grave{}x, \grave{}t + 1);\ M(\grave{}\sigma, \sigma')$$

The development of this window is now complete, so we close it.

Window: 1

$$\sqsupseteq_3 \star \mathsf{var}\ t.\ (z, t) := (0, 0);$$
$$\quad \mathsf{label}\ M.\ \mathsf{if}\ \grave{}t = \grave{}y\ \mathsf{then\ skip\ else}\ (z', t') := (\grave{}z + \grave{}x, \grave{}t + 1);\ M(\grave{}\sigma, \sigma')$$

We have now returned to the top level of the window stack and the focus is completely implemented. This completes the development of the example. The result of the derivation is the following theorem:

$$\vdash \lfloor z' = \grave{}x \times \grave{}y\rfloor \sqsubseteq_3$$
$$\quad \mathsf{var}\ t.\ (z, t) := (0, 0);$$
$$\quad\quad \mathsf{label}\ M.\ \mathsf{if}\ \grave{}t = \grave{}y\ \mathsf{then\ skip\ else}\ (z', t') := (\grave{}z + \grave{}x, \grave{}t + 1);\ M(\grave{}\sigma, \sigma')$$

5.11 Verification

This dissertation is primarily concerned with the development of a method of program refinement. However, it is sometimes argued that a refinement approach can be counterproductive in situations where the specification is so simple that an implementation is obvious. In such situations is may be more productive to adopt a verification style approach.

Window inference could be used to do verification style proofs as well as refinement proofs. The goal of both processes is to construct a theorem of the form $\vdash$ *Spec* $\sqsubseteq_3$ *Imp*. This can be accomplished either by transforming *Spec* into *Imp* while preserving $\sqsubseteq_3$ (program refinement), or by transforming *Spec* $\sqsubseteq_3$ *Imp* into true while preserving $\supset$ (program verification). Indeed the two techniques could be combined within the one system. Window inference could be used to pursue a development up to the point where the remainder of its implementation is obvious, at which point verification is used to prove a lemma that completes the transformation.

5.12 Conclusions

This chapter presented a method of program refinement based on combining the unified treatment of specifications and programs as three valued predicates with the window-inference method of transformational reasoning. The result is a simple and flexible method of refinement. Adopting a predicative approach means that the user can easily pursue the development in any direction that can be justified from the semantics of the target language. Programming language constructs can be introduced into a refinement to either implement or decompose a specification. Using window inference allows subwindows to be opened on the subclauses of a program construct to further pursue their refinement. Window inference also gives the user access to contextual information when refining subclauses. For example, when refining the second clause of a sequential composition, it is possible to assume that the first clause has just been executed. The effective use of contextual information is an important part of any refinement method. Back describes the use of context in his calculus in his paper *A Calculus of Refinements for Program Derivations* [Bac88]. Window inference provides a more general and effective method of managing contextual information than previous styles of reasoning.

The task of introducing and refining labelled blocks is complicated because little can be assumed about the form of a block during the intermediate stages of its development. For example, the body of a partially developed block need not be a monotonic function of its label. It is also difficult to tell from which initial states, if any, the execution of a block will terminate. The method of refinement presented in this chapter avoids these difficulties by using an approximation to the meaning of a recursive block. The approximation is accurate enough to support a natural method of introducing and refining blocks. When the refinement process is complete and the body of the block is executable, then the function of the block is known to be monotonic. The approximate meaning can then be replaced by the true meaning. This trick is presented here in conjunction with the three-valued predicative semantics used throughout the dissertation, but it could be applied with equal effectiveness to refinement methods based on other formalisms.

This chapter identified a number of frequently repeated sequences of refinement steps. Each of these sequences constitutes a single conceptual refinement. Further use of the refinement method proposed would doubtless reveal additional sequences of refinement steps worthy of special consideration. Such sequences could be mechanised, in part or in whole, by program refinement tool constructed using a window-inference interface to an LCF-style theorem prover. The users of such a tool could write their own functions using the meta-language of the theorem prover to mechanise any sequence of refinements they frequently repeated.

A small refinement example was presented at the end of the chapter. This example demonstrates how window inference can be used as a method of program refinement. Note that even though the method is based on using a three-valued logic, all the

detailed reasoning performed in the the example was two-valued.

The development of the example was given structure by assigning each window a number, depending upon its depth. This structure can then be used to browse the refinement rather than simply reading it from end to end. The ability to browse a refinement is particularly useful for large developments, such as the case study presented in chapter 6. Additional comments on the value of the structure imposed on a development by using window inference can be found in section 6.10.

Chapter 6

A Refinement Case Study

6.1 Introduction

This chapter presents a refinement case study. The example chosen is a simple line editor, the sort you might find on a hand-held computer. Although neither large nor complex by the standards of today, the line editor is a nontrivial example that contains features characteristic of real applications. For example, it requires complex data structures (arrays and pointers) and uses data abstraction to provide a simple view of them. The line editor is a complex enough example to demonstrate that the specification and refinement techniques proposed in this dissertation scale up to handle nontrivial problems; but is simple enough that a significant proportion of the development can be presented in the space available.

The line editor is the interface to a hypothetical hand-held computer called the *Generic-PC*. Users compose instructions for the *Generic-PC* with the line editor before having them interpreted by pressing the 'enter' key. The computer displays the results of executing the instructions, then clears the line and returns the user to the editor. Users can insert and delete characters, and move backward and forward along the line. If the user makes an editing error, like trying to delete a character from an empty line, an error light is lit. The light remains on until the user performs a valid editing operation.

The development begins with a specification of the line editor. We construct our specification starting with a simple model, which we elaborate until reaching the desired level of detail. Readers seeking compare this specification with those of similar systems are referred to [Ale85, Ale88, Chi85, Nei90, Suf82]. Next, we present a summary of the development. Because of their size, the details of the development are left until Appendix B. During the specification and development, we assume the existence of some types (pairs, lists and arrays), and some functions on those types. All the relevant properties of these types and functions are described in Appendix A. At the end of the chapter we draw some conclusions about the suitability of the specification and refinement techniques for the development of large systems.

6.2 The Hardware

No program, or specification, can be considered in isolation from its environment. The environment of our line editor is the *Generic-PC*. Figure 6.1 is a picture of the *Generic-PC*.

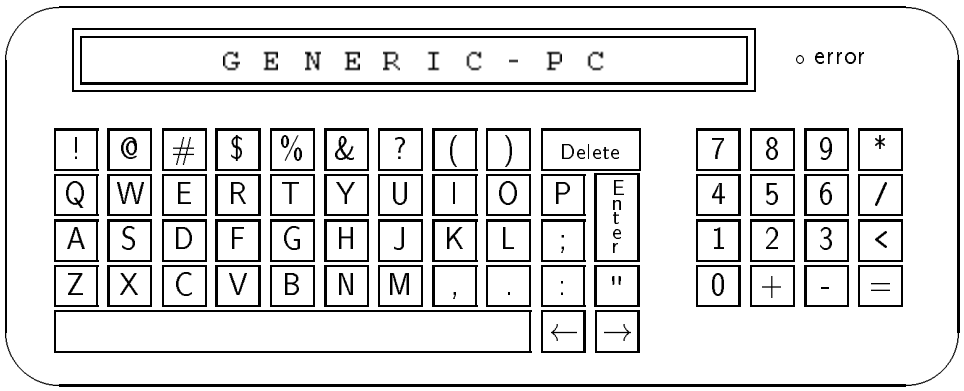


Figure 6.1: The *Generic-PC*

The screen of the *Generic-PC* can display a variety of symbols. These symbols, known as *glyphs*, are mostly pictures of characters. We assume there is a type *GLYPH* which contains these symbols. There is a glyph corresponding to each of the graphic characters. These glyphs are denoted by characters written in a dot-matrix type face. There is also a glyph corresponding to the space character, which we denote with a blank. There is one additional glyph, the empty glyph, which we write as a single point. If a line being edited does not occupy the whole of the screen, the unoccupied positions will display the empty glyph. This allows the user to distinguish between unoccupied positions and positions displaying a representation of the space character. A typical definition of *GLYPH* appears below (the last two glyphs in the definition are the space and empty glyphs).

$$
\begin{array}{lll}
GLYPH & \stackrel{\text{def}}{=} & \mathtt{A} \mid \mathtt{B} \mid \mathtt{C} \mid \mathtt{D} \mid \mathtt{E} \mid \mathtt{F} \mid \mathtt{G} \mid \mathtt{H} \mid \mathtt{I} \\
 & & \mid \mathtt{J} \mid \mathtt{K} \mid \mathtt{L} \mid \mathtt{M} \mid \mathtt{N} \mid \mathtt{O} \mid \mathtt{P} \mid \mathtt{Q} \mid \mathtt{R} \\
 & & \mid \mathtt{S} \mid \mathtt{T} \mid \mathtt{U} \mid \mathtt{V} \mid \mathtt{W} \mid \mathtt{X} \mid \mathtt{Y} \mid \mathtt{Z} \mid \mathtt{0} \\
 & & \mid \mathtt{1} \mid \mathtt{2} \mid \mathtt{3} \mid \mathtt{4} \mid \mathtt{5} \mid \mathtt{6} \mid \mathtt{7} \mid \mathtt{8} \mid \mathtt{9} \\
 & & \mid \mathtt{,} \mid \mathtt{;} \mid \mathtt{:} \mid \mathtt{.} \mid \mathtt{!} \mid \mathtt{?} \mid \mathtt{"} \mid \mathtt{*} \mid \mathtt{+} \\
 & & \mid \mathtt{-} \mid \mathtt{=} \mid \mathtt{<} \mid \mathtt{(} \mid \mathtt{)} \mid \mathtt{/} \mid \mathtt{\#} \mid \mathtt{\&} \mid \mathtt{@} \\
 & & \mid \mathtt{\$} \mid \mathtt{\%} \mid \mathtt{\ } \mid \mathtt{\cdot}
\end{array}
$$

The screen is divided into a fixed number of positions, numbered from zero up to and including *MaxPos*. At each position, the screen displays a glyph. We assume the existence of an inbuilt array of glyphs called *screen*. (Arrays are described in

Appendix A). The hardware of the *Generic-PC* ensures that the physical screen mirrors the contents of *screen*.

screen : array *MaxPos* of *GLYPH*

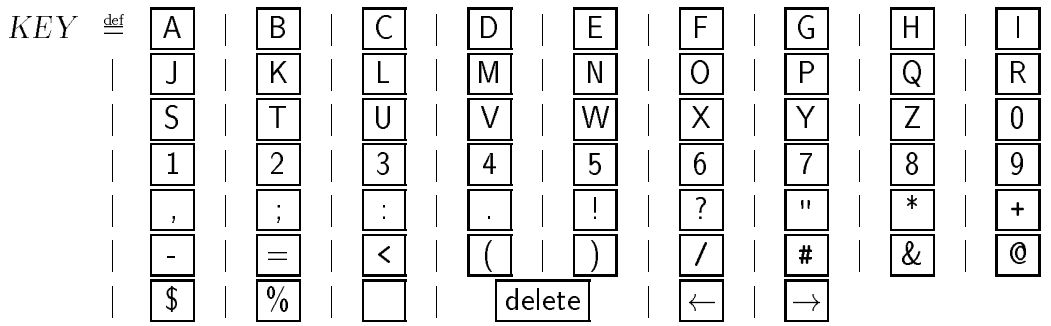
The screen can underline one of its positions. This underlining is used to indicate the current position when editing a line. We assume there exists an inbuilt numeric variable *cursor*. If *cursor* contains a value between zero and *MaxPos*, the hardware of the *Generic-PC* ensures that position is underlined.

cursor : *NAT*

Just to the right of the screen lies a small light with 'error' written next to it. The light should switch on whenever the user attempts an invalid editing operation. We assume the existence of an inbuilt boolean variable *error*. The hardware of the *Generic-PC* ensures that the light is on whenever *error* is true and off otherwise.

error : *BOOL*

Most of the keys on they keyboard of the *Generic-PC* correspond to characters. The keyboard also contains several keys which do not correspond to characters: the 'enter' key, the 'delete' key and the arrow keys. The enter key is not used by the editor, and for our purposes it is not considered to be a real key. We assume a type *KEY* which contains the keys on the keyboard. Elements of *KEY* are denoted by symbols enclosed in a box. The following is a possible definition of *KEY*.



6.3 The System Software

The system software of the *Generic-PC* is very simple. It initialises the editor, then enters an infinite loop. The loop polls the keyboard waiting for the user to press a key. If the enter key is pressed, the text being edited is interpreted as a command, the results are presented to the user, and the editor is reinitialised. If any other key is pressed, the variable *key* is set to reflect this and the editor is called.

key : *KEY*

The editor is expected to perform whatever function is associated with the key that was pressed.

The task of the line editor is to manipulate a line of characters. The type of characters is *CHAR*. Objects of type *CHAR* that denote graphic characters are written as characters in a type-writer like type face. The space character is written ␣. A typical definition of *CHAR* is shown below:

$$
\begin{array}{lcl}
CHAR & \stackrel{\text{def}}{=} & \mathtt{A} \mid \mathtt{B} \mid \mathtt{C} \mid \mathtt{D} \mid \mathtt{E} \mid \mathtt{F} \mid \mathtt{G} \mid \mathtt{H} \mid \mathtt{I} \\
 & & \mid \mathtt{J} \mid \mathtt{K} \mid \mathtt{L} \mid \mathtt{M} \mid \mathtt{N} \mid \mathtt{O} \mid \mathtt{P} \mid \mathtt{Q} \mid \mathtt{R} \\
 & & \mid \mathtt{S} \mid \mathtt{T} \mid \mathtt{U} \mid \mathtt{V} \mid \mathtt{W} \mid \mathtt{X} \mid \mathtt{Y} \mid \mathtt{Z} \mid \mathtt{0} \\
 & & \mid \mathtt{1} \mid \mathtt{2} \mid \mathtt{3} \mid \mathtt{4} \mid \mathtt{5} \mid \mathtt{6} \mid \mathtt{7} \mid \mathtt{8} \mid \mathtt{9} \\
 & & \mid \mathtt{,} \mid \mathtt{;} \mid \mathtt{:} \mid \mathtt{.} \mid \mathtt{!} \mid \mathtt{?} \mid \mathtt{"} \mid \mathtt{*} \mid \mathtt{+} \\
 & & \mid \mathtt{-} \mid \mathtt{=} \mid \mathtt{<} \mid \mathtt{(} \mid \mathtt{)} \mid \mathtt{/} \mid \mathtt{\#} \mid \mathtt{\&} \mid \mathtt{@} \\
 & & \mid \mathtt{\$} \mid \mathtt{\%} \mid \mathtt{\sqcup}
\end{array}
$$

We assume the existence of three functions for converting values between the types *GLYPH*, *KEY* and *CHAR*: *char2glyph*, *is_char_key* and *key2char*. Characters are mapped to their corresponding glyphs by the function *char2glyph*.

$$char2glyph : CHAR \to GLYPH$$

The function *is_char_key* is true of those keys that correspond to characters and false of all others.

$$is\_char\_key : KEY \to BOOL$$

For those keys that do correspond to characters, the function *key2char* maps keys to the corresponding characters.

$$key2char : KEY \to CHAR$$

The interpreter of the *Generic-PC* expects to find the text of the line being edited in an array of characters called *line*. The elements of the array are numbered from zero up to and including *MaxChar*.

$$line : \mathsf{array}\ MaxChar\ \mathsf{of}\ CHAR$$

The maximum possible length of a line is therefore $MaxChar + 1$ characters.

$$\vdash^{\text{def}} MaxLength = MaxChar + 1 \qquad [MaxLength\_\mathsf{DEF}]$$

The interpreter will also need to know the length of the line. This will be stored in a variable called *length*.

$$length : NAT$$

6.4 A Simple Editor

Our task is to specify the two operations *INIT* and *EDIT*, which initialise the editor and perform an editing operation. The editing operations are intended to modify the line stored in the variables *line* and *length*. Editing operations affect the line at the current position. Therefore, in addition to the contents and length of a line, we also need to store the current position within the line. We use a numeric variable *current* to store this position.

$$current : NAT$$

Although we could specify the desired behaviour of the editor in terms of *line*, *length* and *current*, some aspects of the specification will be clearer if we use an abstract representation of the line. For these purposes we will represent a line as a pair of lists of characters. The first list contains those characters lying before the current position, and the second list contains those that lie after it. The current position is implicitly defined as being between the two lists. This model of lines is due to Sufrin [Suf82]. We define the type *LINE* as an abbreviation for the type of pairs of lists of characters (the list type is described in Appendix A).

$$LINE \stackrel{\text{def}}{=} (CHAR\,\mathsf{list}) \times (CHAR\,\mathsf{list})$$

We say that a pair of lists (bef, aft) is an *abstraction* of *line*, *length*, and *current* if they are related by the predicate *IsAbs* defined below. (The operators: '#', '@', '⌢' and '↓' are described in Appendix A.)

$$\begin{aligned} \vdash^{\text{def}} & IsAbs((bef, aft), line, length, current) = \\ & \quad length = \#bef + \#aft \\ & \wedge \quad current = \#bef \\ & \wedge \quad length \leq MaxLength \\ & \wedge \quad \Big(\forall i.\, (i < length) \supset (line@i = (bef \frown aft)\downarrow i)\Big) \end{aligned} \qquad [IsAbs\_\mathsf{DEF}]$$

While framing the specification, we assume the existence of two variables of type *LINE*: l_1 and l_2. We assume that l_1 is an abstraction of the initial values of *line*, *length* and *current*. Where necessary, the specification will compel the final values of *line*, *length* and *current* to be such that l_2 is an abstraction of them. Those aspects of the editor more clearly specified using the abstract representation are phrased as predicates on l_1 and l_2; other aspects are phrased as predicates on the concrete variables.

The Core of the Editor

To help with the specification of the editing functions, we now define some basic constants and functions for lines represented as pairs of lists. (The symbols: '⟨ ⟩', '◁' and '▷' are described in Appendix A).

$empty\_line : LINE$
The empty line.

$$\models^{\text{def}} empty\_line = (\langle\,\rangle, \langle\,\rangle) \qquad [empty\_line\_\mathsf{DEF}]$$

$del : LINE \to LINE$
If there is a character to the left of the current position, then the function del deletes it.

$$\models^{\text{def}} del(bef \triangleleft c, aft) = (bef, aft) \qquad [del\_\mathsf{DEF}]$$

$ins : CHAR \to LINE \to LINE$
If there is room in the line, the functions ins inserts a character before the current position.

$$\models^{\text{def}} ((\#bef + \#aft) < MaxLength) \supset (ins\ c\ (bef, aft) = (bef \triangleleft c, aft)) \qquad [ins\_\mathsf{DEF}]$$

$left : LINE \to LINE$
If the current position is not already the leftmost possible, the function $left$ moves the current position one place to the left.

$$\models^{\text{def}} left(bef \triangleleft c, aft) = (bef, c \triangleright aft) \qquad [left\_\mathsf{DEF}]$$

$right : LINE \to LINE$
If the current position is not already the rightmost possible, the function $right$ moves the current position one place to the right.

$$\models^{\text{def}} right(bef, c \triangleright aft) = (bef \triangleleft c, aft) \qquad [right\_\mathsf{DEF}]$$

Using these definitions, we now have a first attempt at specifying the operations $INIT$ and $EDIT$. We call these first attempts $INIT\,1$ and $EDIT\,1$. For the sake of simplicity, $INIT\,1$ and $EDIT\,1$ ignore the appearance of the screen and the correct handling of errors.

The initialisation operation sets the abstract representation of the line to be empty and requires the concrete variables to mirror the abstract state. Initialisation must always terminate. We may therefore describe the initialisation with an ordinary two-valued predicate lifted up to the domain of three-valued predicates used for specifications.

$$\models^{\text{def}} INIT\,1(l_2, line', length', current') = \lfloor (l_2 = empty\_line) \wedge IsAbs(l_2, line', length', current') \rfloor \qquad [INIT\,1\_\mathsf{DEF}]$$

The edit operation modifies the line in accordance with whatever key has been pressed. Note that this operation must always terminate, and the final state of the concrete variables must mirror the abstract representation of the line. Since the edit operation must always terminate, we can specify it with a two-valued predicate lifted up to the domain of three-valued predicates.

$$\vdash^{\text{def}} EDIT\,1(\grave{}key, l_1, l_2, line', length', current') = \left\lfloor \begin{array}{ll} & (\grave{}key = \boxed{\mathsf{delete}}) \supset (l_2 = del\ l_1) \\ \wedge & (is\_char\_key\ \grave{}key) \supset (l_2 = ins\ (key2char\ \grave{}key)\ l_1) \\ \wedge & (\grave{}key = \boxed{\leftarrow}) \supset (l_2 = left\ l_1) \\ \wedge & (\grave{}key = \boxed{\rightarrow}) \supset (l_2 = right\ l_1) \\ \wedge & IsAbs(l_2, line', length', current') \end{array} \right\rfloor \qquad [EDIT\,1\_\mathsf{DEF}]$$

Note that this specification describes the desired behaviour of the editor in response to any key press.

$$\vdash \forall k.\ (k = \boxed{\mathsf{delete}}) \vee (is\_char\_key\ k) \vee (k = \boxed{\leftarrow}) \vee (k = \boxed{\rightarrow}) \qquad [KEY\_\mathsf{THM}]$$

6.5 Handling Errors

Some editing operations are not always valid. In particular, the following are editing errors:

- attempting to delete a character when there is nothing to the left of the current position;
- attempting to insert a character when the line is already at its maximum length;
- attempting to move the current position to the left when it is already at the left-most position in the line;
- attempting to move the current position to the right when it is already at the right-most position in the line.

The following definition of *Error* describes the possible errors.

$$\vdash^{\text{def}} Error(key, length, current) = \begin{array}{ll} & (key = \boxed{\mathsf{delete}}) \wedge (current = 0) \\ \vee & (is\_char\_key\ key) \wedge (length = MaxLength) \\ \vee & (key = \boxed{\leftarrow}) \wedge (current = 0) \\ \vee & (key = \boxed{\rightarrow}) \wedge (current = length) \end{array} \qquad [Error\_\mathsf{DEF}]$$

We can now extend the definitions of the initialisation and editing operations so they correctly handle editing errors. The initialisation operation should ensure that the error light is off.

$$\models^{\text{def}} INIT\,2(l_2, line', length', current', error') = \\ INIT\,1(l_2, line', length', current') \mathbin{\underline{\triangle}} \lfloor \neg error' \rfloor \qquad [INIT\,2\_\text{DEF}]$$

The clauses of *EDIT 1* do not specify the final value of the line if an error is made. In such situations the line should be unchanged. The edit operation should also set the status of the error light according to whether or not and error was made.

$$\models^{\text{def}} EDIT\,2(`key, l_1, l_2, `line, `length, `current, line', length', current', error') = \\ \left(\begin{array}{ll} Error(`key, `line, `length, `current) & \\ \Rightarrow & \left\lfloor \begin{array}{ll} & error' \\ \wedge & l_2 = l_1 \\ \wedge & IsAbs(l_2, line', length', current') \end{array} \right\rfloor \\ \mid & \left(\begin{array}{ll} & EDIT\,1(`key, l_1, l_2, line', length', current') \\ \underline{\triangle} & \lfloor \neg error' \rfloor \end{array} \right) \end{array} \right) \qquad [EDIT\,2\_\text{DEF}]$$

6.6 The Screen

This section extends the specification to cover the appearance of the screen. Recall from section 6.2 that the screen is modelled by two variables, *screen* and *cursor*. The variable *screen* is an array of glyphs with positions numbered from zero to *MaxPos* inclusive; *cursor* is a numeric variable. Note that the screen may be smaller than the maximum length of a line. This means that we must use the screen as a window onto the line. Let us assume the variable *offset* holds the position of the screen within the line.

$$offset : NAT$$

We say that the screen is a *view* of the line if each position on the screen displays the glyph representing the character at the corresponding position in the line. If the screen is located so that the line does not fill the whole screen, the remaining positions should display the empty glyph.

First, we define a function *glyph_at* which returns the glyph corresponding to the character at a selected position in a line, or the empty glyph if the selected position is after the end of the line.

$$\models^{\text{def}} glyph\_at(line, length, i) = (i < length \Rightarrow char2glyph\ line@i \mid \cdot) \qquad [glyph\_at\_\text{DEF}]$$

The concept of the screen being a view of the line can now be formalised as follows.

$$\models^{\text{def}} IsView(screen, line, length, offset) = \\ \left(\begin{array}{l} \forall n.\, (n \leq MaxPos) \supset \\ \qquad screen@n = glyph\_at(line, length, n + offset) \end{array} \right) \qquad [IsView\_\text{DEF}]$$

It is not enough for the screen to be a view of the line. All editing operations affect the line at the current position. We therefore demand that the screen always be positioned so that we can see the current position. If this were not the case, users would not be able to see what they were doing. The predicate *SeeCurrent* formalises the notion of being able to see the current position.

$$\vdash^{\text{def}} SeeCurrent(offset, current) = \\ 0 \leq current - offset \leq MaxPos \qquad [SeeCurrent\_\text{DEF}]$$

We also require that the cursor be positioned on the screen to indicate the current position. Logically, the current position is thought of as lying between two character positions. For example, if *current* is n, the current position lies between characters $n-1$ and n. However, the physical restrictions of the screen on the *Generic-PC* require us to mark the current position by underlining a position on the screen. We chose to underline the nth position if *current* is n.

$$\vdash^{\text{def}} CursorCurrent(cursor, offset, current) = \\ cursor = |current - offset| \qquad [CursorCurrent\_\text{DEF}]$$

Let us now combine all these conditions into a single predicate, *GoodScreen*:

$$\vdash^{\text{def}} GoodScreen(screen, cursor, offset, line, length, current) = \\ \begin{array}{ll} & IsView(screen, line, length, offset) \\ \wedge & SeeCurrent(offset, current) \\ \wedge & CursorCurrent(cursor, offset, current) \end{array} \qquad [GoodScreen\_\text{DEF}]$$

We now extend the definition of the initialisation operation so that it ensures the screen is correctly set up.

$$\vdash^{\text{def}} INIT\,3(l_2, line', length', current', error', screen', cursor', offset') = \\ \begin{array}{ll} & INIT\,2(l_2, line', length', current', error') \\ \triangleq & \lfloor GoodScreen(screen', cursor', offset', line', length', current') \rfloor \end{array} \qquad \begin{array}{l}[INIT\,3\_\text{DEF}] \\ \\ \\ \end{array}$$

In combination, these requirements insure that the screen is initially blank (displaying the empty glyph at every position) with the cursor at the left-most position.

It is desirable, but not sufficient, to impose these conditions on the final state of the editing operation as well. These conditions alone would allow the screen to move arbitrarily along the line as a result of an editing operation (as long as the current position was kept in view). It would be distracting if the screen were to jump around during editing. We would prefer it if the screen did not move along the line unless it was necessary to keep the current position in view, and then only by the minimum distance required. This concept is formalised by the predicate *MinMove* defined below:

$$\vdash^{\text{def}} MinMove(offset_1, offset_2, current) = \\ \left(\begin{array}{l} \forall o.\, SeeCurrent(o, current) \supset \\ \qquad |\, offset_1 - offset_2| \leq |offset_1 - o| \end{array} \right) \qquad [MinMove\_\text{DEF}]$$

We can now extend the definition of the edit operation to cover the appearance of the screen and the way the screen moves.

$$\begin{array}{l}
\vdash^{\text{def}} EDIT3(\grave{}key, l_1, l_2, \grave{}line, \grave{}length, \grave{}current, line', length', current', \\
\qquad error', \grave{}screen, \grave{}cursor, \grave{}offset, screen', cursor', offset') = \\
\left(\begin{array}{l}
GoodScreen(\grave{}screen, \grave{}cursor, \grave{}offset, \\
\qquad \grave{}line, \grave{}length, \grave{}current) \supset \\
\quad EDIT2(\grave{}key, l_1, l_2, \grave{}line, \grave{}length, \grave{}current, \\
\qquad line', length', current', error') \\
\underline{\wedge} \left\lfloor \begin{array}{l} GoodScreen(screen', cursor', offset', \\ \qquad line', length', current') \end{array} \right\rfloor \\
\underline{\wedge} \lfloor MinMove(\grave{}offset, offset', current') \rfloor
\end{array}\right)
\end{array} \qquad [EDIT3_\mathsf{DEF}]$$

This definition describes the behaviour of the editor under the assumption that the appearance of the screen is correct before the editing operation. It states that the line should be edited as before, and that the appearance of the screen should be updated so that it is correct once more after the editing operation, and that the screen should move by the minimum amount possible. If the screen is not correct before the editing operation, then any behaviour meets the specification represented by *EDIT3*.

6.7 Data Abstraction

For the purposes of specification, we have sometimes viewed lines as pairs of lists. We now describe the operation of the editor purely in terms of the concrete variables. We achieve this by internally quantifying the abstract variables in the specification.

The final version of the initialisation operation has the abstract variable l_2 hidden, as follows.

$$\begin{array}{l}
\vdash^{\text{def}} INIT(line', length', current', error', screen', cursor', offset') = \\
\left(\begin{array}{l} \underline{\exists} l_2 . INIT3(l_2, line', length', current', \\ \qquad error', screen', cursor', offset') \end{array}\right)
\end{array} \qquad [INIT_\mathsf{DEF}]$$

Encapsulating the *EDIT* operation is a little more difficult. While framing the specification of the editor, we have assumed that the variable l_1 is an abstraction of the initial concrete state of the editor. Unfortunately, there are some initial concrete states that do not correspond to any abstract state. These are the states where *current* is greater than *length*. Our specification ensures that the editor should never be placed in such a state, so we need not specify any behaviour for such states. For those initial states that *do* have a corresponding abstract state, we choose l_1 to be

such a state and then behave as before.[1]

$$\models^{\text{def}} EDIT(\grave{}key, \grave{}line, \grave{}length, \grave{}current, line', length', current', error', \grave{}screen, \grave{}cursor, \grave{}offset, screen', cursor', offset') = \left(\begin{array}{l} (\exists l_1.\ IsAbs(l_1, \grave{}line, \grave{}length, \grave{}current)) \supset \\ \left(\begin{array}{l} \mathsf{let}\ l_1 = (\varepsilon l_1.\ IsAbs(l_1, \grave{}line, \grave{}length, \grave{}current))\ \mathsf{in} \\ \underline{\exists} l_2.\ EDIT3(\grave{}key, l_1, l_2, \grave{}line, \grave{}length, \grave{}current, \\ \qquad line', length', current', error', \\ \qquad \grave{}screen, \grave{}cursor, \grave{}offset, \\ \qquad screen', cursor', offset') \end{array} \right) \end{array} \right) \qquad [EDIT\_\mathsf{DEF}]$$

We have now completed the description of the *INIT* and *EDIT* operations. These specifications have been phrased under the assumption that the state space σ is made up of at least the following program variables: *current*, *cursor*, *error*, *key*, *length*, *line*, *offset* and *screen*.

6.8 Comments on the Specification

The preceding sections of this chapter contain the specification of a line editor, thus demonstrating that the language of three-valued predicates proposed in this dissertation is adequate for specifying nontrivial applications.

Since, under normal operating conditions, all the operations of the *Generic-PC* are required to terminate, most of the specification has been phrased simply with two-valued predicates lifted up to the domain of three-valued predicates. The only exceptions to this are in the definition of *EDIT*, where if some desired preconditions fail to hold we do not wish to say anything about the behaviour the system, even whether it terminates or not. It is felt that this will be the case for most applications, so the potential complexity of reasoning with a three-valued logic can be largely avoided.

The method of data abstraction employed in this specification is slightly unusual in several respects. First, we have used both the concrete and abstract representations throughout the specification, depending on which offered the clearer method of capturing any particular requirement. Second, we have not proved any of the usual properties required of our abstraction. For example, we have not proved that every abstract state has a concrete representation. Rather, we have encapsulated the abstract states in such a way that if these properties do not hold when we expect them to (for example, if the editor were required to terminate in an abstract state that did not correspond to any concrete state), then the specification would require miraculous behaviour and thus be impossible to implement. This method of data-abstraction was chosen not because we were compelled to by our approach of using three-valued predicates, but because it led to a pleasing specification.

[1] The symbol 'ε' denotes Hilbert's choice function. For any predicate P involving x, if there is an x that satisfies P, then $\varepsilon x.\, P$ denotes such an x.

6.9 The Refinement

This section presents an overview of the development of the editor, the details of which can be found in Appendix B. The overview is intended to highlight the major design decisions of the development and, where appropriate, give pointers to the corresponding sections of the detailed presentation. After reading this section, it may be helpful to skim though Appendix B before returning to read the conclusions of the case-study.

The development is partitioned into three sections. The first describes the refinement of the initialisation operation, *INIT*. The second describes the refinement of the *EDIT* operation. During the development of *INIT*, we encounter a requirement to update the contents of the screen. This requirement will likely reoccur in the development of *EDIT*. Accordingly, we define a predicate *PAINT* that captures this requirement. *PAINT* is subsequently used in the implementation of both *INIT* and *EDIT*. The development of *PAINT* is presented in the third, and final section.

Refinement of *INIT*

The refinement of the initialisation operation begins in Appendix B on page 127, when *INIT* is made the focus of a window that preserves refinement. The first step is to expand the definition of *INIT*, which internally quantifies the abstract representation of the line l_2. We expand further definitions until we reach a conjunction of *INIT 2* and a requirement that the screen should correctly reflect the contents of the line. In Window 2 on page 128, these requirements are sequentially decomposed as follows.

$$\sqsupseteq_3 \star \mathit{INIT\,2}(l_2, \mathit{line}', \mathit{length}', \mathit{current}', \mathit{error}');$$
$$\left\lfloor \begin{array}{ll} & (\mathit{line}', \mathit{length}', \mathit{current}', \mathit{error}') = (\grave{}\mathit{line}, \grave{}\mathit{length}, \grave{}\mathit{current}, \grave{}\mathit{error}) \\ \wedge & \mathit{GoodScreen}(\mathit{screen}', \mathit{cursor}', \mathit{offset}', \grave{}\mathit{line}, \grave{}\mathit{length}, \grave{}\mathit{current}) \end{array} \right\rfloor$$

In Window 3 on page 128 we expand *INIT 2* into a conjunction of *INIT 1* and a requirement that *error*′ should be false. The latter requirement is implemented with an assignment.

$$\sqsupseteq_3 \star \mathit{INIT\,1}(l_2, \mathit{line}', \mathit{length}', \mathit{current}'); \mathit{error}' := \mathsf{F}$$

INIT 1 is expanded into a requirement that the abstract representation of the line should be empty and the concrete variables should represent the abstraction. After simplification, these requirements are implemented by assigning *length* and *current* the value zero in Window 4 on page 130. The implementation leaves behind a 'residue' of requirements on the abstract state, namely that l_2 should be empty.

$$\sqsupseteq_3 \star \lfloor l_2 = (\langle\rangle, \langle\rangle) \rfloor \mathrel{\underline{\triangle}} ((\mathit{length}', \mathit{current}') := (0, 0))$$

After closing several windows, we return to Window 1 on page 131. In this window the quantification of l_2 is in scope. Since all interrelationships between l_2

and the concrete variables have been removed, the abstraction can be dropped. This leaves us with the partial implementation shown below.

$$\begin{array}{l} \sqsupseteq_3 \star\,(length', current') := (0,0); \\ \quad error' := \mathsf{F}; \\ \quad \left\lfloor \begin{array}{ll} & (line', length', current', error') = (\grave{}line, \grave{}length, \grave{}current, \grave{}error) \\ \wedge & GoodScreen(screen', cursor', offset', \grave{}line, \grave{}length, \grave{}current) \end{array} \right\rfloor \end{array}$$

In Window 2 on page 132 we turn our attention to the requirements on the screen. After expanding the definition of *GoodScreen*, we decompose the resulting requirements into a sequential composition requiring us to first initialise the position of the screen and then update the contents of the screen. The requirement to update the screen will doubtless reoccur in the implementation of *EDIT*. Rather than implement these requirements now, we define a predicate *PAINT* to capture them. This predicate can be used in the implementations of both *INIT* and *EDIT*. We repeat the definition of paint here:

$$\begin{array}{l} \vdash^{\mathsf{def}} PAINT(\grave{}line, \grave{}length, \grave{}current, line', length', current', \\ \qquad \grave{}error, error', \grave{}offset, screen', cursor', offset') = \\ \left(\begin{array}{l} SeeCurrent(\grave{}cursor, \grave{}offset) \supset \\ \quad \left\lfloor \begin{array}{ll} & (offset', line', length', current', error') = \\ & \quad (\grave{}offset, \grave{}line, \grave{}length, \grave{}current, \grave{}error) \\ \wedge & IsView(screen', \grave{}line, \grave{}length, \grave{}offset) \\ \wedge & CursorCurrent(cursor', \grave{}offset, \grave{}current) \end{array} \right\rfloor \end{array} \right) \end{array} \qquad [PAINT\_\mathsf{DEF}]$$

The remaining requirement to initialise the position of the screen is satisfied by assigning *offset* the value zero. This concludes the development of *INIT*. The result of which is the following theorem:

$$\begin{array}{l} \vdash INIT(line', length', current', error', screen', cursor', offset') \sqsubseteq_3 \\ \quad (length', current') := (0,0); error' := \mathsf{F}; offset' := 0; \\ \quad PAINT(\grave{}line, \grave{}length, \grave{}current, line', length', current', \\ \qquad \grave{}error, error', \grave{}offset, screen', cursor', offset') \end{array}$$

Refinement of *EDIT*

The refinement of the *EDIT* operation is commenced on page 135 in Appendix B. The first step is to expand the definition of *EDIT*. The resulting specification is conditional on whether there is an abstraction of the initial state of the editor. The development proceeds under the assumption that there is such an abstraction. The requirements are framed in terms of two abstract variables, l_1 and l_2. Inside the scope of l_1 and l_2, we expand more definitions. From this we see that the requirements are also conditional on the initial state of the screen correctly reflecting the initial state

of the editor. Again, we assume that this is the case, since no particular behaviour is required if it is not.

In Window 3 on page 136 the specification requires us to edit the line, reposition the screen, and update the contents of the screen. We sequentially decompose these requirements as follows.

$$\sqsupseteq_3 \star \left(\begin{array}{ll} & EDIT2(\grave{}key, l_1, l_2, \grave{}line, \grave{}length, \grave{}current, \\ & \qquad line', length', current', error') \\ \underline{\triangle} & \lfloor \grave{}offset = offset' \rfloor \end{array} \right) ;$$
$$\left\lfloor \begin{array}{ll} & (line', length', current', error') = \\ & \quad (\grave{}line, \grave{}length, \grave{}current, \grave{}error) \\ \wedge & MinMove(\grave{}offset, offset', \grave{}current) \\ \wedge & SeeCurrent(offset', \grave{}current) \end{array} \right\rfloor ;$$
$$\left\lfloor \begin{array}{ll} & (offset', line', length', current', error') = \\ & \quad (\grave{}offset, \grave{}line, \grave{}length, \grave{}current, \grave{}error) \\ \wedge & IsView(screen', \grave{}line, \grave{}length, \grave{}offset) \\ \wedge & CursorCurrent(cursor', \grave{}offset, \grave{}current) \end{array} \right\rfloor$$

The last clause of the composition can be refined into an instance of the predicate $PAINT$.

The second clause of the composition is a requirement to reposition the screen so that the cursor is in view. We commence the development of this clause in Window 4 on page 138. The requirement to reposition the screen is partitioned into three cases: the case where no action is required, the case where we must move the screen to the left to bring the current position into view, and the case where we must move the screen to the right. The first case is implemented with a skip statement, the others are implemented with assignments. The result is this program fragment taken from Window 4 on page 143.

$\sqsupseteq_3 \star$ if $0 \leq \grave{}current - \grave{}offset \leq MaxPos$ then skip
else if $\grave{}current < \grave{}offset$ then $offset' := \grave{}current$
else $offset' := \grave{}current - MaxPos$

The development of the first clause of the composition, which must actually edit the line, is commenced in Window 4 on page 144. After expanding some definitions, we see that the requirements depend on whether the key pressed was the delete key, a character key, or the left or right arrow key. The development of the editor in its entirety would be too large to present in this dissertation. Therefore, we chose to develop only one of these cases and elide the others from the presentation. The delete and insert operations are of similar complexity, and both are significantly more complex than the movement operations. We therefore chose to develop only the delete operation, this operation being representative of the complexity of all the operations of the editor.

The development of the delete operation commences in Window 5 on page 146. After making some simplifications, the requirements for this operation are as follows:

$$\sqsupseteq_3 \star \quad \left(\begin{array}{l} `current = 0 \\ \quad \Rightarrow \quad \lfloor error' \wedge (l_2 = l_1) \wedge IsAbs(l_2, line', length', current') \rfloor \\ \quad \mathbf{I} \quad \left(\begin{array}{l} \quad \lfloor (l_2 = del\ l_1) \wedge IsAbs(l_2, line', length', current') \rfloor \\ \triangle \quad \lfloor \neg error' \rfloor \end{array}\right) \end{array}\right) \\ \triangle \quad \lfloor offset' = `offset \rfloor$$

The requirements are conditional on whether $`current$ is equal to zero. In the first case the user has made an error in pressing the delete key as there is no character to the left of the current position to delete. For this case an assignment sets *error* to true and leaves the line unchanged. The implementation leaves behind a residue of requirements on the abstract variables l_1 and l_2, namely that they should be equal. We are able to remove this residue later, in Window 1 on page 170.

In Window 6 on page 149 we commence the development of the case where the user has not made an error. Here, the character to the left of the current position must be deleted and the *error* flag set to false. The second requirement is implemented with an assignment. The remaining requirements are expressed in terms of a relationship between the abstract states and a requirement that the final concrete state should mirror the final abstract state. We must re-express these requirements purely in terms of a relationship between the concrete states. This process is completed in Window 7 on page 154; the results are shown below.

$$\sqsupseteq_3 \star \left|\begin{array}{ll} & length' = `length - 1 \\ \wedge & current' = `current - 1 \\ \wedge & length' \leq MaxLength \\ \wedge & \big(\forall i.\, (i < current') \supset (line'@i = `line@i)\big) \\ \wedge & \big(\forall i.\, (current' \leq i < length') \supset (line'@i = `line@(i+1))\big) \\ \wedge & offset' = `offset \end{array}\right|$$

The simplified requirements state that the length of the line and the current position should be decremented, and that the contents of the line above the current position should be moved down by one. The first of these requirements is implemented with a parallel assignment statement. The implementation of the remaining requirement is the most difficult part of the refinement, and the only part that had to be attempted twice. This requirement is implemented with a recursive block that works up the screen from the current position, moving the contents of each position down one as it goes. A local variable j is introduced to mark the working position in the line. The invariant and variant of the block are defined on page 157. Their definitions are repeated here:

$$\vdash^{\text{def}} I_d\ \sigma = (j \leq length) \wedge (length \leq MaxChar) \qquad [I_d\text{\_DEF}]$$

$$\models^{\text{def}} v_d\ \sigma = |length - j| \qquad [v_d\_\mathsf{DEF}]$$

We then prove that the invariant holds and introduce the block in Window 9 on page 158. The newly introduced block is shown below.

$$\sqsupseteq_3 \star \mathsf{label}_{(v_d, O_d, I_d)}\ D.$$
$$\left|\begin{array}{ll} & (length', current', offset') = (\grave{}length, \grave{}current, \grave{}offset) \\ \wedge & \Big(\forall i.\, (i < \grave{}j) \supset (line'@i = \grave{}line@i)\Big) \\ \wedge & \Big(\forall i.\, (\grave{}j \leq i < \grave{}length) \supset (line'@i = \grave{}line@(i+1))\Big) \end{array}\right|$$

Inside the block, we partition the requirements into two cases. The first, where j is already at the end of the line, is easily implemented with a skip statement. The second, where there are elements of the line above j yet to be moved down, is more difficult. We prefix this requirement with a specification that maintains the invariant, and moves a block of characters in the line down. In Window 13 on page 163 we derive the fact that the original requirement may now be refined by a recursive call to the label. Using this fact, we refine the second case in Window 11 on page 165, as shown below.

$$\sqsupseteq_3 \star \left|\begin{array}{ll} & \Big(\forall i.\, (i < \grave{}j) \supset (line'@i = \grave{}line@i)\Big) \\ \wedge & \Big(\forall i.\, (\grave{}j \leq i < j') \supset (line'@i = \grave{}line@(i+1))\Big) \\ \wedge & \Big(\forall i.\, (j' \leq i < \grave{}length + 1) \supset (line'@i = \grave{}line@i)\Big) \\ \wedge & j' \leq length' \\ \wedge & length' \leq MaxChar \\ \wedge & \grave{}j < j' \\ \wedge & (length', current', offset') = (\grave{}length, \grave{}current, \grave{}offset) \end{array}\right| ;$$
$$D(\grave{}\sigma, \sigma')$$

In Window 12 on page 165 we focus on implementing the code to move a block of characters down. We choose to move a block of size one, a task which can be accomplished with an assignment statement. This virtually completes the implementation, so we begin to close windows.

In Window 2 on page 170 we find that we have only implemented the case where the screen initially depicted the initial contents of the line. However, since no particular behaviour is required of the other case, we can use the same implementation for both. This window is also closed and we return to the top level. Here, in Window 1 on page 170, we have a similar situation, where we have only refined the case were there exists an abstraction of the initial state of the editor. Again, we may use the same implementation for both.

Finally, our implementation still contains a residue of requirements interrelating the abstract states. Fortunately, since these states are internally quantified in the

implementation, we can remove them all together. The result of this refinement process is the following theorem:

$$
\begin{array}{l}
\vdash EDIT(\grave{}key, \grave{}line, \grave{}length, \grave{}current, line', length', current', \\
\qquad error', \grave{}screen, \grave{}cursor, \grave{}offset, screen', cursor', offset') \sqsubseteq_3 \\
\quad \mathsf{if}\ \grave{}key = \boxed{\mathsf{delete}}\ \mathsf{then} \\
\qquad \mathsf{if}\ \grave{}current = 0\ \ \mathsf{then}\ error' := \mathsf{T} \\
\qquad \mathsf{else}\ (length', current') := (\grave{}length - 1, \grave{}current - 1); \\
\qquad\quad \mathsf{var}\, j.\, j' := \grave{}current; \\
\qquad\qquad \mathsf{label}\, D.\, \mathsf{if}\ \grave{}j = \grave{}length\ \ \mathsf{then\ skip} \\
\qquad\qquad\qquad \mathsf{else}\ (j', line') := (\grave{}j + 1, \grave{}line \dagger (\grave{}j, \grave{}line@(\grave{}j + 1))); \\
\qquad\qquad\qquad\quad D(\grave{}\sigma, \sigma'); \\
\qquad\quad error' := \mathsf{F} \\
\quad \mathsf{else} \ldots; \\
\quad \mathsf{if}\ 0 \leq \grave{}current - \grave{}offset \leq MaxPos\ \ \mathsf{then\ skip} \\
\quad \mathsf{else\,if}\ \grave{}current < \grave{}offset\ \ \mathsf{then}\ \ offset' := \grave{}current \\
\quad \mathsf{else}\ offset' := \grave{}current - MaxPos; \\
\quad PAINT(\grave{}line, \grave{}length, \grave{}current, line', length', current', \\
\qquad\quad \grave{}error, error', \grave{}offset, screen', cursor', offset')
\end{array}
$$

Refinement of *PAINT*

The refinement of *PAINT* begins on page 172. The definition of *PAINT* is conditional on the screen being correctly positioned. We assume that is the case, since no particular behaviour is required if it is not.

The requirements to update the screen comprise a requirement to update the position of the cursor and a requirement to update the glyphs appearing on the screen. These requirements are sequentially decomposed as shown below.

$$
\sqsupseteq_3 \star \left\lfloor \begin{array}{ll} & CursorCurrent(cursor', \grave{}offset, \grave{}current) \\ \wedge & (offset', line', length', current', error') = \\ & \quad (\grave{}offset, \grave{}line, \grave{}length, \grave{}current, \grave{}error) \end{array} \right\rfloor ; \left\lfloor \begin{array}{ll} & cursor' = \grave{}cursor \\ \wedge & (offset', line', length', current', error') = \\ & \quad (\grave{}offset, \grave{}line, \grave{}length, \grave{}current, \grave{}error) \\ \wedge & IsView(screen', \grave{}line, \grave{}length, \grave{}offset) \end{array} \right\rfloor
$$

The first requirement is implemented with an assignment in Window 3 on page 173.

The development of the second commences in Window 3 on page 174. The requirements on the screen state that each position should contain the glyph representing the character at the corresponding position in the line, or the empty glyph if it corresponds to a position beyond the end of the line. These requirements are implemented

with a recursive block that works up the screen setting the contents of each position as it goes. To this end, a local variable i is introduced to mark the working position.

Within the scope of i, we define the invariant and variant of the block. We repeat these definitions here:

$$\models^{\text{def}} I_p\ \sigma = \quad i \leq MaxPos + 1 \quad \wedge \quad \Big(\forall n.\,(n < i) \supset (screen@n = glyph\_at(line, length, n + offset))\Big) \qquad [I_p\text{\_DEF}]$$

$$\models^{\text{def}} v_p\ \sigma = |MaxPos + 1 - i| \qquad [v_p\text{\_DEF}]$$

We prefix our original requirement with a requirement to establish the invariant. The implementation of that requirement is delayed until Window 5 on page 189. The recursive block is then introduced.

$$\sqsupseteq_3 \star \mathsf{label}_{(v_p, O_p, I_p)}\ P. \left\lfloor \begin{array}{l} (offset', line', length', current', error', cursor') = \\ \quad (`offset, `line, `length, `current, `error, `cursor) \\ \wedge \quad \Big(\forall n.\,(n \leq MaxPos) \supset \\ \qquad screen'@n = glyph\_at(`line, `length, n + `offset)\Big) \end{array} \right\rfloor$$

Inside the block, the requirements are partitioned into two cases. In the first case, where i has already reached the top of the screen, the task of the block has been completed. Accordingly, this case is refined into a skip statement in Window 7 on page 178. In the next instance of Window 7 on page 179, we turn our attention to the second, more difficult, case. We prefix our original requirement with a requirement to decrease the variant while maintaining the invariant. We then show, in Window 9 on page 180, that the original specification may be refined into a recursive call to the label. The result of this refinement is as follows.

$$\sqsupseteq_3 \star \left\lfloor \begin{array}{l} I_p\ \sigma' \wedge (v_p\ \sigma' < v_p\ \sigma') \\ \wedge \quad (offset', line', length', current', error', cursor') = \\ \qquad (`offset, `line, `length, `current, `error, `cursor) \end{array} \right\rfloor ; \\ P(`\sigma, \sigma')$$

To complete the refinement of the recursive block, we must implement the requirement to decrease the invariant while maintaining the variant. We decide to decrease the invariant by one. The requirements can then simplified to show that we must increment the value of i, and update the contents of the screen at position i. The desired contents of the screen depends on whether i corresponds to a position in the line or a position beyond the end of the line. Both cases are implemented with assignments in Window 8 on page 187.

$$\sqsupseteq_3 \star \mathsf{if}\ `i + `offset < `length\ \mathsf{then} \\ \quad (i', screen') := (`i + 1, `screen \dagger (`i, char2glyph\ `line@(`i + `offset))) \\ \mathsf{else}\ (i', screen') := (`i + 1, `screen \dagger (`i, \cdot))$$

This completes the implementation of the recursive block, so we begin to close windows. In Window 5 on page 189, we find that the initialisation of the block remains unimplemented. This requirement is satisfied by assigning i the value zero. We then continue to close the remaining windows.

Once we return to Window 1 on page 190, we see that we have implemented only the case where the screen was initially correctly positioned. Fortunately, since no particular behaviour is required of the other case, the same implementation will do for both. This concludes the refinement of $PAINT$. The result of this refinement is the following theorem:

$$
\begin{array}{l}
\vdash PAINT(\grave{}line, \grave{}length, \grave{}current, line', length', current', \\
\qquad \grave{}error, error', \grave{}offset, screen', cursor', offset') \sqsubseteq_3 \\
\quad cursor' := \grave{}current - \grave{}offset; \\
\quad \mathsf{var}\, i.\, i' := 0; \\
\qquad \mathsf{label}\, P.\, \mathsf{if}\ \grave{}i = MaxPos + 1\ \mathsf{then\ skip} \\
\qquad\quad \mathsf{else\ if}\ \grave{}i + \grave{}offset < \grave{}length\ \mathsf{then} \\
\qquad\qquad (i', screen') := \\
\qquad\qquad\quad (\grave{}i + 1, \grave{}screen \dagger (\grave{}i, char2glyph\ \grave{}line @ (\grave{}i + \grave{}offset))) \\
\qquad\quad \mathsf{else}\ (i', screen') := (\grave{}i + 1, \grave{}screen \dagger (\grave{}i, \cdot)); \\
\qquad\quad P(\grave{}\sigma, \sigma')
\end{array}
$$

6.10 Comments on the Refinement

The conclusions drawn in this section relate not only to the derivation of the line editor as described here, but also to the detailed presentation of the derivation given in Appendix B. It may therefore be helpful to skim through that appendix before reading on.

The line editor is an example of a real-world system. Though neither particularly large nor complex, it is a nontrivial example with features characteristic of large developments. For example, the editor makes use of complex data structures and data abstraction. It is hoped, therefore, that the case study has demonstrated that the specification and refinement techniques proposed in this dissertation are capable of scaling up to handle real software projects.

Note that the use of a three-valued logic was not obtrusive during the refinement. This is because most of the requirements were phrased as two-valued predicates. On those occasions when the specification being considered allowed nontermination, it was invariably phrased in the form $Pre \supset \lfloor Rel \rfloor$. Such specifications are best refined by opening a subwindow on the two-valued clause $\lfloor Rel \rfloor$ and refining it under the assumption that Pre is true. Assuming the result of that process is an implementation Imp, the enclosing term $Pre \supset Imp$ can be simply refined to Imp.

The abstract and concrete data structures of the line editor were chosen during its specification. To enhance the clarity of the specification, most of the requirements were phrased in terms of the abstract data structures. During the refinement, it was necessary to transform these requirements so that they were phrased in terms of the concrete data structures. Such transformations are usually referred to as *data refinements* (or sometimes *data reifications*). In the refinement of the *INIT* operation, this transformation takes place in Window 5 on page 129. Similar transformations are made in the development of the *EDIT* operation in Window 6 on page 148 and in Window 7 on page 150. We make no distinction between data refinement and ordinary, *operational*, refinement. We simply transform predicates using window inference to solve whatever refinement problems we encounter.

The window numbers used throughout the refinement in Appendix B imposed a tree structure on the text of the refinement. This structure mirrors the pattern of the development and can be used to help readers browse the document. For example, by skipping over windows with a greater number than the last one, readers can effectively 'fast-forward' though those parts of a development that they find obvious or in which they are not interested. This structuring technique is particularly useful for large derivations, where it is important for the detail of the development be recorded, but where reading the entire document would prove tedious. Tools could be provided that automate this style of browsing to improve the presentation of large refinements. Such a tool might elide all subwindows from the presentation and provide a commentary linking the visible steps. Readers could then elect to expand the subwindows between those steps they were interested in.

It is a feature of large software systems that their maintenance is frequently a more expensive and lengthy process than the original development. The potential for reducing maintenance costs may yet lead to the application of refinement to projects that are not safety or mission critical. The most obvious potential saving in maintenance arises from the reduction in the number of bugs which need to be fixed after the system is released. However, maintenance also arises out of a need to meet changing requirements. Refinement can help there too. The product of a refinement is not just the code, but a record of the development, including all the design decisions. This record is a valuable piece of documentation which can help maintainers understand the code and how it works.

Of course, to preserve trust in a product and the value of the refinement document, it is the derivation, not just the code, that must be maintained. The structure imposed on derivations though the use of window inference can also prove to be useful in their maintenance. For example, the specification of the *EDIT* operation requires that when a character is deleted from the line, all the characters above the current position must be moved down. In the derivation of *EDIT*, we implemented this requirement with a recursive block that moved each of the characters one at a time. Suppose that in the next version of the *Generic-PC* (the *Generic-PC+* perhaps?) the designers add some block movement hardware to the system. We could then replace

our recursive implementation with a call to the block-movement hardware. It would not be necessary to repeat the entire derivation to establish the correctness of this replacement. We need only redevelop the refinement commenced in Window 8 on page 155 which led to the recursive implementation.

Chapter 7

Conclusions

The preceding chapters contain the development of a method of program refinement. This method is a synthesis of a general style of transformational reasoning and a unified treatment of specifications and programs. The intention has been to develop a method of refinement that is applicable to the development of large programs. With this in mind, the method has been designed to be simple, intuitive, and amenable to tool support.

The traditional, goal oriented, approach to theorem proving is well suited to the construction of correct software by verification. It is, however, less suited to the development of software by refinement. Therefore, an important first step was to create a style of transformational reasoning for use with refinement. The approach taken was to generalise an existing, goal oriented, style of reasoning called 'window inference' [RS93]. The result was a transformational style of reasoning, also called window inference, which can be tailored to support transformations that preserve any given preorder. Window inference allows its users to concentrate on transforming selected subexpressions, supports arbitrarily fine manipulations of a term, and gives users access to information implicit in the context of an expression. These features make window inference well suited to refinement applications.

By describing the operations of window inference in terms of natural deduction, it became possible to build a window-inference interface to the HOL theorem-proving system. This interface is distributed as a library with the HOL system. That the window-inference tool has already found applications beyond its use in this dissertation (see section 2.9) is a testimony to the utility of the technique.

The original intention was to create a method of refinement by combining generalised window-inference with a unified treatment of specifications and programs as predicates. A prototype refinement tool was developed based on this approach and presented at *The 5th Refinement Workshop* [Gru92]. Unfortunately, refinements developed with this tool guaranteed only partial correctness. The desire to overcome this shortcoming motivated a survey of existing models for treating specifications and programs as predicates, which was presented in chapter 3. The models were examined for their suitability for use with program refinement. In particular, the following

aspects of the models were considered: their ability to distinguish between specifications of terminating and nonterminating behaviour, the ease with which they could describe the semantics of a simple imperative programming language, and the simplicity of their refinement relation. The conclusion of the survey was that none of the models examined constituted a good basis for a method of refinement. In addition, by examining several different possible interpretations, the survey illustrated that there is no commonly held understanding of what a predicate means a specification of behaviour.

To overcome the difficulties associated with the various models examined in the survey, a new model based on predicates from a three-valued logic was developed. Three-valued logics have often been used to deal with the complexities introduced by nontermination. The most notable work being Jones's 'Logic of Partial Functions' [BCJ84] and Scott's 'Logic of Computable Functions' [SS71]. The logic proposed here extends only the boolean type with an extra value, and is therefore slightly simpler than these others. By using the third truth-value to represent potential nontermination, the new model is able to distinguish more easily between descriptions of terminating and nonterminating behaviour than those examined in the survey. Predicates in the three-valued logic can also give an elegant semantics to an imperative programming language, as simple as any possible with the two-valued models surveyed. They also have a simple refinement relation. These qualities make the new model well suited for use with program refinement.

The combination of the generalised window-inference style of reasoning with the unified treatment of specifications and programs as three-valued predicates yields a simple and flexible method of refinement. Users of the method are free to pursue a refinement in any direction that can be justified by the semantics of the target language. Window inference allows users to concentrate their attention on selected parts of a refinement while giving them access to contextual information. Experience with the method reveals some frequently repeated sequences of refinements that constitute single conceptual refinement steps. These sequences have been examined so that, where possible, they may be used as a single refinement step.

It is sometimes argued that a refinement approach to software development can be counterproductive in situations where the specification is simple enough that a complete implementation is obvious. In such situations, it may be easier to adopt a verification approach and simply prove the implementation correct. The generalised window inference system could, in principle, be used for verification style proofs by transforming expressions of the form $Imp \sqsupseteq_3 Spec$ to true while preserving implication. Indeed, the two styles of development could be merged, with refinement used to the point where the remainder of the implementation is obvious, at which point verification can be used.

The refinement method includes a technique for developing recursive programs. The difficulties in establishing the correctness of the technique are side-stepped by using an approximate meaning for recursive blocks. When the refinement process is

complete, the approximate meaning can be replaced by the true meaning. This trick is presented with the three-valued predicative semantics, but it could also be applied with equal effectiveness to refinement methods based on other formalisms.

The method of program refinement proposed in this dissertation has been developed with the intention of being used with tool support. As yet, no refinement tool has been built. The tool envisaged would automate the opening and closing of windows, and check the validity of the transformations performed by the user. The tool must also be programmable so that users can automate those refinements they frequently repeat. The construction of the window inference tool [Gru91c] and the prototype refinement tool [Gru92] demonstrate the feasibility of building such a tool.

The goal of the work presented here has been to create a method of constructing correct software that is applicable to the development of large programs. A case-study has been included to illustrate the effectiveness of the method proposed. The example chosen was that of a line editor. While not a particularly large program by the standards of today, the line editor is a nontrivial example with a 'real-world' flavour. It also has some features characteristic of large programs, including the need for complex data structures, and the use of data abstraction. It is hoped that the ease with which the case-study was implemented demonstrates that specification and refinement methods developed here can scale up to tackle large problems.

The use of window inference as the basis of a refinement method imposes a tree-like structure on the record of a development. This structure can be used to browse a development to the desired level of detail, thus easing the task of reading and understanding it. It may also prove useful to the maintenance of the resulting program and its record of development. Such considerations are important as the maintenance of large programs is often a more expensive and lengthy process than their original development.

7.1 Future Work

There is room to further explore the ideas developed in this dissertation. For example, window inference seems particularly well suited for use with a graphic user interface. Laurent Théry has already built a graphic front-end for the window-inference tool [Thé93]. Further experiments in this area may prove useful as part of a general investigation into the use of graphic interfaces to theorem provers.

The refinement method proposed in this dissertation used a programming language with a semantics described using predicates from a three-valued logic. However, it is possible to develop similar styles of refinement based on window inference that use other kinds of semantics. For example, Joakim von Wright has already used the window-inference tool to do program refinement using a more traditional, weakest-precondition, style of semantics [vW94, vWHLL93]. Another approach that may prove rewarding to examine is the timed predicative approach used by Eric

Hehner [Heh90].

The generalised version of window-inference offers a new style of reasoning that may find application beyond the development of imperative programs investigated here. Perhaps the most obvious potential application areas are general theorem proving, and the development and optimisation of functional programs and hardware designs. Mats Larsson has already investigated the use of window inference for the transformational design of hardware [Lar93], while Andrew Gordon has performed some brief, but encouraging, experiments on using window inference to optimise designs expressed in various hardware description languages [Gor93].

Perhaps the most obvious task in need of attention is the implementation of the proposed refinement tool. The implementation would require the following:

- a formalisation of the three-valued logic inside the HOL theorem prover;
- a definition in HOL of the semantics of the target language in terms of predicates from the three-valued logic;
- the implementation of a parser and unparser for the target language. Support for the context dependent nature of the language would involve modifying the window-inference system to store the syntactic context of each window.

The prototype refinement tool described in [Gru92] has already explored some of these issues. The prototype contained a definition of the semantics of the target language in terms of two-valued predicates. It also contained an unparser for the target language, including the necessary modifications to support its context dependant features.

Appendix A

Definitions

Elsewhere in this dissertation, the existence of types of pairs, lists and arrays has been assumed. The existence of several functions on these types has also been assumed, along with theorems about the effect of those functions. Those functions are now described, along with the theorems we desire to hold of them.

A.1 Pairs

We have assumed the following operations on pairs:

$\_,\_ : \alpha \to \beta \to (\alpha \times \beta)$
Pair construction.

$\mathsf{fst} : \alpha \times \beta \to \alpha$
The first component of a pair.

$\mathsf{snd} : \alpha \times \beta \to \beta$
The second component of a pair.

Theorems About Pairs

We have also assumed the following theorems about those functions.

$$\vdash \forall x\ y.\, \mathsf{fst}(x, y) = x \qquad \text{[fst\_THM]}$$

$$\vdash \forall x\ y.\, \mathsf{snd}(x, y) = y \qquad \text{[snd\_THM]}$$

$$\vdash \forall p.\, p = (\mathsf{fst}\ p, \mathsf{snd}\ p) \qquad \text{[fst,snd\_THM]}$$

A.2 Lists

We have assumed the following constants and functions about lists:

$\langle\rangle : \alpha\,\mathsf{list}$
The empty list.

$\_\triangleright\_ : \alpha \to (\alpha\,\mathsf{list}) \to (\alpha\,\mathsf{list})$
The function $\triangleright$ takes an element and a list and returns the list with the element prepended.

$\_\triangleleft\_ : (\alpha\,\mathsf{list}) \to \alpha \to (\alpha\,\mathsf{list})$
The function $\triangleleft$ takes a list and an element and returns the list with the element appended.

$\_\frown\_ : (\alpha\,\mathsf{list}) \to (\alpha\,\mathsf{list}) \to (\alpha\,\mathsf{list})$
The function $\frown$ concatenates two lists.

$\# : (\alpha\,\mathsf{list}) \to NAT$
The function # takes a list and returns the length.

$\_\downarrow\_ : (\alpha\,\mathsf{list}) \to NAT \to \alpha$
Assuming that the list l contains at least $i+1$ elements, the expression $l\downarrow i$ denotes the element at position i (lists are numbered from zero).

$\mathsf{head} : (\alpha\,\mathsf{list}) \to \alpha$
Assuming that the list l is nonempty, $\mathsf{head}\,l$ returns the first element of l.

$\mathsf{tail} : (\alpha\,\mathsf{list}) \to (\alpha\,\mathsf{list})$
Assuming that the list l is nonempty, $\mathsf{tail}\,l$ returns the list containing all but the first element of l.

$\mathsf{last} : (\alpha\,\mathsf{list}) \to \alpha$
Assuming that the list l is nonempty, $\mathsf{last}\,l$ returns the last element of l.

$\mathsf{but\_last} : (\alpha\,\mathsf{list}) \to (\alpha\,\mathsf{list})$
Assuming that the list l is nonempty, $\mathsf{but\_last}\,l$ returns the list containing all but the last element of l.

Theorems About Lists

We have also assumed the following theorems about those functions.

$$\vdash \#\langle\rangle = 0 \qquad [\#\langle\rangle\_\mathsf{THM}]$$

$$\vdash \forall e\ l.\ \#(e \triangleright l) = \#l + 1 \qquad [\#\triangleright\_\mathsf{THM}]$$

$$\vdash \forall l\ e.\ \#(l \triangleleft e) = \#l + 1 \qquad [\#\triangleleft\_\text{THM}]$$

$$\vdash \forall l\ m.\ \#(l \frown m) = \#l + \#m \qquad [\#\frown\_\text{THM}]$$

$$\vdash \forall l.\ (\#l = 0) = (l = \langle\rangle) \qquad [\#0\_\text{THM}]$$

$$\vdash \forall l.\ (\#l \neq 0) \supset (l = (\mathsf{head}\ l) \triangleright (\mathsf{tail}\ l)) \qquad [\neg\#0\triangleright\_\text{THM}]$$

$$\vdash \forall l.\ (\#l \neq 0) \supset (l = (\mathsf{but\_last}\ l) \triangleleft (\mathsf{last}\ l)) \qquad [\neg\#0\triangleleft\_\text{THM}]$$

$$\vdash \forall e\ l.\ (e \triangleright l)\downarrow 0 = e \qquad [\triangleright\downarrow 0\_\text{THM}]$$

$$\vdash \forall e\ l\ i.\ (0 < i \leq \#l) \supset ((e \triangleright l)\downarrow i = l\downarrow(i-1)) \qquad [\triangleright\downarrow i\_\text{THM}]$$

$$\vdash \forall l\ e\ i.\ (i < \#l) \supset ((l \triangleleft e)\downarrow i = l\downarrow i) \qquad [\triangleleft\downarrow i\_\text{THM}]$$

$$\vdash \forall l\ e.\ (l \triangleleft e)\downarrow(\#l) = e \qquad [\triangleleft\downarrow\#\_\text{THM}]$$

$$\vdash \forall l\ m\ i.\ (i < \#l) \supset ((l \frown m)\downarrow i = l\downarrow i) \qquad [\frown\downarrow 1\_\text{THM}]$$

$$\vdash \forall l\ m\ i.\ (\#l \leq i < \#l + \#m) \supset ((l \frown m)\downarrow i = m\downarrow(i - \#l)) \qquad [\frown\downarrow 2\_\text{THM}]$$

A.3 Arrays

We have assumed the following operations on arrays.

$\_@\_ : (\mathsf{array}\ Max\ \mathsf{of}\ \alpha) \to NAT \to \alpha$
: Assuming that i is less than or equal to Max for the array a, the expression $a@i$ denotes the element at position i in the array (arrays are numbered from zero).

$\_\dagger\_ : (\mathsf{array}\ Max\ \mathsf{of}\ \alpha) \to (NAT \times \alpha) \to (\mathsf{array}\ Max\ \mathsf{of}\ \alpha)$
: Assuming that i is less than or equal to Max for the array a, the expression $a\dagger(i, x)$ denotes the array whose elements are equal to the corresponding elements of a, except the ith element which is equal to x.

$\mathsf{max} : (\mathsf{array}\ Max\ \mathsf{of}\ \alpha) \to NAT$
: The expression $\mathsf{max}\ a$ returns the index of the highest numbered position in the array a.

Note that these functions are dependently typed. Dependant typing is not directly available in HOL, but its effect can be simulated [JM93].

Theorems About Arrays

We have also assumed the following theorems about arrays.

$$\vdash \forall a\ i\ e.(i \leq \mathsf{max}\ a) \supset ((a\dagger(i,e))@i = e) \qquad [\dagger@ii\_\mathsf{THM}]$$

$$\vdash \forall a\ i\ j\ e.((i \leq \mathsf{max}\ a) \wedge (j \leq \mathsf{max}\ a) \wedge (i \neq j)) \supset (a\dagger(i,e))@j = a@j \qquad [\dagger@ij\_\mathsf{THM}]$$

Appendix B

Refinement of the Line Editor

B.1 Introduction

This appendix presents the derivation of an implementation for the line editor described in chapter 6. The derivation may be easier to follow if read in conjunction with the overview presented in the final section of that chapter.

The development is divided into three sections. The first contains the refinement of the initialisation operation, *INIT*. The second describes the refinement of the *EDIT* operation. During the development of *INIT*, a requirement to update the contents of the screen is identified. As it seems likely that this requirement will reoccur in the development of *EDIT*, a predicate, *PAINT*, is defined that captures the requirement. *PAINT* is subsequently used in the development of both *INIT* and *EDIT*. The final section of the development contains the refinement of an implementation for *PAINT*.

Because of the large size of the derivation, it may prove particularly helpful to browse the derivation using the method described in section 5.10. Essentially, it is recommended that the reader skips over some of the subwindows when reading the development. For example, if you are currently reading a window of depth n and the next window has depth $n+1$, then skip forward to the next window of depth n. If you find that the step taken between these two windows was simple enough that you have confidence in its correctness, then continue reading. Otherwise, return to where you were and read the intervening subwindows.

B.2 The Refinement of *INIT*

In this section we implement the initialisation operation of the editor. We begin by making *INIT* the focus of a window which can then be transformed while preserving refinement. Note that the state space σ contains at least the following program variables: *current*, *cursor*, *error*, *length*, *line*, *offset* and *screen*.

Window: 1

$$\sqsupseteq_3 \star INIT(line', length', current', error', screen', cursor', offset')$$

The first step of the refinement is to use $INIT\_\mathsf{DEF}$ to replace $INIT$ by its definition.

$$\sqsupseteq_3 \star \underline{\exists} l_2 . \boxed{INIT3(l_2, line', length', current', error', screen', cursor', offset')}$$

The initialisation was defined in terms of $INIT3$ and an abstract representation of the line l_2. We concentrate our attention on finding an implementation for $INIT3$ that does not involve the abstraction l_2 by opening a subwindow inside the scope of l_2.

Window: 2

$$\sqsupseteq_3 \star INIT3(l_2, line', length', current', error', screen', cursor', offset')$$

We replace $INIT3$ with its definition using the theorem $INIT3\_\mathsf{DEF}$.

$$\begin{array}{ll} \sqsupseteq_3 \star & INIT2(l_2, line', length', current', error') \\ \quad \underline{\triangle} & \lfloor GoodScreen(screen', cursor', offset', line', length', current') \rfloor \end{array}$$

The focus requires an implementation that initialises the line and the error state and ensures that the screen correctly depicts the the initial state of the line. We sequentially decompose this specification into one which allows us to first initialise the line and error state, and then initialise the screen.

$$\sqsupseteq_3 \star \boxed{INIT2(l_2, line', length', current', error')}; \left\lfloor \begin{array}{ll} & (line', length', current', error') = (`line, `length, `current, `error) \\ \wedge & GoodScreen(screen', cursor', offset', `line, `length, `current) \end{array} \right\rfloor$$

First we concentrate on refining the initialisation of the line and error state by opening a window on $INIT2$. Later, we return and implement the initialisation of the screen.

Window: 3

$$\begin{array}{l} \vdots \\ \sqsupseteq_3 \star INIT2(l_2, line', length', current', error') \end{array}$$

Using the theorem $INIT2\_\mathsf{DEF}$ we expand the definition of $INIT2$.

$$\sqsupseteq_3 \star INIT1(l_2, line', length', current') \underline{\triangle} \lfloor \neg error' \rfloor$$

The predicate in the focus describes a requirement to initialise the line and error state. We decompose this into a requirement to first initialise the line, and then initialise the error state.

$$\begin{array}{l}\sqsupseteq_3 \star\, INIT\,1(l_2, line', length', current');\\ \quad (\lfloor (line', length', current') = (`line, `length, `current)\rfloor \mathbin{\underline{\wedge}} \lfloor \neg error' \rfloor)\end{array}$$

The second clause of the composition must set $error'$ to false and leave the other variables unchanged. This requirement is trivially refined by an assignment statement.

$$\sqsupseteq_3 \star \boxed{INIT\,1(l_2, line', length', current')}; error' := \mathsf{F}$$

We now turn our attention to the first clause of the composition and open a subwindow there.

Window: 4

$$\begin{array}{l}\vdots\\ \sqsupseteq_3 \star\, INIT\,1(l_2, line', length', current')\end{array}$$

INIT 1 is replaced by its definition according to *INIT 1*_DEF.

$$\sqsupseteq_3 \star \lfloor (l_2 = empty\_line) \wedge IsAbs(l_2, line', length', current') \rfloor$$

Using *empty_line*_DEF we replace *empty_line* by its definition.

$$\sqsupseteq_3 \star \lfloor (l_2 = (\langle\,\rangle, \langle\,\rangle)) \wedge \boxed{IsAbs(l_2, line', length', current')} \rfloor$$

We now proceed by opening a subwindow to examine more closely the predicate *IsAbs*.

Window: 5

$$\begin{array}{l}\vdots\\ !\ l_2 = (\langle\,\rangle, \langle\,\rangle)\\ \supset \star\, IsAbs(l_2, line', length', current')\end{array}$$

Using the assumption, we can replace l_2 by $(\langle\,\rangle, \langle\,\rangle)$.

$$\supset \star\, IsAbs((\langle\,\rangle, \langle\,\rangle), line', length', current')$$

We can now replace *IsAbs* with its definition according to *IsAbs*_DEF.

$$\begin{array}{ll} \supset\star & length' = \#\langle\,\rangle + \#\langle\,\rangle \\ \wedge & current' = \#\langle\,\rangle \\ \wedge & length' \leq MaxLength \\ \wedge & \Big(\forall i.\,(i < length') \supset (line'@i = (\langle\,\rangle^\frown\langle\,\rangle)\downarrow i)\Big) \end{array}$$

From the theorem $\#\langle\,\rangle$_THM (given in Appendix A) we know that the length of an empty list is zero. This allows us to simplify the focus as follows.

$$\supset\star\, length' = 0 \wedge current' = 0 \wedge \boxed{\begin{array}{ll} & length' \leq MaxLength \\ \wedge & \Big(\forall i.\,(i < length') \supset (line'@i = (\langle\,\rangle^\frown\langle\,\rangle)\downarrow i)\Big) \end{array}}$$

We open a subwindow to further simplify the focus.

Window: 6

$$\begin{array}{l} \vdots \\ !\; length' = 0 \\ !\; current' = 0 \\ \supset\star\,(length' \leq MaxLength) \wedge \Big(\forall i.\,(i < length') \supset (line'@i = (\langle\,\rangle^\frown\langle\,\rangle)\downarrow i)\Big) \end{array}$$

Using the assumption $length' = 0$, the focus may be simplified to true.

$$\supset\star\,\mathsf{T}$$

We now close the window and return to its parent.

Window: 5

$$\begin{array}{l} \vdots \\ \supset\star\,(length' = 0) \wedge (current' = 0) \wedge \mathsf{T} \\ \supset\star\,(length' = 0) \wedge (current' = 0) \end{array}$$

We simplify and close this window.

Window: 4

$$\begin{array}{l} \vdots \\ \sqsupseteq_3\star\,\lfloor (l_2 = (\langle\,\rangle, \langle\,\rangle)) \wedge (length' = 0) \wedge (current' = 0)\rfloor \end{array}$$

Let us partition off the subclause relating to the abstract variable into a three-valued predicate of its own.

$$\sqsupseteq_3\star\,\lfloor l_2 = (\langle\,\rangle, \langle\,\rangle)\rfloor \mathbin{\underline{\wedge}} \lfloor (length' = 0) \wedge (current' = 0)\rfloor$$

The second clause of the three-valued conjunction is required to set $length'$ and $current'$ to zero. This clause is trivially refined by an assignment statement.

$$\sqsupseteq_3 \star \lfloor l_2 = (\langle\,\rangle, \langle\,\rangle) \rfloor \mathbin{\underline{\triangle}} ((length', current') := (0,0))$$

We have now found an implementation for this window that does not relate any of the concrete variables to the abstract variable l_2. We close this window and return to its parent.

Window: 3

$$\begin{array}{l} \vdots \\ \sqsupseteq_3 \star (\lfloor l_2 = (\langle\,\rangle, \langle\,\rangle) \rfloor \mathbin{\underline{\triangle}} ((length', current') := (0,0))); \\ \quad error' := \mathsf{F} \end{array}$$

Similarly this window is now completely implemented so it too can be closed.

Window: 2

$$\begin{array}{l} \sqsupseteq_3 \star (\lfloor l_2 = (\langle\,\rangle, \langle\,\rangle) \rfloor \mathbin{\underline{\triangle}} ((length', current') := (0,0))); \\ \quad error' := \mathsf{F}; \\ \quad \left\lfloor \begin{array}{ll} & (line', length', current', error') = (\grave{}line, \grave{}length, \grave{}current, \grave{}error) \\ \wedge & GoodScreen(screen', cursor', offset', \grave{}line, \grave{}length, \grave{}current) \end{array} \right\rfloor \end{array}$$

At this point we could proceed to implement the initialisation of the screen. Instead we close the window and return to the top level where we be able to remove all reference to the abstract variable l_2.

Window: 1

$$\begin{array}{l} \sqsupseteq_3 \star \underline{\exists} l_2 . (\lfloor l_2 = (\langle\,\rangle, \langle\,\rangle) \rfloor \mathbin{\underline{\triangle}} ((length', current') := (0,0))); \\ \quad error' := \mathsf{F}; \\ \quad \left\lfloor \begin{array}{ll} & (line', length', current', error') = \\ & \quad (\grave{}line, \grave{}length, \grave{}current, \grave{}error) \\ \wedge & GoodScreen(screen', cursor', offset', \grave{}line, \grave{}length, \grave{}current) \end{array} \right\rfloor \end{array}$$

We can simplify the focus by moving the scope of the existential quantifier in until it contains just the clause $\underline{\exists} l_2 . \lfloor l_2 = (\langle\,\rangle, \langle\,\rangle) \rfloor$.

$$\begin{array}{l} \sqsupseteq_3 \star (\underline{\exists} l_2 . \lfloor l_2 = (\langle\,\rangle, \langle\,\rangle) \rfloor) \mathbin{\underline{\triangle}} (length', current') := (0,0); \\ \quad error' := \mathsf{F}; \\ \quad \left\lfloor \begin{array}{ll} & (line', length', current', error') = (\grave{}line, \grave{}length, \grave{}current, \grave{}error) \\ \wedge & GoodScreen(screen', cursor', offset', \grave{}line, \grave{}length, \grave{}current) \end{array} \right\rfloor \end{array}$$

This clause is obviously true, and so can be simplified away.

$$\begin{array}{l} \sqsupseteq_3 \star\, (\mathit{length}', \mathit{current}') := (0,0); \\ \quad \mathit{error}' := \mathsf{F}; \\ \boxed{\left\lfloor \begin{array}{ll} & (\mathit{line}', \mathit{length}', \mathit{current}', \mathit{error}') = (\grave{}\mathit{line}, \grave{}\mathit{length}, \grave{}\mathit{current}, \grave{}\mathit{error}) \\ \wedge & \mathit{GoodScreen}(\mathit{screen}', \mathit{cursor}', \mathit{offset}', \grave{}\mathit{line}, \grave{}\mathit{length}, \grave{}\mathit{current}) \end{array} \right\rfloor} \end{array}$$

We now open a window on that portion of the focus that remains unimplemented—the initialisation of the screen.

Window: 2

$$\begin{array}{l} ! \,((\tilde{}\mathit{length}, \tilde{}\mathit{current}) := (0,0)) \neq \mathsf{F}_3 \\ ! \,(\grave{}\mathit{error} := \mathsf{F}) \neq \mathsf{F}_3 \\ \sqsupseteq_3 \star \left\lfloor \begin{array}{ll} & (\mathit{line}', \mathit{length}', \mathit{current}', \mathit{error}') = (\grave{}\mathit{line}, \grave{}\mathit{length}, \grave{}\mathit{current}, \grave{}\mathit{error}) \\ \wedge & \mathit{GoodScreen}(\mathit{screen}', \mathit{cursor}', \mathit{offset}', \grave{}\mathit{line}, \grave{}\mathit{length}, \grave{}\mathit{current}) \end{array} \right\rfloor \end{array}$$

Using the theorem *GoodScreen*_DEF we expand the definition of *GoodScreen*.

$$\sqsupseteq_3 \star \left\lfloor \begin{array}{ll} & (\mathit{line}', \mathit{length}', \mathit{current}', \mathit{error}') = (\grave{}\mathit{line}, \grave{}\mathit{length}, \grave{}\mathit{current}, \grave{}\mathit{error}) \\ \wedge & \mathit{IsView}(\mathit{screen}', \grave{}\mathit{line}, \grave{}\mathit{length}, \mathit{offset}') \\ \wedge & \mathit{SeeCurrent}(\mathit{offset}', \grave{}\mathit{current}) \\ \wedge & \mathit{CursorCurrent}(\mathit{cursor}', \mathit{offset}', \grave{}\mathit{current}) \end{array} \right\rfloor$$

The predicate *SeeCurrent* requires us to position the screen so that the current position can be seen. The other predicates require us to make sure that the screen correctly depicts the line at that position (without changing the contents of the line or the error state). We implement this by first positioning the screen and then updating its contents.

$$\begin{array}{l} \sqsupseteq_3 \star \left\lfloor \begin{array}{ll} & \mathit{SeeCurrent}(\mathit{offset}', \grave{}\mathit{current}) \\ \wedge & (\mathit{line}', \mathit{length}', \mathit{current}', \mathit{error}') = (\grave{}\mathit{line}, \grave{}\mathit{length}, \grave{}\mathit{current}, \grave{}\mathit{error}) \end{array} \right\rfloor ; \\ \boxed{\left\lfloor \begin{array}{ll} & \mathit{offset}' = \grave{}\mathit{offset} \\ \wedge & (\mathit{line}', \mathit{length}', \mathit{current}', \mathit{error}') = (\grave{}\mathit{line}, \grave{}\mathit{length}, \grave{}\mathit{current}, \grave{}\mathit{error}) \\ \wedge & \mathit{IsView}(\mathit{screen}', \grave{}\mathit{line}, \grave{}\mathit{length}, \grave{}\mathit{offset}) \\ \wedge & \mathit{CursorCurrent}(\mathit{cursor}', \grave{}\mathit{offset}, \grave{}\mathit{current}) \end{array} \right\rfloor} \end{array}$$

Let us first concentrate on the second clause of the composition, which describes how the screen should be updated, once it has been correctly positioned.

Window: 3

$$
\begin{array}{l}
\vdots \\
! \; SeeCurrent(\grave{}offset, \tilde{}current) \\
! \; (\ldots, \grave{}current, \ldots) = (\ldots, \tilde{}current, \ldots) \\
\sqsupseteq_3 \star \left\lfloor \begin{array}{ll} & offset' = \grave{}offset \\ \wedge & (line', length', current', error') = (\grave{}line, \grave{}length, \grave{}current, \grave{}error) \\ \wedge & IsView(screen', \grave{}line, \grave{}length, \grave{}offset) \\ \wedge & CursorCurrent(cursor', \grave{}offset, \grave{}current) \end{array} \right\rfloor
\end{array}
$$

The focus of the current window requires us to ensure that the screen correctly depicts the contents of the line where the screen is positioned. We call this *painting* the screen.

We will also be required to paint the screen during the *EDIT* operation. To avoid duplicating the development of code to paint the screen we define a predicate *PAINT* that captures the requirements of painting the screen. Whenever the screen requires paining we refine the requirement to the predicate *PAINT*. Later, in section B.4, we will refine *PAINT* into an implementation.

$$
\begin{array}{l}
\vdash^{\text{def}} PAINT(\grave{}line, \grave{}length, \grave{}current, line', length', current', \\
\qquad \grave{}error, error', \grave{}offset, screen', cursor', offset') = \\
\left(\begin{array}{l} SeeCurrent(\grave{}offset, \grave{}cursor) \supset \\ \quad \left\lfloor \begin{array}{ll} & (offset', line', length', current', error') = \\ & \quad (\grave{}offset, \grave{}line, \grave{}length, \grave{}current, \grave{}error) \\ \wedge & IsView(screen', \grave{}line, \grave{}length, \grave{}offset) \\ \wedge & CursorCurrent(cursor', \grave{}offset, \grave{}current) \end{array} \right\rfloor \end{array} \right) \qquad [PAINT\_\mathsf{DEF}]
\end{array}
$$

It follows directly from $PAINT\_\mathsf{DEF}$ and the assumptions that we can refine the focus into an instance of *PAINT*.

$$
\begin{array}{l}
\vdots \\
\sqsupseteq_3 \star PAINT(\grave{}line, \grave{}length, \grave{}current, line', length', current', \\
\qquad \grave{}error, error', \grave{}offset, screen', cursor', offset')
\end{array}
$$

Having resolved to find an implementation for *PAINT* later, we close this window.

Window: 2

$$
\begin{array}{l}
! \; ((\tilde{}length, \tilde{}current) := (0,0)) \neq \mathsf{F}_3 \\
! \; (\grave{}error := \mathsf{F}) \neq \mathsf{F}_3 \\
\sqsupseteq_3 \star \boxed{\left\lfloor \begin{array}{ll} & SeeCurrent(offset', \grave{}current) \\ \wedge & (line', length', current', error') = (\grave{}line, \grave{}length, \grave{}current, \grave{}error) \end{array} \right\rfloor} ; \\
\quad PAINT(\grave{}line, \grave{}length, \grave{}current, line', length', current', \\
\qquad \grave{}error, error', \grave{}offset, screen', cursor', offset')
\end{array}
$$

We proceed by opening a window on that portion of the focus that remains unimplemented.

Window: 3

$$\begin{array}{l} \vdots \\ !\ ((\tilde{}\mathit{length}, \tilde{}\mathit{current}) := (0,0)) \neq \mathsf{F}_3 \\ !\ (\grave{}\mathit{error} := \mathsf{F}) \neq \mathsf{F}_3 \\ \sqsupseteq_3 \star \left\lfloor \begin{array}{ll} & \mathit{SeeCurrent}(\mathit{offset}', \grave{}\mathit{current}) \\ \wedge & (\mathit{line}', \mathit{length}', \mathit{current}', \mathit{error}') = (\grave{}\mathit{line}, \grave{}\mathit{length}, \grave{}\mathit{current}, \grave{}\mathit{error}) \end{array} \right\rfloor \end{array}$$

From the assumptions

$$\begin{array}{l} (\grave{}\mathit{error} := \mathsf{F}) \neq \mathsf{F}_3 \\ ((\tilde{}\mathit{length}, \tilde{}\mathit{current}) := (0,0)) \neq \mathsf{F}_3 \end{array}$$

and the definition of assignment, we know that $\grave{}\mathit{current} = \tilde{}\mathit{current}$ and $\tilde{}\mathit{current}$ is zero. Therefore we can replace $\grave{}\mathit{current}$ by zero in the focus as follows.

$$\sqsupseteq_3 \star \left\lfloor \begin{array}{ll} & \mathit{SeeCurrent}(\mathit{offset}', 0) \\ \wedge & (\mathit{line}', \mathit{length}', \mathit{current}', \mathit{error}') = (\grave{}\mathit{line}, \grave{}\mathit{length}, \grave{}\mathit{current}, \grave{}\mathit{error}) \end{array} \right\rfloor$$

Using the theorem $\mathit{SeeCurrent}\_\mathsf{DEF}$ we replace $\mathit{SeeCurrent}$ by its definition.

$$\sqsupseteq_3 \star \left\lfloor \begin{array}{ll} & 0 \leq 0 - \mathit{offset}' \leq \mathit{MaxPos} \\ \wedge & (\mathit{line}', \mathit{length}', \mathit{current}', \mathit{error}') = (\grave{}\mathit{line}, \grave{}\mathit{length}, \grave{}\mathit{current}, \grave{}\mathit{error}) \end{array} \right\rfloor$$

It is clear that this predicate is satisfied by assigning offset' to zero.

$$\sqsupseteq_3 \star \mathit{offset}' := 0$$

We now close the window.

Window: 2

$$\begin{array}{l} \sqsupseteq_3 \star \mathit{offset}' := 0; \\ \quad \mathit{PAINT}(\grave{}\mathit{line}, \grave{}\mathit{length}, \grave{}\mathit{current}, \mathit{line}', \mathit{length}', \mathit{current}', \\ \qquad \grave{}\mathit{error}, \mathit{error}', \grave{}\mathit{offset}, \mathit{screen}', \mathit{cursor}', \mathit{offset}') \end{array}$$

The focus of this window is now completely implemented, so we close it and return to the top level.

Window: 1

$$\begin{array}{l} \sqsupseteq_3 \star (\mathit{length}', \mathit{current}') := (0,0); \mathit{error}' := \mathsf{F}; \mathit{offset}' := 0; \\ \quad \mathit{PAINT}(\grave{}\mathit{line}, \grave{}\mathit{length}, \grave{}\mathit{current}, \mathit{line}', \mathit{length}', \mathit{current}', \\ \qquad \grave{}\mathit{error}, \mathit{error}', \grave{}\mathit{offset}, \mathit{screen}', \mathit{cursor}', \mathit{offset}') \end{array}$$

Aside from the details of the predicate PAINT, the implementation of the initialisation operation is now complete. (The implementation of PAINT can be found on page 191.)

B.3 The Refinement of *EDIT*

In this section we refine the edit operation of the line editor defined in chapter 6. We commence the development by making *EDIT* the focus of a window that preserves refinement. Note that the state σ contains at least the following program variables: *current*, *cursor*, *error*, *key*, *length*, *line*, *offset* and *screen*.

Window: 1

$$\sqsupseteq_3 \star EDIT(\grave{}key, \grave{}line, \grave{}length, \grave{}current, line', length', current', error', \grave{}screen, \grave{}cursor, \grave{}offset, screen', cursor', offset')$$

In order to see what to do next, we must expand the definition of *EDIT*.

$$\sqsupseteq_3 \star (\exists l_1 . IsAbs(l_1, \grave{}line, \grave{}length, \grave{}current)) \supset \left(\begin{array}{l} \mathsf{let}\ l_1 = (\varepsilon l_1 . IsAbs(l_1, \grave{}line, \grave{}length, \grave{}current))\ \mathsf{in} \\ \quad \exists l_2 . \boxed{\begin{array}{l} EDIT3(\grave{}key, l_1, l_2, \grave{}line, \grave{}length, \grave{}current, \\ \qquad line', length', current', error', \\ \qquad \grave{}screen, \grave{}cursor, \grave{}offset, \\ \qquad screen', cursor', offset') \end{array}} \end{array} \right)$$

The specification requires that if the initial values of the concrete variables *line*, *length* and *current* are such that they represent a line, then the line should be edited in accordance with the user's commands. The initial values of *line*, *length* and *current* should always represent a line, so no particular behaviour is required of the other case. We continue to work by opening a window on the core of the specification—the predicate *EDIT3*, which describes how the line should be edited.

Window: 2

$$\begin{array}{l} ! \ \exists l_1 . IsAbs(l_1, \grave{}line, \grave{}length, \grave{}current) \\ ! \ l_1 = (\varepsilon l_1 . IsAbs(l_1, \grave{}line, \grave{}length, \grave{}current)) \\ \sqsupseteq_3 \star EDIT3(\grave{}key, l_1, l_2, \grave{}line, \grave{}length, \grave{}current, line', length', current', \\ \qquad error', \grave{}screen, \grave{}cursor, \grave{}offset, screen', cursor', offset' \ ) \end{array}$$

In this context our assumptions allow us to deduce that l_1 is an abstraction of the initial values of *line*, *length*, and *current*.

$$\begin{array}{l} ¡ \ IsAbs(l_1, \grave{}line, \grave{}length, \grave{}current) \\ \sqsupseteq_3 \star EDIT3(\grave{}key, l_1, l_2, \grave{}line, \grave{}length, \grave{}current, line', length', current', \\ \qquad error', \grave{}screen, \grave{}cursor, \grave{}offset, screen', cursor', offset' \ ) \end{array}$$

Let us expand the definition of *EDIT 3*.

$$\exists_3 \star \mathit{GoodScreen}(\grave{}\mathit{screen}, \grave{}\mathit{cursor}, \grave{}\mathit{offset}, \grave{}\mathit{line}, \grave{}\mathit{length}, \grave{}\mathit{current}) \supset$$

$$\begin{array}{|ll|}\hline & \mathit{EDIT\,2}(\grave{}\mathit{key}, l_1, l_2, \grave{}\mathit{line}, \grave{}\mathit{length}, \grave{}\mathit{current}, \\ & \qquad \mathit{line}', \mathit{length}', \mathit{current}', \mathit{error}' \quad) \\ \underline{\triangle} & \lfloor \mathit{GoodScreen}(\mathit{screen}', \mathit{cursor}', \mathit{offset}', \mathit{line}', \mathit{length}', \mathit{current}') \rfloor \\ \underline{\triangle} & \lfloor \mathit{MinMove}(\grave{}\mathit{offset}, \mathit{offset}', \mathit{current}') \rfloor \\ \hline \end{array}$$

The specification is conditional on the screen correctly reflecting the initial contents of the line. This represents the normal operation of the *Generic-PC*, and no particular behaviour is required if this is not the case. We therefore concentrate our attention on the behaviour required when the contents of the screen are as we expect.

Window: 3

$$\begin{array}{ll} & \vdots \\ & ¡\ \mathit{IsAbs}(l_1, \grave{}\mathit{line}, \grave{}\mathit{length}, \grave{}\mathit{current}) \\ & !\ \mathit{GoodScreen}(\grave{}\mathit{screen}, \grave{}\mathit{cursor}, \grave{}\mathit{offset}, \grave{}\mathit{line}, \grave{}\mathit{length}, \grave{}\mathit{current}) \\ \exists_3 \star & \mathit{EDIT\,2}(\grave{}\mathit{key}, l_1, l_2, \grave{}\mathit{line}, \grave{}\mathit{length}, \grave{}\mathit{current}, \\ & \qquad \mathit{line}', \mathit{length}', \mathit{current}', \mathit{error}' \quad) \\ \underline{\triangle} & \lfloor \mathit{GoodScreen}(\mathit{screen}', \mathit{cursor}', \mathit{offset}', \mathit{line}', \mathit{length}', \mathit{current}') \rfloor \\ \underline{\triangle} & \lfloor \mathit{MinMove}(\grave{}\mathit{offset}, \mathit{offset}', \mathit{current}') \rfloor \end{array}$$

We expand the definition of *GoodScreen* in the focus..

$$\begin{array}{ll} \exists_3 \star & \mathit{EDIT\,2}(\grave{}\mathit{key}, l_1, l_2, \grave{}\mathit{line}, \grave{}\mathit{length}, \grave{}\mathit{current}, \\ & \qquad \mathit{line}', \mathit{length}', \mathit{current}', \mathit{error}' \quad) \\ \underline{\triangle} & \left\lfloor \begin{array}{ll} & \mathit{IsView}(\mathit{screen}', \mathit{line}', \mathit{length}', \mathit{offset}') \\ \wedge & \mathit{SeeCurrent}(\mathit{offset}', \mathit{current}') \\ \wedge & \mathit{CursorCurrent}(\mathit{cusor}', \mathit{offset}', \mathit{current}') \end{array} \right\rfloor \\ \underline{\triangle} & \lfloor \mathit{MinMove}(\grave{}\mathit{offset}, \mathit{offset}', \mathit{current}') \rfloor \end{array}$$

The focus requires an implementation that edits the line in accordance with the user's instructions (as described by *EDIT 2*), positions the screen (with the minimum amount of movement) so the the current position can be seen, and ensures that the screen reflects the contents of the line at that position. We use sequential compositions to decompose these requirements so that we may first edit the line, then position

the screen, and then ensure that the screen reflects the contents of the line.

$$\sqsupseteq_3 \star \left(\begin{array}{ll} & EDIT\,2(\grave{}key, l_1, l_2, \grave{}line, \grave{}length, \grave{}current, \\ & \qquad\qquad line', length', current', error') \\ \triangle & \lfloor \grave{}offset = offset' \rfloor \end{array}\right);$$
$$\left|\begin{array}{ll} & (line', length', current', error') = \\ & \quad (\grave{}line, \grave{}length, \grave{}current, \grave{}error) \\ \wedge & MinMove(\grave{}offset, offset', \grave{}current) \\ \wedge & SeeCurrent(offset', \grave{}current) \end{array}\right|;$$
$$\boxed{\left|\begin{array}{ll} & (offset', line', length', current', error') = \\ & \quad (\grave{}offset, \grave{}line, \grave{}length, \grave{}current, \grave{}error) \\ \wedge & IsView(screen', \grave{}line, \grave{}length, \grave{}offset) \\ \wedge & CursorCurrent(cursor', \grave{}offset, \grave{}current) \end{array}\right|}$$

We open a window on the last clause of the sequential composition.

Window: 4

$$\begin{array}{l} \vdots \\ \mathbin{¡} IsAbs(l_1, \acute{}line, \acute{}length, \acute{}offset) \\ \mathbin{!} GoodScreen(\acute{}screen, \acute{}cursor, \acute{}offset, \acute{}line, \acute{}length, \acute{}current) \\ \mathbin{!} EDIT\,2(\check{}key, l_1, l_2, \acute{}line, \acute{}length, \acute{}current, \\ \qquad\qquad \check{}line, \check{}length, \check{}current, \check{}error) \neq \mathsf{F}_3 \\ \mathbin{!} (\grave{}line, \grave{}length, \grave{}current, \grave{}error) = (\check{}line, \check{}length, \check{}current, \check{}error) \\ \mathbin{!} MinMove(\check{}offset, \grave{}offset, \check{}current) \\ \mathbin{!} SeeCurrent(\grave{}offset, \check{}current) \end{array}$$
$$\sqsupseteq_3 \star \left|\begin{array}{ll} & (offset', line', length', current', error') = \\ & \quad (\grave{}offset, \grave{}line, \grave{}length, \grave{}current, \grave{}error) \\ \wedge & IsView(screen', \grave{}line, \grave{}length, \grave{}offset) \\ \wedge & CursorCurrent(cursor', \grave{}offset, \grave{}current) \end{array}\right|$$

These requirements are trivially implemented by the predicate *PAINT* (the implementation of *PAINT* is described in section B.4). *PAINT* is a naive implementation of these requirements that redraws the entire screen. By exploiting the assumption that the screen initially depicted the line and knowledge of how the line was edited, we could have developed a more elaborate implementation that redrew only that part of the screen effected by the edit operation. However, for reasons of brevity we have chosen to re-use the more simple implementation.

$$\sqsupseteq_3 \star PAINT(\grave{}line, \grave{}length, \grave{}current, line', length', current', \\ \grave{}error, error', \grave{}offset, screen', cursor', offset')$$

Having completed the implementation of these requirements, we close the window.

Window: 3

$$
\begin{array}{l}
\vdots \\
\mathsf{i}\ IsAbs(l_1, \grave{}line, \grave{}length, \grave{}current) \\
\sqsupseteq_3 \star \left(\begin{array}{ll} & EDIT2(\grave{}key, l_1, l_2, \grave{}line, \grave{}length, \grave{}current, \\ & \qquad line', length', current', error') \\ \underline{\wedge} & \lfloor \grave{}offset = offset' \rfloor \end{array} \right); \\
\boxed{\left| \begin{array}{ll} & (line', length', current', error') = \\ & \quad (\grave{}line, \grave{}length, \grave{}current, \grave{}error) \\ \wedge & MinMove(\grave{}offset, offset', \grave{}current) \\ \wedge & SeeCurrent(offset', \grave{}current) \end{array} \right|}; \\
PAINT(\grave{}line, \grave{}length, \grave{}current, line', length', current', \\
\qquad \grave{}error, error', \grave{}offset, screen', cursor', offset')
\end{array}
$$

We now concentrate our attention on the second clause of the sequential composition.

Window: 4

$$
\begin{array}{l}
\vdots \\
\sqsupseteq_3 \star \left| \begin{array}{ll} & (line', length', current', error') = \\ & \quad (\grave{}line, \grave{}length, \grave{}current, \grave{}error) \\ \wedge & MinMove(\grave{}offset, offset', \grave{}current) \\ \wedge & SeeCurrent(offset', \grave{}current) \end{array} \right|
\end{array}
$$

The focus of the current window requires that the screen be positioned, with the minimum amount of movement, so that the current position can be seen. Of course, if the current position can already be seen, then the screen will not have to be moved at all. We introduce a conditional so that we can solve this case on its own.

$$
\begin{array}{l}
\sqsupseteq_3 \star \mathsf{if}\ SeeCurrent(\grave{}offset, \grave{}current)\ \mathsf{then} \\
\qquad \boxed{\left| \begin{array}{ll} & (line', length', current', error') = \\ & \quad (\grave{}line, \grave{}length, \grave{}current, \grave{}error) \\ \wedge & MinMove(\grave{}offset, offset', \grave{}current) \\ \wedge & SeeCurrent(offset', \grave{}current) \end{array} \right|} \\
\quad \mathsf{else} \\
\qquad \left| \begin{array}{ll} & (line', length', current', error') = \\ & \quad (\grave{}line, \grave{}length, \grave{}current, \grave{}error) \\ \wedge & MinMove(\grave{}offset, offset', \grave{}current) \\ \wedge & SeeCurrent(offset', \grave{}current) \end{array} \right|
\end{array}
$$

We now open a window on the first branch of the conditional, and consider the case where the current position can already be seen.

Window: 5

$$\vdots$$
$$! \; SeeCurrent(\grave{}offset, \grave{}current)$$
$$\sqsupseteq_3 \star \left[\begin{array}{ll} & (line', length', current', error') = \\ & \quad (\grave{}line, \grave{}length, \grave{}current, \grave{}error) \\ \wedge & MinMove(\grave{}offset, offset', \grave{}current) \\ \wedge & SeeCurrent(offset', \grave{}current) \end{array}\right]$$

We expand the definition of *MinMove* to see exactly what it requires.

$$\sqsupseteq_3 \star \left[\begin{array}{ll} & (line', length', current', error') = \\ & \quad (\grave{}line, \grave{}length, \grave{}current, \grave{}error) \\ \wedge & \left(\begin{array}{l} \forall o.\, SeeCurrent(o, \grave{}current) \supset \\ \qquad |\grave{}offset - offset'| \leq |\grave{}offset - o| \end{array}\right) \\ \wedge & SeeCurrent(offset', \grave{}current) \end{array}\right]$$

We can now see that maintaining the value of all variables would make this focus true. Therefore, we can refine the current focus to skip.

$$\sqsupseteq_3 \star \mathsf{skip}$$

Having completed the implementation of this case, we close the window and return to the parent.

Window: 4

$$\vdots$$
$$\sqsupseteq_3 \star \mathsf{if}\ SeeCurrent(\grave{}offset, \grave{}current)\ \mathsf{then\ skip}$$
$$\mathsf{else}$$
$$\boxed{\left[\begin{array}{ll} & (line', length', current', error') = \\ & \quad (\grave{}line, \grave{}length, \grave{}current, \grave{}error) \\ \wedge & MinMove(\grave{}offset, offset', \grave{}current) \\ \wedge & SeeCurrent(offset', \grave{}current) \end{array}\right]}$$

We open a subwindow to concentrate on the case in which the current position is not already in view.

Window: 5

$$\vdots$$
$$! \; \boxed{\neg SeeCurrent(\grave{}offset, \grave{}current)}$$
$$\sqsupseteq_3 \star \left[\begin{array}{ll} & (line', length', current', error') = \\ & \quad (\grave{}line, \grave{}length, \grave{}current, \grave{}error) \\ \wedge & MinMove(\grave{}offset, offset', \grave{}current) \\ \wedge & SeeCurrent(offset', \grave{}current) \end{array}\right]$$

In this context we may assume that $\neg SeeCurrent(\grave{}offset, \grave{}current)$. We open a window on this assumption to see what useful facts may be derived from it.

Window: 6

$$\vdots$$
$$\supset \star \neg SeeCurrent(\grave{}offset, \grave{}current)$$

First, we expand the definition of $SeeCurrent$.

$$\supset \star \neg(0 \leq \grave{}current - \grave{}offset \leq MaxPos)$$

Next, we perform a series of algebraic simplifications on the focus.

$$\supset \star \neg((0 \leq \grave{}current - \grave{}offset) \wedge (\grave{}current - \grave{}offset \leq MaxPos))$$
$$\supset \star \neg(0 \leq \grave{}current - \grave{}offset) \vee \neg(\grave{}current - \grave{}offset \leq MaxPos)$$
$$\supset \star \neg(\grave{}offset \leq \grave{}current) \vee \neg(\grave{}current \leq MaxPos + \grave{}offset)$$
$$\supset \star (\grave{}current < \grave{}offset) \vee (MaxPos + \grave{}offset < \grave{}current)$$

From the initial assumption that the current position is not in view, we have deduced that the current position must either lie to the left of the screen or to the right of the screen.

Window: 5

$$\vdots$$
$$\mathsf{i}\ (\grave{}current < \grave{}offset) \vee (MaxPos + \grave{}offset < \grave{}current)$$
$$\exists\!\!\exists_3 \star \left| \begin{array}{ll} & (line', length', current', error') = \\ & \quad (\grave{}line, \grave{}length, \grave{}current, \grave{}error) \\ \wedge & MinMove(\grave{}offset, offset', \grave{}current) \\ \wedge & SeeCurrent(offset', \grave{}current) \end{array} \right|$$

We now know that the current position lies either to the left or to the right of the screen. We introduce a conditional to consider these two cases separately.

$$\exists\!\!\exists_3 \star \mathsf{if}\ \grave{}current < \grave{}offset\ \mathsf{then}$$
$$\boxed{\left| \begin{array}{ll} & (line', length', current', error') = \\ & \quad (\grave{}line, \grave{}length, \grave{}current, \grave{}error) \\ \wedge & MinMove(\grave{}offset, offset', \grave{}current) \\ \wedge & SeeCurrent(offset', \grave{}current) \end{array} \right|}$$
$$\mathsf{else}$$
$$\left| \begin{array}{ll} & (line', length', current', error') = \\ & \quad (\grave{}line, \grave{}length, \grave{}current, \grave{}error) \\ \wedge & MinMove(\grave{}offset, offset', \grave{}current) \\ \wedge & SeeCurrent(offset', \grave{}current) \end{array} \right|$$

We open a subwindow on the first case, where the current position lies to the left of the screen.

Window: 6

$$
\begin{array}{l}
\vdots \\
! \ `current < `offset \\
\sqsupseteq_3 \star \left\lfloor \begin{array}{ll} & (line', length', current', error') = \\ & \quad (`line, `length, `current, `error) \\ \wedge & MinMove(`offset, offset', `current) \\ \wedge & SeeCurrent(offset', `current) \end{array} \right\rfloor
\end{array}
$$

Since the current position lies to the left of the screen, we can bring the current position back into view by moving the screen as far left as the current position. We therefore decide to implement the current focus with the assignment $offset' := `current$.

$$
\begin{array}{ll}
\sqsupseteq_3 \star & (offset' := `current) \\
\underline{\triangle} & \left\lfloor \boxed{\begin{array}{ll} & (line', length', current', error') = \\ & \quad (`line, `length, `current, `error) \\ \wedge & MinMove(`offset, offset', `current) \\ \wedge & SeeCurrent(offset', `current) \end{array}} \right\rfloor
\end{array}
$$

We open a subwindow to simplify away the remaining clauses now that we have decided how to implement them.

Window: 7

$$
\begin{array}{l}
\vdots \\
! \ `current < `offset \\
! \ (offset' := `current) \neq \mathsf{F}_3 \\
\supset \star \begin{array}{ll} & (`line, `length, `current, `error) = \\ & \quad (`line, `length, `current, `error) \\ \wedge & MinMove(`offset, offset', `current) \\ \wedge & SeeCurrent(offset', `current) \end{array}
\end{array}
$$

Using the assumption $(offset' := `current) \neq \mathsf{F}_3$, we rewrite selected instances of $offset'$ and $current'$ to $`current$.

$$
\begin{array}{ll}
\supset \star & (`line, `length, `current, `error) = \\ & \quad (`line, `length, `current, `error) \\ \wedge & MinMove(`offset, `current, `current) \\ \wedge & SeeCurrent(`current, `current)
\end{array}
$$

Next, we expand with the definitions of *MinMove* and *SeeCurrent*.

$$
\begin{array}{ll}
\supset \star & (`line, `length, `current, `error) = \\ & \quad (`line, `length, `current, `error) \\ \wedge & \left(\begin{array}{l} \forall o. (0 \leq `current - o \leq MaxPos) \supset \\ \qquad |`offset - `current| \leq |`offset - o| \end{array} \right) \\ \wedge & 0 \leq `current - `current \leq MaxPos
\end{array}
$$

Then, we make some obvious simplifications.

$$\supset \star \forall o.\, (0 \leq \grave{}current - o \leq MaxPos) \supset$$
$$\boxed{|\grave{}offset - \grave{}current| \leq |\grave{}offset - o|}$$

We open a subwindow to simplify the expression $|\grave{}offset - \grave{}current| \leq |\grave{}offset - o|$.

Window: 8

$$\vdots$$
$$! \ \grave{}current < \grave{}offset$$
$$! \ 0 \leq \grave{}current - o \leq MaxPos$$
$$\supset \star |\grave{}offset - \grave{}current| \leq |\grave{}offset - o|$$

Since we know $\grave{}current$ is less than $\grave{}offset$, we may remove the absolute value function from $|\grave{}offset - \grave{}current|$. We may also remove the absolute value function from $|\grave{}offset - o|$ since we know that $\grave{}current - o$ is not negative and $\grave{}current$ is less than $\grave{}offset$.

$$\supset \star \grave{}offset - \grave{}current \leq \grave{}offset - o$$

Next, we make some algebraic simplifications to the focus.

$$\supset \star -\grave{}current \leq -o$$
$$\supset \star o \leq \grave{}current$$

From the assumption $0 \leq \grave{}current - o$, we can deduce that the focus is true.

$$\supset \star \mathsf{T}$$

We close the window.

Window: 7

$$\vdots$$
$$\supset \star \forall o.\, (0 \leq \grave{}current - o \leq MaxPos) \supset \mathsf{T}$$

We can now simplify the focus to true.

$$\supset \star \mathsf{T}$$

Having simplified the focus, we close the window.

Window: 6

$$\begin{array}{l} \vdots \\ \sqsupseteq_3 \star (\mathit{offset}' := \grave{}\mathit{current}) \mathbin{\triangle} \lfloor \mathsf{T} \rfloor \end{array}$$

We have now shown that the assignment takes care of all the other requirements. The focus can now be simplified as follows.

$$\begin{array}{l} \vdots \\ \sqsupseteq_3 \star \mathit{offset}' := \grave{}\mathit{current} \end{array}$$

We have now finished implementing the case where the current position lies to the left of the screen. We therefore close the window and return to the parent.

Window: 5

$$\begin{array}{l} \vdots \\ \quad \mathsf{i}\ (\grave{}\mathit{current} < \grave{}\mathit{offset}) \vee (\mathit{MaxPos} + \grave{}\mathit{offset} < \grave{}\mathit{current}) \\ \sqsupseteq_3 \star \mathsf{if}\ \grave{}\mathit{current} < \grave{}\mathit{offset}\ \mathsf{then}\ \mathit{offset}' := \grave{}\mathit{current} \\ \quad \mathsf{else} \\ \qquad \left\lfloor \begin{array}{ll} & (\mathit{line}', \mathit{length}', \mathit{current}', \mathit{error}') = \\ & \quad (\grave{}\mathit{line}, \grave{}\mathit{length}, \grave{}\mathit{current}, \grave{}\mathit{error}) \\ \wedge & \mathit{MinMove}(\grave{}\mathit{offset}, \mathit{offset}', \grave{}\mathit{current}) \\ \wedge & \mathit{SeeCurrent}(\mathit{offset}', \grave{}\mathit{current}) \end{array} \right\rfloor \end{array}$$

The refinement of the case where the current position lies to the right of the screen is essentially the same as the previous case. We therefore do not present the details of refinement of the else clause here. Instead, we skip straight to the result of that refinement.

$$\begin{array}{l} \sqsupseteq_3 \star \mathsf{if}\ \grave{}\mathit{current} < \grave{}\mathit{offset}\ \mathsf{then}\ \mathit{offset}' := \grave{}\mathit{current} \\ \quad \mathsf{else}\ \mathit{offset}' := \grave{}\mathit{current} - \mathit{MaxPos} \end{array}$$

We have now completed the refinement of this window. We therefore close the window and return to its parent.

Window: 4

$$\begin{array}{l} \vdots \\ \sqsupseteq_3 \star \mathsf{if}\ \mathit{SeeCurrent}(\grave{}\mathit{offset}, \grave{}\mathit{current})\ \mathsf{then\ skip} \\ \quad \mathsf{else\ if}\ \grave{}\mathit{current} < \grave{}\mathit{offset}\ \mathsf{then}\ \mathit{offset}' := \grave{}\mathit{current} \\ \quad \mathsf{else}\ \mathit{offset}' := \grave{}\mathit{current} - \mathit{MaxPos} \end{array}$$

We have almost finished the implementation of this window too. Only the test of the first conditional, $SeeCurrent(\grave{}offset, \grave{}current)$, remains unimplemented. We therefore expand the definition of $SeeCurrent$ in this test.

$$\begin{array}{l}\sqsupseteq_3 \star \mathsf{if}\ 0 \leq \grave{}current - \grave{}offset \leq MaxPos\ \mathsf{then\ skip}\\ \quad \mathsf{else\ if}\ \grave{}current < \grave{}offset\ \mathsf{then}\ offset' := \grave{}current\\ \quad \mathsf{else}\ offset' := \grave{}current - MaxPos\end{array}$$

Assuming that the term $0 \leq \grave{}current - \grave{}offset \leq MaxPos$ is executable, this completes the implementation of the current window, so it too can be closed.

Window: 3

$$\begin{array}{l}\vdots\\ \mathsf{i}\ IsAbs(l_1, \grave{}line, \grave{}length, \grave{}current)\\ \sqsupseteq_3 \star \boxed{\begin{pmatrix} EDIT\,2(\grave{}key, l_1, l_2, \grave{}line, \grave{}length, \grave{}current, \\ \qquad\qquad line', length', current', error') \\ \underline{\triangle}\ \ \lfloor offset' = \grave{}offset \rfloor \end{pmatrix}};\\ \quad \mathsf{if}\ 0 \leq \grave{}current - \grave{}offset \leq MaxPos\ \mathsf{then\ skip}\\ \quad \mathsf{else\ if}\ \grave{}current < \grave{}offset\ \mathsf{then}\ offset' := \grave{}current\\ \quad \mathsf{else}\ offset' := \grave{}current - MaxPos;\\ \quad PAINT(\grave{}line, \grave{}length, \grave{}current, line', length', current',\\ \qquad\qquad \grave{}error, error', \grave{}offset, screen', cursor', offset')\end{array}$$

In this window, the first clause of the sequential composition remains unimplemented. This clause requires us to modify the line being edited in accordance with the user's instructions.

We open a subwindow on the first clause of the sequential composition, to implement the core of the editor.

Window: 4

$$\begin{array}{l}\vdots\\ \mathsf{i}\ IsAbs(l_1, \grave{}line, \grave{}length, \grave{}current)\\ \sqsupseteq_3 \star \quad EDIT\,2(\grave{}key, l_1, l_2, \grave{}line, \grave{}length, \grave{}current,\\ \qquad\qquad\qquad line', length', current', error')\\ \quad \underline{\triangle}\ \ \lfloor offset' = \grave{}offset \rfloor\end{array}$$

So that we can see what is required, we expand the definition of $EDIT\,2$.

$$\begin{array}{l}\sqsupseteq_3 \star \quad \begin{pmatrix} Error(\grave{}key, l_1, l_2, \grave{}line, \grave{}length, \grave{}current) \\ \quad \Rightarrow\ \ \lfloor error' \wedge (l_2 = l_1) \wedge IsAbs(l_2, line', length', current') \rfloor \\ \quad \mid\ \ EDIT\,1(\grave{}key, l_1, l_2, line', length', current',) \,\underline{\triangle}\, \lfloor \neg error' \rfloor \end{pmatrix}\\ \quad \underline{\triangle}\ \ \lfloor offset' = \grave{}offset \rfloor\end{array}$$

Next, we expand definitions of *Error* and *EDIT 1*.

$$
\begin{array}{l}
\sqsupseteq_3 \star \left(\begin{array}{l}
\left(\begin{array}{ll}
 & (\grave{}key = \boxed{\mathsf{delete}}) \wedge (\grave{}current = 0) \\
\vee & (is\_char\_key\ \grave{}key) \wedge (\grave{}length = MaxLength) \\
\vee & (\grave{}key = \boxed{\leftarrow}) \wedge (\grave{}current = 0) \\
\vee & (\grave{}key = \boxed{\rightarrow}) \wedge (\grave{}current = \grave{}length)
\end{array} \right) \\
\quad \Rightarrow \lfloor error' \wedge (l_2 = l_1) \wedge IsAbs(l_2, line', length', current') \rfloor \\
\quad \mathbf{I} \left(\begin{array}{l|l}
 & (\grave{}key = \boxed{\mathsf{delete}}) \supset (l_2 = del\ l_1) \\
\wedge & (is\_char\_key\ \grave{}key) \supset \\
 & \quad l_2 = ins\ (key2char\ \grave{}key)\ l_1 \\
\wedge & (\grave{}key = \boxed{\leftarrow}) \supset (l_2 = left\ l_1) \\
\wedge & (\grave{}key = \boxed{\rightarrow}) \supset (l_2 = right\ l_1) \\
\wedge & IsAbs(l_2, line', length', current')
\end{array} \right) \\
\quad \underline{\wedge} \lfloor \neg error' \rfloor
\end{array} \right) \\
\quad \underline{\wedge} \lfloor offset' = \grave{}offset \rfloor
\end{array}
$$

The behaviour of the editor depends on whether the key hit is the delete key, a character key, or the left or right movement keys. We introduce conditionals to consider each of these cases. For the sake of brevity, we demonstrate the refinement of only one case. The implementation of insertion is of similar complexity to the implementation of deletion, both of which are more complex than a simple movement. We therefore choose to refine the delete operation as it is representative of the complexity of all the operations. We elide the other cases from the presentation.

$\sqsupseteq_3 \star$ if $\grave{}key = \boxed{\mathsf{delete}}$ then

$$
\begin{array}{l}
\left(\begin{array}{l}
\left(\begin{array}{ll}
 & (\grave{}key = \boxed{\mathsf{delete}}) \wedge (\grave{}current = 0) \\
\vee & (is\_char\_key\ \grave{}key) \wedge (\grave{}length = MaxLength) \\
\vee & (\grave{}key = \boxed{\leftarrow}) \wedge (\grave{}current = 0) \\
\vee & (\grave{}key = \boxed{\rightarrow}) \wedge (\grave{}current = \grave{}length)
\end{array} \right) \\
\quad \Rightarrow \lfloor error' \wedge (l_2 = l_1) \wedge IsAbs(l_2, line', length', current') \rfloor \\
\quad \mathbf{I} \left(\begin{array}{l|l}
 & (\grave{}key = \boxed{\mathsf{delete}}) \supset (l_2 = del\ l_1) \\
\wedge & \left(\begin{array}{l} is\_char\_key\ \grave{}key \supset \\ \quad l_2 = ins\ (key2char\ \grave{}key)\ l_1 \end{array} \right) \\
\wedge & (\grave{}key = \boxed{\leftarrow}) \supset (l_2 = left\ l_1) \\
\wedge & (\grave{}key = \boxed{\rightarrow}) \supset (l_2 = right\ l_1) \\
\wedge & IsAbs(l_2, line', length', current')
\end{array} \right) \\
\quad \underline{\wedge} \lfloor \neg error' \rfloor
\end{array} \right) \\
\underline{\wedge} \lfloor offset' = \grave{}offset \rfloor
\end{array}
$$

else . . .

We open a subwindow on the case where the delete key has been pressed to proceed with its implementation.

Window: 5

$$
\begin{array}{l}
\vdots \\
\mathbin{¡}\ IsAbs(l_1, \grave{}line, \grave{}length, \grave{}current) \\
!\ \grave{}key = \boxed{\mathsf{delete}} \\
\quad \sqsupseteq_3 \star \left(\begin{array}{l}
\left(\begin{array}{ll}
 & (\grave{}key = \boxed{\mathsf{delete}}) \wedge (\grave{}current = 0) \\
\vee & (is\_char\_key\ \grave{}key) \wedge (\grave{}length = MaxLength) \\
\vee & (\grave{}key = \boxed{\leftarrow}) \wedge (\grave{}current = 0) \\
\vee & (\grave{}key = \boxed{\rightarrow}) \wedge (\grave{}current = \grave{}length)
\end{array} \right) \\
\quad \Rightarrow \ \lfloor error' \wedge (l_2 = l_1) \wedge IsAbs(l_2, line', length', current') \rfloor \\
\quad \mathbin{!} \left(\begin{array}{l}
\left| \begin{array}{ll}
 & (\grave{}key = \boxed{\mathsf{delete}}) \supset (l_2 = del\ l_1) \\
\wedge & \left(\begin{array}{l} is\_char\_key\ \grave{}key \supset \\ \quad l_2 = ins\ (key2char\ \grave{}key)\ l_1 \end{array} \right) \\
\wedge & (\grave{}key = \boxed{\leftarrow}) \supset (l_2 = left\ l_1) \\
\wedge & (\grave{}key = \boxed{\rightarrow}) \supset (l_2 = right\ l_1) \\
\wedge & IsAbs(l_2, line', length', current')
\end{array} \right| \\
\underline{\triangle}\ \lfloor \neg error' \rfloor
\end{array} \right)
\end{array} \right) \\
\quad \underline{\triangle}\ \lfloor offset' = \grave{}offset \rfloor
\end{array}
$$

Using the assumption that the delete key was pressed, we may simplify away those clauses of the specification that relate to how the editor should behave when the other keys are pressed. The resulting focus contains only those terms that describe the behaviour of the delete operation.

$$
\begin{array}{l}
\sqsupseteq_3 \star \left(\begin{array}{l}
\grave{}current = 0 \\
\quad \Rightarrow \ \lfloor error' \wedge (l_2 = l_1) \wedge IsAbs(l_2, line', length', current') \rfloor \\
\quad \mathbin{!} \left(\begin{array}{ll}
 & \lfloor (l_2 = del\ l_1) \wedge IsAbs(l_2, line', length', current') \rfloor \\
\underline{\triangle} & \lfloor \neg error' \rfloor
\end{array} \right)
\end{array} \right) \\
\quad \underline{\triangle}\ \lfloor offset' = \grave{}offset \rfloor
\end{array}
$$

It is now easy to see that if the current position is at the beginning of the line, then the user has made an error (because there is no character to the left of the current position to delete). However, if there is a character to the left of the current position, then we must delete that character from the line. We introduce a conditional to

consider these two cases separately.

$$\sqsupseteq_3 \star \textsf{ if } \grave{}current = 0 \textsf{ then}$$
$$\boxed{\begin{array}{l} \left(\begin{array}{ll} \grave{}current = 0 & \\ \quad \Rightarrow & \lfloor error' \wedge (l_2 = l_1) \wedge IsAbs(l_2, line', length', current') \rfloor \\ \quad \mid & \left(\begin{array}{ll} & \lfloor (l_2 = del\ l_1) \wedge IsAbs(l_2, line', length', current') \rfloor \\ \underline{\triangle} & \lfloor \neg error' \rfloor \end{array}\right) \end{array}\right) \\ \underline{\triangle} \quad \lfloor offset' = \grave{}offset \rfloor \end{array}}$$
$$\textsf{else}$$
$$\left(\begin{array}{ll} \grave{}current = 0 & \\ \quad \Rightarrow & \lfloor error' \wedge (l_2 = l_1) \wedge IsAbs(l_2, line', length', current') \rfloor \\ \quad \mid & \left(\begin{array}{ll} & \lfloor (l_2 = del\ l_1) \wedge IsAbs(l_2, line', length', current') \rfloor \\ \underline{\triangle} & \lfloor \neg error' \rfloor \end{array}\right) \end{array}\right)$$
$$\underline{\triangle} \quad \lfloor offset' = \grave{}offset \rfloor$$

We open a subwindow first on the simple case, in which the user has made an error.

Window: 6

$$\begin{array}{l} \vdots \\ ¡\ IsAbs(l_1, \grave{}line, \grave{}length, \grave{}current) \\ !\ \grave{}current = 0 \end{array}$$
$$\sqsupseteq_3 \star \quad \left(\begin{array}{ll} \grave{}current = 0 & \\ \quad \Rightarrow & \lfloor error' \wedge (l_2 = l_1) \wedge IsAbs(l_2, line', length', current') \rfloor \\ \quad \mid & \left(\begin{array}{ll} & \lfloor (l_2 = del\ l_1) \wedge IsAbs(l_2, line', length', current') \rfloor \\ \underline{\triangle} & \lfloor \neg error' \rfloor \end{array}\right) \end{array}\right)$$
$$\underline{\triangle} \quad \lfloor offset' = \grave{}offset \rfloor$$

Using the assumption that $\grave{}current = 0$ we can eliminate the other case from the specification.

$$\sqsupseteq_3 \star \quad \lfloor error' \wedge (l_2 = l_1) \wedge IsAbs(\boxed{l_2}, line', length', current') \rfloor$$
$$\underline{\triangle} \quad \lfloor offset' = \grave{}offset \rfloor$$

In the context of the l_2 in the predicate *IsAbs*, we know that l_2 is equal to l_1. We open a subwindow to make this substitution.

Window: 7

$$\begin{array}{l} \vdots \\ !\ l_2 = l_1 \end{array}$$
$$= \star\ l_2$$

Rewrite with $l_2 = l_1$.

$$= \star\, l_1$$

Having made the substitution, we close the window.

Window: 6

$$\begin{array}{l} \vdots \\ \mathsf{i}\ \mathit{IsAbs}(l_1, \grave{}\mathit{line}, \grave{}\mathit{length}, \grave{}\mathit{current}) \\ \sqsupseteq_3 \star \quad \lfloor \mathit{error}' \wedge (l_2 = l_1) \wedge \mathit{IsAbs}(l_1, \mathit{line}', \mathit{length}', \mathit{current}') \rfloor \\ \quad \underline{\triangle}\ \ \lfloor \mathit{offset}' = \grave{}\mathit{offset} \rfloor \end{array}$$

Aside from the expression $l_2 = l_1$, these requirements may now be implemented with an assignment. We rearrange the requirements to make this possible.

$$\sqsupseteq_3 \star \lfloor l_2 = l_1 \rfloor \,\underline{\triangle}\, \lfloor \mathit{error}' \wedge \mathit{IsAbs}(l_1, \mathit{line}', \mathit{length}', \mathit{current}') \wedge \mathit{offset}' = \grave{}\mathit{offset} \rfloor$$

Next, we introduce the assignment, which allows us to simplify away the requirements other than $l_2 = l_1$.

$$\sqsupseteq_3 \star \lfloor l_2 = l_1 \rfloor \,\underline{\triangle}\, \mathit{error}' := \mathsf{T}$$

We now close this window and return to its parent.

Window: 5

$$\begin{array}{l} \sqsupseteq_3 \star \mathsf{if}\ \grave{}\mathit{current} = 0\ \mathsf{then} \\ \qquad \lfloor l_2 = l_1 \rfloor \,\underline{\triangle}\, \mathit{error}' := \mathsf{T} \\ \quad \mathsf{else} \\ \qquad \boxed{\begin{array}{l} \left(\begin{array}{l} \grave{}\mathit{current} = 0 \\ \quad \Rightarrow \ \ \lfloor \mathit{error}' \wedge (l_2 = l_1) \wedge \mathit{IsAbs}(l_2, \mathit{line}', \mathit{length}', \mathit{current}') \rfloor \\ \quad \mathsf{I} \ \ \left(\begin{array}{l} \lfloor (l_2 = \mathit{del}\ l_1) \wedge \mathit{IsAbs}(l_2, \mathit{line}', \mathit{length}', \mathit{current}') \rfloor \\ \underline{\triangle}\ \ \lfloor \neg \mathit{error}' \rfloor \end{array} \right) \end{array} \right) \\ \underline{\triangle}\ \ \lfloor \mathit{offset}' = \grave{}\mathit{offset} \rfloor \end{array}} \end{array}$$

We now turn our attention to the other case, where the user has not made an error, and where we are required to implement a deletion.

Window: 6

$$
\begin{array}{l}
\vdots \\
\mathbin{¡}\ IsAbs(l_1, \grave{}line, \grave{}length, \grave{}current) \\
!\ \grave{}current \neq 0 \\
\sqsupseteq_3 \star \quad \left(\begin{array}{l} \grave{}current = 0 \\ \quad \Rightarrow \ \lfloor error' \wedge (l_2 = l_1) \wedge IsAbs(l_2, line', length', current') \rfloor \\ \quad \mathsf{I} \ \left(\begin{array}{l} \lfloor (l_2 = del\ l_1) \wedge IsAbs(l_2, line', length', current') \rfloor \\ \underline{\wedge} \ \lfloor \neg error' \rfloor \end{array} \right) \end{array} \right) \\
\qquad \underline{\wedge} \ \lfloor offset' = \grave{}offset \rfloor
\end{array}
$$

We can simplify this case under the assumption that $\grave{}current$ is not zero.

$$
\begin{array}{ll}
\sqsupseteq_3 \star & \lfloor (l_2 = del\ l_1) \wedge IsAbs(l_2, line', length', current') \rfloor \\
\quad \underline{\wedge} & \lfloor \neg error' \rfloor \\
\quad \underline{\wedge} & \lfloor offset' = \grave{}offset \rfloor
\end{array}
$$

The focus of this window requires us to ensure that the final state of the concrete variables is such that they represent the initial abstract line with one character deleted. We must also set the error indicator to false. Let us decompose this operation into two steps. First we make the deletion from the line; then we set the error flag to false.

$$
\begin{array}{l}
\sqsupseteq_3 \star \lfloor (l_2 = del\ l_1) \wedge IsAbs(l_2, line', length', current') \wedge (offset' = \grave{}offset) \rfloor; \\
\quad \left\lfloor \begin{array}{ll} & (line', length', current', offset') = (\grave{}line, \grave{}length, \grave{}current, \grave{}offset) \\ \wedge & \neg error' \end{array} \right\rfloor
\end{array}
$$

We can implement the second clause of the sequential composition with an assignment.

$$
\begin{array}{l}
\sqsupseteq_3 \star \lfloor (l_2 = del\ l_1) \wedge IsAbs(\boxed{l_2}, line', length', current') \wedge (offset' = \grave{}offset) \rfloor; \\
\quad error' := \mathsf{F}
\end{array}
$$

In the context of the instance of l_2 in the predicate $IsAbs$, we may assume that l_2 is equal to $del\ l_1$. We open a window at this position to make the substitution.

Window: 7

$$
\begin{array}{l}
\vdots \\
!\ l_2 = del\ l_1 \\
= \star\ l_2
\end{array}
$$

We make the substitution.

$$= \star\ del\ l_1$$

Having made the substitution, we close the window.

Window: 6

$$
\begin{array}{l}
\vdots \\
\mathsf{i}\ \mathit{IsAbs}(l_1, \grave{}\mathit{line}, \grave{}\mathit{length}, \grave{}\mathit{current}) \\
!\ \grave{}\mathit{current} \neq 0 \\
\sqsupseteq_3 \star \lfloor (l_2 = \mathit{del}\ l_1) \wedge \mathit{IsAbs}(\mathit{del}\ l_1, \mathit{line}', \mathit{length}', \mathit{current}') \wedge (\mathit{offset}' = \grave{}\mathit{offset}) \rfloor; \\
\quad \mathit{error}' := \mathsf{F}
\end{array}
$$

Next, we rearrange the focus to separate out the interrelationship between the abstract states.

$$
\sqsupseteq_3 \star \left(\begin{array}{ll} & \lfloor (l_2 = \mathit{del}\ l_1) \rfloor \\ \underline{\wedge} & \boxed{\lfloor \mathit{IsAbs}(\mathit{del}\ l_1, \mathit{line}', \mathit{length}', \mathit{current}') \wedge (\mathit{offset}' = \grave{}\mathit{offset}) \rfloor} \end{array} \right); \\
\mathit{error}' := \mathsf{F}
$$

We open a window to concentrate on implementing the remaining constraints on the concrete variables.

Window: 7

$$
\begin{array}{l}
\vdots \\
\mathsf{i}\ \mathit{IsAbs}(l_1, \grave{}\mathit{line}, \grave{}\mathit{length}, \grave{}\mathit{current}) \\
!\ \neg(\grave{}\mathit{current} = 0) \\
\sqsupseteq_3 \star \lfloor \mathit{IsAbs}(\mathit{del}\ l_1, \mathit{line}', \mathit{length}', \mathit{current}') \wedge (\mathit{offset}' = \grave{}\mathit{offset}) \rfloor
\end{array}
$$

First, we replace l_1 with $(\mathsf{fst}\ l_1, \mathsf{snd}\ l_1)$ using fst,snd_THM (see Appendix A) in both the context and the focus.

$$
\begin{array}{l}
\mathsf{i}\ \mathit{IsAbs}((\mathsf{fst}\ l_1, \mathsf{snd}\ l_1), \grave{}\mathit{line}, \grave{}\mathit{length}, \grave{}\mathit{current}) \\
\sqsupseteq_3 \star \lfloor \mathit{IsAbs}(\mathit{del}(\mathsf{fst}\ l_1, \mathsf{snd}\ l_1), \mathit{line}', \mathit{length}', \mathit{current}') \wedge (\mathit{offset}' = \grave{}\mathit{offset}) \rfloor
\end{array}
$$

Then, we expand the instance of *IsAbs* in the context and add the resulting conjuncts as lemmas.

$$
\begin{array}{l}
\mathsf{i}\ \grave{}\mathit{length} = \#(\mathsf{fst}\ l_1) + \#(\mathsf{snd}\ l_1) \\
\mathsf{i}\ \grave{}\mathit{current} = \#(\mathsf{fst}\ l_1) \\
\mathsf{i}\ \grave{}\mathit{length} \leq \mathit{MaxLength} \\
\mathsf{i}\ \forall i.\, (i < \grave{}\mathit{length}) \supset (\grave{}\mathit{line} @ i = ((\mathsf{fst}\ l_1) \frown (\mathsf{snd}\ l_1)) \downarrow i) \\
\sqsupseteq_3 \star \lfloor \mathit{IsAbs}(\mathit{del}(\mathsf{fst}\ l_1, \mathsf{snd}\ l_1), \mathit{line}', \mathit{length}', \mathit{current}') \wedge (\mathit{offset}' = \grave{}\mathit{offset}) \rfloor
\end{array}
$$

From the assumption that $\grave{}\mathit{current}$ is not zero and the lemma $\grave{}\mathit{current} = \#(\mathsf{fst}\ l_1)$, we derive the following theorem.

$$
\ldots \vdash \#(\mathsf{fst}\ l_1) \neq 0
$$

From this fact and the theorem $\neg\#0\triangleleft\_\mathsf{THM}$ (see Appendix A) we can derive the following theorem:

$$\ldots \vdash (\mathsf{fst}\ l_1) = (\mathsf{but\_last}(\mathsf{fst}\ l_1))\triangleleft(\mathsf{last}(\mathsf{fst}\ l_1))$$

We use this theorem to replace $\mathsf{fst}\ l_1$ thought the context and the focus.

$$\begin{array}{l}
\mathsf{i}\ `\mathit{length} = \#((\mathsf{but_last}(\mathsf{fst}\ l_1))\triangleleft(\mathsf{last}(\mathsf{fst}\ l_1))) + \#(\mathsf{snd}\ l_1) \\
\mathsf{i}\ `\mathit{current} = \#((\mathsf{but_last}(\mathsf{fst}\ l_1))\triangleleft(\mathsf{last}(\mathsf{fst}\ l_1))) \\
\mathsf{i}\ \forall i.\,(i < `\mathit{length}) \supset (`\mathit{line}@i = (((\mathsf{but_last}(\mathsf{fst}\ l_1))\triangleleft(\mathsf{last}(\mathsf{fst}\ l_1)))\frown(\mathsf{snd}\ l_1))\downarrow i) \\
\sqsupseteq_3 \star \left\lfloor \begin{array}{ll} & \mathit{IsAbs}(\mathit{del}((\mathsf{but_last}(\mathsf{fst}\ l_1))\triangleleft(\mathsf{last}(\mathsf{fst}\ l_1))), \mathsf{snd}\ l_1), \\ & \qquad \mathit{line}', \mathit{length}', \mathit{current}' \qquad\qquad) \\ \wedge & \mathit{offset}' = `\mathit{offset} \end{array} \right\rfloor
\end{array}$$

Using the definition of *del*, we simplify the focus as follows.

$$\sqsupseteq_3 \star \left\lfloor \begin{array}{ll} & \mathit{IsAbs}((\ \mathsf{but\_last}(\mathsf{fst}\ l_1), \mathsf{snd}\ l_1), \mathit{line}', \mathit{length}', \mathit{current}') \\ \wedge & \mathit{offset}' = `\mathit{offset} \end{array} \right\rfloor$$

Now we can expand the definition of *IsAbs* in the focus.

$$\sqsupseteq_3 \star \left\lfloor \begin{array}{ll} & \mathit{length}' = \#(\ \mathsf{but\_last}(\mathsf{fst}\ l_1)) + \#(\mathsf{snd}\ l_1) \\ \wedge & \mathit{current}' = \#(\ \mathsf{but\_last}(\mathsf{fst}\ l_1)) \\ \wedge & \mathit{length}' \leq \mathit{MaxLength} \\ \wedge & \boxed{\Big(\forall i.\,(i < \mathit{length}') \supset ((\mathit{line}'@i = ((\ \mathsf{but\_last}(\mathsf{fst}\ l_1))\frown(\mathsf{snd}\ l_1))\downarrow i))\Big)} \\ \wedge & \mathit{offset}' = `\mathit{offset} \end{array} \right\rfloor$$

We open a subwindow to establish how the new contents of the line relate to the previous contents.

Window: 8

$$\begin{array}{l}
\vdots \\
\mathsf{i}\ `\mathit{length} = \#((\mathsf{but_last}(\mathsf{fst}\ l_1))\triangleleft(\mathsf{last}(\mathsf{fst}\ l_1))) + \#(\mathsf{snd}\ l_1) \\
\mathsf{i}\ `\mathit{current} = \#((\mathsf{but_last}(\mathsf{fst}\ l_1))\triangleleft(\mathsf{last}(\mathsf{fst}\ l_1))) \\
\mathsf{i}\ \forall i.\,(i < `\mathit{length}) \supset (`\mathit{line}@i = (((\mathsf{but_last}(\mathsf{fst}\ l_1))\triangleleft(\mathsf{last}(\mathsf{fst}\ l_1)))\frown(\mathsf{snd}\ l_1))\downarrow i) \\
!\ \mathit{length}' = \#(\ \mathsf{but_last}(\mathsf{fst}\ l_1)) + \#(\mathsf{snd}\ l_1) \\
!\ \mathit{current}' = \#(\ \mathsf{but_last}(\mathsf{fst}\ l_1)) \\
\supset \star\, \forall i.\,(i < \mathit{length}') \supset ((\mathit{line}'@i = ((\ \mathsf{but_last}(\mathsf{fst}\ l_1))\frown(\mathsf{snd}\ l_1))\downarrow i))
\end{array}$$

The delete operation should leave the contents of the line before the current position unchanged, while moving the contents of the line after the current position down by one. From the context of this window we can deduce that $\mathit{current}' \leq \mathit{length}'$.

This means that we can partition the universal quantification in the focus into two separate cases, the first describes the contents of the line before the current position, the second the contents of the line after the current position.

$$\supset\star \quad \Big(\forall i.(i < current') \supset \boxed{((line'@i = ((\,\mathsf{but\_last}(\mathsf{fst}\ l_1))\frown(\mathsf{snd}\ l_1))\downarrow i))}\Big)$$
$$\wedge \quad \begin{pmatrix}\forall i.(current' \leq i < length') \supset \\ \qquad ((line'@i = ((\,\mathsf{but\_last}(\mathsf{fst}\ l_1))\frown(\mathsf{snd}\ l_1))\downarrow i))\end{pmatrix}$$

We open a subwindow to examine the contents of the line before the current position.

Window: 9

$$\begin{array}{l}\vdots \\ \text{¡}\ `length = \#((\mathsf{but\_last}(\mathsf{fst}\ l_1))\triangleleft(\mathsf{last}(\mathsf{fst}\ l_1))) + \#(\mathsf{snd}\ l_1) \\ \text{¡}\ `current = \#((\mathsf{but\_last}(\mathsf{fst}\ l_1))\triangleleft(\mathsf{last}(\mathsf{fst}\ l_1))) \\ \text{¡}\ \boxed{\forall i.(i < `length) \supset (`line@i = (((\mathsf{but\_last}(\mathsf{fst}\ l_1))\triangleleft(\mathsf{last}(\mathsf{fst}\ l_1)))\frown(\mathsf{snd}\ l_1))\downarrow i)} \\ !\ length' = \#(\,\mathsf{but\_last}(\mathsf{fst}\ l_1)) + \#(\mathsf{snd}\ l_1) \\ !\ current' = \#(\,\mathsf{but\_last}(\mathsf{fst}\ l_1)) \\ !\ i < current' \\ \supset\star\ line'@i = ((\,\mathsf{but\_last}(\mathsf{fst}\ l_1))\frown(\mathsf{snd}\ l_1))\downarrow i\end{array}$$

From the context we know that i is less than $\#(\,\mathsf{but\_last}(\mathsf{fst}\ l_1))$. We can therefore simplify the focus with the theorem $\frown\downarrow 1\_\mathsf{THM}$.

$$\supset\star\ line'@i = (\,\mathsf{but\_last}(\mathsf{fst}\ l_1))\downarrow i$$

Next we open a subwindow on the lemma which describes the contents of the line prior to the edit operation.

Window: 10

$$\begin{array}{l}\vdots \\ \text{¡}\ `length = \#((\mathsf{but\_last}(\mathsf{fst}\ l_1))\triangleleft(\mathsf{last}(\mathsf{fst}\ l_1))) + \#(\mathsf{snd}\ l_1) \\ !\ current' = \#(\,\mathsf{but\_last}(\mathsf{fst}\ l_1)) \\ !\ i < current' \\ \subset\star\forall i.(i < `length) \supset (`line@i = (((\mathsf{but\_last}(\mathsf{fst}\ l_1))\triangleleft(\mathsf{last}(\mathsf{fst}\ l_1)))\frown(\mathsf{snd}\ l_1))\downarrow i)\end{array}$$

We specialise the universal quantification to the particular i we are interested in.

$$\subset\star\,(i < `length) \supset (`line@i = (((\mathsf{but\_last}(\mathsf{fst}\ l_1))\triangleleft(\mathsf{last}(\mathsf{fst}\ l_1)))\frown(\mathsf{snd}\ l_1))\downarrow i)$$

From the context and the theorem $\#\triangleleft\_\mathsf{THM}$ we can deduce that i is less than $`length$ and we simplify the focus accordingly.

$$\subset\star\ `line@i = (((\mathsf{but\_last}(\mathsf{fst}\ l_1))\triangleleft(\mathsf{last}(\mathsf{fst}\ l_1)))\frown(\mathsf{snd}\ l_1))\downarrow i$$

Similarly, we can deduce that i is less than $\#(((\mathsf{but\_last}(\mathsf{fst}\ l_1))\triangleleft(\mathsf{last}(\mathsf{fst}\ l_1)))\supset(\mathsf{snd}\ l_1))$ using $\#\triangleleft$_THM and $\#\supset$_THM, which allows us allows us to use $\supset\downarrow 1$_THM to simplify the focus again.

$$\subset\star\ `line@i = ((\mathsf{but\_last}(\mathsf{fst}\ l_1))\triangleleft(\mathsf{last}(\mathsf{fst}\ l_1)))\downarrow i$$

Finally, since i is less than $\#(\mathsf{but\_last}(\mathsf{fst}\ l_1))$ we may use $\triangleleft\downarrow i$_THM to simplify the focus as follows.

$$\subset\star\ `line@i = (\mathsf{but\_last}(\mathsf{fst}\ l_1))\downarrow i$$

Having completed the simplification we close this window.

Window: 9

$$\begin{array}{ll} & \vdots \\ \text{¡} & `line@i = (\mathsf{but\_last}(\mathsf{fst}\ l_1))\downarrow i \\ \supset\star & line'@i = (\mathsf{but\_last}(\mathsf{fst}\ l_1))\downarrow i \end{array}$$

We can see from the lemma just derived that the contents of the line at i were unaffected by the delete operation. We use this fact to refine the focus into one which expresses the desired relation purely in terms of the concrete variables.

$$\supset\star\ line'@i = `line@i$$

The window is now closed.

Window: 8

$$\begin{array}{lll} & & \vdots \\ & \text{¡} & `length = \#((\mathsf{but\_last}(\mathsf{fst}\ l_1))\triangleleft(\mathsf{last}(\mathsf{fst}\ l_1))) + \#(\mathsf{snd}\ l_1) \\ & \text{¡} & \forall i.\,(i < `length) \supset (`line@i = (((\mathsf{but\_last}(\mathsf{fst}\ l_1))\triangleleft(\mathsf{last}(\mathsf{fst}\ l_1)))\supset(\mathsf{snd}\ l_1))\downarrow i) \\ & ! & length' = \#(\mathsf{but\_last}(\mathsf{fst}\ l_1)) + \#(\mathsf{snd}\ l_1) \\ & ! & current' = \#(\mathsf{but\_last}(\mathsf{fst}\ l_1)) \\ \supset\star & & (\forall i.\,(i < current') \supset (line'@i = `line@i)) \\ & \wedge & \left(\begin{array}{l}\forall i.\,(current' \le i < length') \supset \\ \qquad ((line'@i = ((\mathsf{but\_last}(\mathsf{fst}\ l_1))\supset(\mathsf{snd}\ l_1))\downarrow i))\end{array}\right) \end{array}$$

The refinement of the requirements for the contents of the line after the current position is similar to the refinement just performed for the contents of the line before the current position. We therefore do not present the details of this refinement, but will skip straight to its result.

$$\begin{array}{lll} \supset\star & & \Big(\forall i.\,(i < current') \supset (line'@i = `line@i)\Big) \\ & \wedge & \Big(\forall i.\,(current' \le i < length') \supset (line'@i = `line@(i+1))\Big) \end{array}$$

Now that we have expressed the requirements on the contents of the line purely in terms of the concrete variables we close the window.

Window: 7

$$
\begin{array}{l}
\vdots \\
\mathsf{i}\ \grave{}length = \#(\mathsf{fst}\ l_1) + \#(\mathsf{snd}\ l_1) \\
\mathsf{i}\ \grave{}current = \#(\mathsf{fst}\ l_1) \\
\mathsf{i}\ \grave{}length \leq MaxLength \\
\mathsf{i}\ \grave{}length = \#((\mathsf{but\_last}(\mathsf{fst}\ l_1)) \triangleleft (\mathsf{last}(\mathsf{fst}\ l_1))) + \#(\mathsf{snd}\ l_1) \\
\mathsf{i}\ \grave{}current = \#((\mathsf{but\_last}(\mathsf{fst}\ l_1)) \triangleleft (\mathsf{last}(\mathsf{fst}\ l_1))) \\
\sqsupseteq_3 \star \left| \begin{array}{ll}
 & length' = \#(\,\mathsf{but\_last}(\mathsf{fst}\ l_1)) + \#(\mathsf{snd}\ l_1) \\
\wedge & current' = \#(\,\mathsf{but\_last}(\mathsf{fst}\ l_1)) \\
\wedge & length' \leq MaxLength \\
\wedge & \Big(\forall i.\,(i < current') \supset (line'@i = \grave{}line@i)\Big) \\
\wedge & \Big(\forall i.\,(current' \leq i < length') \supset (line'@i = \grave{}line@(i+1))\Big) \\
\wedge & offset' = \grave{}offset
\end{array} \right|
\end{array}
$$

Using the lemmas

$$
\begin{array}{l}
\grave{}length = \#((\mathsf{but\_last}(\mathsf{fst}\ l_1)) \triangleleft (\mathsf{last}(\mathsf{fst}\ l_1))) + \#(\mathsf{snd}\ l_1) \\
\grave{}current = \#((\mathsf{but\_last}(\mathsf{fst}\ l_1)) \triangleleft (\mathsf{last}(\mathsf{fst}\ l_1)))
\end{array}
$$

and the theorems $\#\triangleleft\_\mathsf{THM}$, we can derive the following theorems:

$$
\begin{array}{l}
\ldots \vdash \grave{}length - 1 = \#(\,\mathsf{but\_last}(\mathsf{fst}\ l_1)) + \#(\mathsf{snd}\ l_1) \\
\ldots \vdash \grave{}current - 1 = \#(\,\mathsf{but\_last}(\mathsf{fst}\ l_1))
\end{array}
$$

These theorems allow us to simplify the requirements on *length* and *current* as follows.

$$
\sqsupseteq_3 \star \left| \begin{array}{ll}
 & length' = \grave{}length - 1 \\
\wedge & current' = \grave{}current - 1 \\
\wedge & length' \leq MaxLength \\
\wedge & \Big(\forall i.\,(i < current') \supset (line'@i = \grave{}line@i)\Big) \\
\wedge & \Big(\forall i.\,(current' \leq i < length') \supset (line'@i = \grave{}line@(i+1))\Big) \\
\wedge & offset' = \grave{}offset
\end{array} \right|
$$

The current focus requires us to decrement the values of *length* and *current* while moving the contents of the line above *current* down one position. Let us decompose these requirements into a requirement to first decrement the values of *length* and *current*, then update the contents of the line.

$$
\begin{array}{l}
\sqsupseteq_3 \star \left| \begin{array}{ll}
 & length' = \grave{}length - 1 \\
\wedge & current' = \grave{}current - 1 \\
\wedge & length' \leq MaxLength \\
\wedge & (line', offset') = (\grave{}line, \grave{}offset)
\end{array} \right| ; \\
\left| \begin{array}{ll}
 & (length', current', offset') = (\grave{}length, \grave{}current, \grave{}offset) \\
\wedge & \Big(\forall i.\,(i < \grave{}current) \supset (line'@i = \grave{}line@i)\Big) \\
\wedge & \Big(\forall i.\,(\grave{}current \leq i < \grave{}length) \supset (line'@i = \grave{}line@(i+1))\Big)
\end{array} \right|
\end{array}
$$

The first of these requirements is easily implemented with an assignment statement.

$$\sqsupseteq_3 \star (\mathit{length}', \mathit{current}') := (\grave{}\mathit{length} - 1, \grave{}\mathit{current} - 1);$$
$$\boxed{\left|\begin{array}{ll} & (\mathit{length}', \mathit{current}', \mathit{offset}') = (\grave{}\mathit{length}, \grave{}\mathit{current}, \grave{}\mathit{offset}) \\ \wedge & \Big(\forall i.\, (i < \grave{}\mathit{current}) \supset (\mathit{line}'\mathtt{@}i = \grave{}\mathit{line}\mathtt{@}i)\Big) \\ \wedge & \Big(\forall i.\, (\grave{}\mathit{current} \leq i < \grave{}\mathit{length}) \supset (\mathit{line}'\mathtt{@}i = \grave{}\mathit{line}\mathtt{@}(i+1))\Big) \end{array}\right|}$$

We now open a subwindow to concentrate on implementing the second of these requirements.

Window: 8

$$\begin{array}{l} \vdots \\ \mathsf{i}\ \tilde{}\mathit{length} = \#(\mathsf{fst}\ l_1) + \#(\mathsf{snd}\ l_1) \\ \mathsf{i}\ \tilde{}\mathit{current} = \#(\mathsf{fst}\ l_1) \\ !\ \tilde{}\mathit{length} \leq \mathit{MaxLength} \\ !\ ((\grave{}\mathit{length}, \grave{}\mathit{current}) := (\tilde{}\mathit{length} - 1, \tilde{}\mathit{current} - 1)) \neq \mathsf{F}_3 \\ \sqsupseteq_3 \star \left|\begin{array}{ll} & (\mathit{length}', \mathit{current}', \mathit{offset}') = (\grave{}\mathit{length}, \grave{}\mathit{current}, \grave{}\mathit{offset}) \\ \wedge & \Big(\forall i.\, (i < \grave{}\mathit{current}) \supset (\mathit{line}'\mathtt{@}i = \grave{}\mathit{line}\mathtt{@}i)\Big) \\ \wedge & \Big(\forall i.\, (\grave{}\mathit{current} \leq i < \grave{}\mathit{length}) \supset (\mathit{line}'\mathtt{@}i = \grave{}\mathit{line}\mathtt{@}(i+1))\Big) \end{array}\right| \end{array}$$

We implement these requirements with a recursive block. First we introduce a new variable j, which we initialise to the current position. The recursive block will increment j from an initial value at the current position up to the length of the line, while moving the contents of the line at j down one position on each step.

$$\sqsupseteq_3 \star \mathsf{var}\, j.\, j' := \grave{}\mathit{current};$$
$$\left|\begin{array}{ll} & (\mathit{length}', \mathit{current}', \mathit{offset}') = (\grave{}\mathit{length}, \grave{}\mathit{current}, \grave{}\mathit{offset}) \\ \wedge & \Big(\forall i.\, (i < \boxed{\grave{}\mathit{current}}) \supset (\mathit{line}'\mathtt{@}i = \grave{}\mathit{line}\mathtt{@}i)\Big) \\ \wedge & \Big(\forall i.\, (\grave{}\mathit{current} \leq i < \grave{}\mathit{length}) \supset (\mathit{line}'\mathtt{@}i = \grave{}\mathit{line}\mathtt{@}(i+1))\Big) \end{array}\right|$$

We open a subwindow to replace the first instance of $\grave{}\mathit{current}$ in the description of the contents of the line with $\grave{}j$.

Window: 9 Note that within the context of this window, the state σ includes the local variable j.

$$\begin{array}{l} \vdots \\ !\ (\grave{}j := \tilde{}\mathit{current}) \neq \mathsf{F}_3 \\ \geq \star\ \grave{}\mathit{current} \end{array}$$

From the assumption shown, and the semantics of assignments, we know that $\grave{}current$ is equal to $\tilde{}current$.

$$\geq \star \tilde{}current$$

Again, from the assumption, we know that $\tilde{}current$ is equal to $\grave{}j$.

$$\geq \star \grave{}j$$

Having made the replacement we close the window.

Window: 8

$$\begin{array}{l}
\vdots \\
\mathsf{i}\ \tilde{}length = \#(\mathsf{fst}\ l_1) + \#(\mathsf{snd}\ l_1) \\
\mathsf{i}\ \tilde{}current = \#(\mathsf{fst}\ l_1) \\
!\ \tilde{}length \leq MaxLength \\
!\ ((\grave{}length, \grave{}current) := (\tilde{}length - 1, \tilde{}current - 1)) \neq \mathsf{F}_3 \\
\sqsupseteq_3 \star \mathsf{var}\, j.\, j' := \grave{}current; \\
\qquad \left\lfloor \begin{array}{ll} & (length', current', offset') = (\grave{}length, \grave{}current, \grave{}offset) \\ \wedge & \Big(\forall i.\, (i < \grave{}j) \supset (line'@i = \grave{}line@i)\Big) \\ \wedge & \Big(\forall i.\, (\grave{}current \leq i < \grave{}length) \supset (line'@i = \grave{}line@(i+1))\Big) \end{array} \right\rceil
\end{array}$$

Similarly, the second instance of $\grave{}current$ in the description of the contents of the line can be replaced by $\grave{}j$.

$$\begin{array}{l}
\sqsupseteq_3 \star \mathsf{var}\, j.\, j' := \grave{}current; \\
\qquad \boxed{\left\lfloor \begin{array}{ll} & (length', current', offset') = (\grave{}length, \grave{}current, \grave{}offset) \\ \wedge & \Big(\forall i.\, (i < \grave{}j) \supset (line'@i = \grave{}line@i)\Big) \\ \wedge & \Big(\forall i.\, (\grave{}j \leq i < \grave{}length) \supset (line'@i = \grave{}line@(i+1))\Big) \end{array} \right\rceil}
\end{array}$$

We now open a window to concentrate our attention on what will become the recursive block.

Window: 9

$$\begin{array}{l}
\vdots \\
\mathsf{i}\ \tilde{}\tilde{}length = \#(\mathsf{fst}\ l_1) + \#(\mathsf{snd}\ l_1) \\
\mathsf{i}\ \tilde{}\tilde{}current = \#(\mathsf{fst}\ l_1) \\
!\ \tilde{}\tilde{}length \leq MaxLength \\
!\ ((\grave{}length, \grave{}current) := (\tilde{}length - 1, \tilde{}current - 1)) \neq \mathsf{F}_3 \\
!\ (\grave{}j := \tilde{}current) \neq \mathsf{F}_3 \\
\sqsupseteq_3 \star \left\lfloor \begin{array}{ll} & (length', current', offset') = (\grave{}length, \grave{}current, \grave{}offset) \\ \wedge & \Big(\forall i.\, (i < \grave{}j) \supset (line'@i = \grave{}line@i)\Big) \\ \wedge & \Big(\forall i.\, (\grave{}j \leq i < \grave{}length) \supset (line'@i = \grave{}line@(i+1))\Big) \end{array} \right\rceil
\end{array}$$

We are almost ready to introduce the recursive block, which we give the label D, for 'delete'. The invariant of this block is simple. First we need to know that the j never progresses beyond the length of the line. Second, when updating the array *line*, we need to know that the position being updated is within the physical bounds of the array. The following invariant provides this information.

$$\stackrel{\text{def}}{\vdash} I_d\ \sigma = (j \leq length) \wedge (length \leq MaxChar) \qquad [I_d\text{\_DEF}]$$

The variant of the block should require that j progresses towards the length of the line. This requirement is captured by defining the variant as follows.

$$\stackrel{\text{def}}{\vdash} v_d\ \sigma = |length - j| \qquad [v_d\text{\_DEF}]$$

The original specification that the block will be introduced to implement is the current focus.

$$\stackrel{\text{def}}{\vdash} O_d(\sigma_1, \sigma_2) = \left\lfloor \begin{array}{ll} & (length_2, current_2, offset_2) = (length_1, current_1, offset_1) \\ \wedge & \big(\forall i.\,(i < j_1) \supset (line_2 @ i = line_1 @ i)\big) \\ \wedge & \big(\forall i.\,(j_1 \leq i < length_1) \supset (line_2 @ i = line_1 @ (i+1))\big) \end{array} \right\rfloor \qquad [O_d\text{\_DEF}]$$

In order to introduce a recursive block we must know that its invariant is satisfied. We open a subwindow with focus $I_d\ `\sigma$ to prove this is the case.

Window: 10

$$\begin{array}{l} \vdots \\ \mathsf{i}\ \ ``length = \#(\mathsf{fst}\ l_1) + \#(\mathsf{snd}\ l_1) \\ \mathsf{i}\ \ ``current = \#(\mathsf{fst}\ l_1) \\ !\ ``length \leq MaxLength \\ !\ ((`length, `current) := (``length - 1, ``current - 1)) \neq \mathsf{F}_3 \\ !\ (`j := ``current) \neq \mathsf{F}_3 \\ \supset \star\ I_d\ `\sigma \end{array}$$

First, we expand the definition of I_d.

$$\supset \star\ (`j \leq `length) \wedge (`length \leq MaxChar)$$

The assumption $(`j := ``current) \neq \mathsf{F}_3$ tells us that $`j$ is equal to $``current$, and that $`length$ is equal to $``length$.

$$\supset \star\ (``current \leq ``length) \wedge (``length \leq MaxChar)$$

Next, we rewrite the focus using the assumption:

$$((\grave{}\mathit{length}, \grave{}\mathit{current}) := (\grave{}\mathit{length} - 1, \grave{}\mathit{current} - 1)) \neq \mathsf{F}_3$$

$$\supset \star (\grave{}\grave{}\mathit{current} - 1 \leq \grave{}\grave{}\mathit{length} - 1) \wedge (\grave{}\grave{}\mathit{length} - 1 \leq \mathit{MaxChar})$$

We add 1 to both sides of each inequation.

$$\supset \star (\grave{}\grave{}\mathit{current} \leq \grave{}\grave{}\mathit{length}) \wedge (\grave{}\grave{}\mathit{length} \leq \mathit{MaxChar} + 1)$$

Now we rewrite with the assumptions:

$$\grave{}\grave{}\mathit{current} = \#(\mathsf{fst}\ l_1)$$
$$\grave{}\grave{}\mathit{length} = \#(\mathsf{fst}\ l_1) + \#(\mathsf{snd}\ l_1)$$

$$\supset \star (\#(\mathsf{fst}\ l_1) \leq \#(\mathsf{fst}\ l_1) + \#(\mathsf{snd}\ l_1)) \wedge (\grave{}\grave{}\mathit{length} \leq \mathit{MaxChar} + 1)$$

The first conjunct of the focus is obviously true.

$$\supset \star \grave{}\grave{}\mathit{length} \leq \mathit{MaxChar} + 1$$

The definition of *MaxLength* states that $\mathit{MaxChar} + 1$ is equal to *MaxLength*.

$$\supset \star \grave{}\grave{}\mathit{length} \leq \mathit{MaxLength}$$

The truth of the current focus now follows directly from one of the assumptions.

$$\supset \star \mathsf{T}$$

Having established that the invariant does indeed hold, we close this window.

Window: 9

$$\begin{array}{l} \vdots \\ \mathsf{i}\ I_d\ \grave{}\sigma \\ \exists_3 \star \left| \begin{array}{ll} & (\mathit{length}', \mathit{current}', \mathit{offset}') = (\grave{}\mathit{length}, \grave{}\mathit{current}, \grave{}\mathit{offset}) \\ \wedge & \left(\forall i.\, (i < \grave{}j) \supset (\mathit{line}'\texttt{@}i = \grave{}\mathit{line}\texttt{@}i)\right) \\ \wedge & \left(\forall i.\, (\grave{}j \leq i < \grave{}\mathit{length}) \supset (\mathit{line}'\texttt{@}i = \grave{}\mathit{line}\texttt{@}(i+1))\right) \end{array} \right| \end{array}$$

Now that we know the invariant holds, we can introduce the recursive block.

$$\exists_3 \star \mathsf{label}_{(v_d, O_d, I_d)}\ D.$$

$$\boxed{\left| \begin{array}{ll} & (\mathit{length}', \mathit{current}', \mathit{offset}') = (\grave{}\mathit{length}, \grave{}\mathit{current}, \grave{}\mathit{offset}) \\ \wedge & \left(\forall i.\, (i < \grave{}j) \supset (\mathit{line}'\texttt{@}i = \grave{}\mathit{line}\texttt{@}i)\right) \\ \wedge & \left(\forall i.\, (\grave{}j \leq i < \grave{}\mathit{length}) \supset (\mathit{line}'\texttt{@}i = \grave{}\mathit{line}\texttt{@}(i+1))\right) \end{array} \right|}$$

We open a subwindow so that we may derive the body of the block.

Window: 10

$$
\begin{array}{l}
\vdots \\
! \; I_d \; \grave{} \sigma \\
! \; \forall \sigma_1 \, \sigma_2 . \, I_d \, \sigma_1 \wedge (v_d \, \sigma_1 < v_d \, \grave{} \sigma) \supset (D(\sigma_1, \sigma_2) \sqsupseteq_3 O_d(\sigma_1, \sigma_2)) \\
\sqsupseteq_3 \star \left\lfloor \begin{array}{ll}
 & (length', current', offset') = (\grave{} length, \grave{} current, \grave{} offset) \\
\wedge & \Big(\forall i . (i < \grave{} j) \supset (line' @ i = \grave{} line @ i)\Big) \\
\wedge & \Big(\forall i . (\grave{} j \leq i < \grave{} length) \supset (line' @ i = \grave{} line @ (i+1))\Big)
\end{array} \right\rfloor
\end{array}
$$

In the context of the body of the block, we may assume that the invariant holds. We may also assume that in any context where the invariant holds and in which the value of the variant function has decreased from its present value, a recursive call to the label of the block will implement the original specification. We expand the definition of the invariant and add both of its conjuncts as lemmas.

$$
\begin{array}{l}
¡ \; \grave{} j \leq \grave{} length \\
¡ \; \grave{} length \leq MaxChar \\
\sqsupseteq_3 \star \left\lfloor \begin{array}{ll}
 & (length', current', offset') = (\grave{} length, \grave{} current, \grave{} offset) \\
\wedge & \Big(\forall i . (i < \grave{} j) \supset (line' @ i = \grave{} line @ i)\Big) \\
\wedge & \Big(\forall i . (\grave{} j \leq i < \grave{} length) \supset (line' @ i = \grave{} line @ (i+1))\Big)
\end{array} \right\rfloor
\end{array}
$$

If $\grave{} j$ is equal to $\grave{} length$ then the requirements of the focus have already been satisfied. We introduce a conditional to consider this case separately.

$$
\begin{array}{l}
\sqsupseteq_3 \star \, \mathsf{if} \; \grave{} j = \grave{} length \; \mathsf{then} \\
\qquad \boxed{\left\lfloor \begin{array}{ll}
 & (length', current', offset') = (\grave{} length, \grave{} current, \grave{} offset) \\
\wedge & \Big(\forall i . (i < \grave{} j) \supset (line' @ i = \grave{} line @ i)\Big) \\
\wedge & \Big(\forall i . (\grave{} j \leq i < \grave{} length) \supset (line' @ i = \grave{} line @ (i+1))\Big)
\end{array} \right\rfloor} \\
\quad \mathsf{else} \\
\qquad \left\lfloor \begin{array}{ll}
 & (length', current', offset') = (\grave{} length, \grave{} current, \grave{} offset) \\
\wedge & \Big(\forall i . (i < \grave{} j) \supset (line' @ i = \grave{} line @ i)\Big) \\
\wedge & \Big(\forall i . (\grave{} j \leq i < \grave{} length) \supset (line' @ i = \grave{} line @ (i+1))\Big)
\end{array} \right\rfloor
\end{array}
$$

We open a subwindow on the case where $\grave{} j$ is equal to $\grave{} length$.

Window: 11

$$
\begin{array}{l}
\vdots \\
! \; \grave{} j = \grave{} length \\
\sqsupseteq_3 \star \left\lfloor \begin{array}{ll}
 & (length', current', offset') = (\grave{} length, \grave{} current, \grave{} offset) \\
\wedge & \Big(\forall i . (i < \grave{} j) \supset (line' @ i = \grave{} line @ i)\Big) \\
\wedge & \Big(\forall i . (\grave{} j \leq i < \grave{} length) \supset (line' @ i = \grave{} line @ (i+1))\Big)
\end{array} \right\rfloor
\end{array}
$$

Using the assumption that $\grave{}j$ is equal to $\grave{}length$, we can see that the current focus can be implemented with skip.

$$\sqsupseteq_3 \star \mathsf{skip}$$

Having implemented this case, we return to the parent window.

Window: 10

$$\begin{array}{l}
\vdots \\
! \; \forall \sigma_1 \, \sigma_2 . \, I_d \, \sigma_1 \wedge (v_d \, \sigma_1 < v_d \, \grave{}\sigma) \supset (D(\sigma_1, \sigma_2) \sqsupseteq_3 O_d(\sigma_1, \sigma_2)) \\
¡ \; \grave{}j \leq \grave{}length \\
¡ \; \grave{}length \leq MaxChar \\
\sqsupseteq_3 \star \mathsf{if} \; \grave{}j = \grave{}length \; \mathsf{then} \; \mathsf{skip} \\
\quad \mathsf{else} \\
\qquad \left\lfloor \begin{array}{ll} & (length', current', offset') = (\grave{}length, \grave{}current, \grave{}offset) \\ \wedge & \Big(\forall i . (i < \grave{}j) \supset (line'@i = \grave{}line@i)\Big) \\ \wedge & \Big(\forall i . (\grave{}j \leq i < \grave{}length) \supset (line'@i = \grave{}line@(i+1))\Big) \end{array} \right\rfloor
\end{array}$$

We now open a subwindow on the other case of the conditional, in which $\grave{}j$ is not equal to $\grave{}length$. In this case we know that some of the elements of the array *line* still need to be moved down a position.

Window: 11

$$\begin{array}{l}
\vdots \\
! \; \forall \sigma_1 \, \sigma_2 . \, I_d \, \sigma_1 \wedge (v_d \, \sigma_1 < v_d \, \grave{}\sigma) \supset (D(\sigma_1, \sigma_2) \sqsupseteq_3 O_d(\sigma_1, \sigma_2)) \\
¡ \; \grave{}j \leq \grave{}length \\
¡ \; \grave{}length \leq MaxChar \\
! \; \grave{}j \neq \grave{}length \\
\sqsupseteq_3 \star \left\lfloor \begin{array}{ll} & (length', current', offset') = (\grave{}length, \grave{}current, \grave{}offset) \\ \wedge & \Big(\forall i . (i < \grave{}j) \supset (line'@i = \grave{}line@i)\Big) \\ \wedge & \Big(\forall i . (\grave{}j \leq i < \grave{}length) \supset (line'@i = \grave{}line@(i+1))\Big) \end{array} \right\rfloor
\end{array}$$

The focus of this window is a requirement to maintain the contents of the array *line* below the $\grave{}j$th position, while moving elements of that array above $\grave{}j$ down one position. We introduce a sequential composition and prefix this requirement with one which does some of the work by moving a block of elements of *line* down by one position. The second (original) requirement can then be refined into a requirement to finish the task.

The new specification is required to increase the value of j while maintaining the invariant I_d. It should also move the elements of the array between the initial and

final values of j down by one position. The other elements of the array, and the other variables should be left unchanged. The second statement of the composition then need only move down those elements of the array above the new value of j.

$$\sqsupseteq_3 \star \left| \begin{array}{ll} & \Big(\forall i.\,(i < \grave{}j) \supset (line\acute{}@i = \grave{}line@i)\Big) \\ \wedge & \Big(\forall i.\,(\grave{}j \leq i < j\acute{}) \supset (line\acute{}@i = \grave{}line@(i+1))\Big) \\ \wedge & \Big(\forall i.\,(j\acute{} \leq i < \grave{}length + 1) \supset (line\acute{}@i = \grave{}line@i)\Big) \\ \wedge & j\acute{} \leq length\acute{} \\ \wedge & length\acute{} \leq MaxChar \\ \wedge & \grave{}j < j\acute{} \\ \wedge & (length\acute{}, current\acute{}, offset\acute{}) = (\grave{}length, \grave{}current, \grave{}offset) \end{array} \right| ;$$

$$\boxed{\left| \begin{array}{ll} & (length\acute{}, current\acute{}, offset\acute{}) = (\breve{}length, \breve{}current, \breve{}offset) \\ \wedge & \Big(\forall i.\,(i < \breve{}j) \supset (line\acute{}@i = \breve{}line@i)\Big) \\ \wedge & \Big(\forall i.\,(\breve{}j \leq i < \breve{}length) \supset (line\acute{}@i = \breve{}line@(i+1))\Big) \end{array} \right|}$$

We now open a subwindow on the second (original) requirement of the sequential composition. Using the assumptions that follow from the first clause of the sequential composition, we will be able to transform this clause into its original form, and from there refine it into a recursive call to the label of the block.

Window: 12

$$\begin{array}{l} \vdots \\ !\ \forall \sigma_1\, \sigma_2.\, I_d\, \sigma_1 \wedge (v_d\, \sigma_1 < v_d\, \breve{}\sigma) \supset (D(\sigma_1, \sigma_2) \sqsupseteq_3 O_d(\sigma_1, \sigma_2)) \\ ¡\ \breve{}j \leq \breve{}length \\ !\ \breve{}j \neq \breve{}length \\ !\ \forall i.\,(i < \breve{}j) \supset (\grave{}line@i = \breve{}line@i) \\ !\ \forall i.\,(\breve{}j \leq i < \grave{}j) \supset (\grave{}line@i = \breve{}line@(i+1)) \\ !\ \forall i.\,(\grave{}j \leq i < \breve{}length + 1) \supset (\grave{}line@i = \breve{}line@i) \\ !\ \grave{}j \leq \grave{}length \\ ¡\ \grave{}length \leq MaxChar \\ !\ \breve{}j < \grave{}j \\ !\ (\grave{}length, \grave{}current, \grave{}offset) = (\breve{}length, \breve{}current, \breve{}offset) \end{array}$$

$$\sqsupseteq_3 \star \left| \begin{array}{ll} & (length\acute{}, current\acute{}, offset\acute{}) = (\breve{}length, \breve{}current, \breve{}offset) \\ \wedge & \Big(\forall i.\,(i < \breve{}j) \supset (line\acute{}@i = \breve{}line@i)\Big) \\ \wedge & \Big(\forall i.\,(\breve{}j \leq i < \breve{}length) \supset (line\acute{}@i = \breve{}line@(i+1))\Big) \end{array} \right|$$

We rewrite the focus with the assumption that *length*, *current* and *offset* are unchanged.

$$\sqsupseteq_3 \star \left| \begin{array}{ll} & (length\acute{}, current\acute{}, offset\acute{}) = (\grave{}length, \grave{}current, \grave{}offset) \\ \wedge & \Big(\forall i.\,(i < \breve{}j) \supset (line\acute{}@i = \breve{}line@i)\Big) \\ \wedge & \Big(\forall i.\,(\breve{}j \leq i < \grave{}length) \supset (line\acute{}@i = \breve{}line@(i+1))\Big) \end{array} \right|$$

We partition the requirement that the elements of *line* above $\breve{}j$ should be moved down by one into two cases. The first case states that the variables between $\breve{}j$ and $\grave{}j$ should be moved down by one (this case will have been taken care of by the previous statement). The second case states that the variable above $\grave{}j$ should be moved down by one.

$$\sqsupseteq_3 \star \left| \begin{array}{ll} & (\mathit{length}', \mathit{current}', \mathit{offset}') = (\grave{}\mathit{length}, \grave{}\mathit{current}, \grave{}\mathit{offset}) \\ \wedge & \Big(\forall i.\, (i < \breve{}j) \supset \boxed{(\mathit{line}'@i = \breve{}\mathit{line}@i)}\Big) \\ \wedge & \Big(\forall i.\, (\breve{}j \leq i < \grave{}j) \supset (\mathit{line}'@i = \breve{}\mathit{line}@(i+1))\Big) \\ \wedge & \Big(\forall i.\, (\grave{}j \leq i < \grave{}\mathit{length}) \supset (\mathit{line}'@i = \breve{}\mathit{line}@(i+1))\Big) \end{array} \right|$$

We open a subwindow on the term which describes the final contents of the array *line* below $\breve{}j$.

Window: 13

$$\begin{array}{l} \vdots \\ ! \; \forall i.\, (i < \breve{}j) \supset (\grave{}\mathit{line}@i = \breve{}\mathit{line}@i) \\ ! \; i < \breve{}j \\ \supset \star \mathit{line}'@i = \breve{}\mathit{line}@i \end{array}$$

We know from our assumptions that i is less than $\breve{}j$, and that for all such i, $\breve{}\mathit{line}@i$ is equal to $\grave{}\mathit{line}@i$. We may therefore transform the focus as follows.

$$\supset \star \mathit{line}'@i = \grave{}\mathit{line}@i$$

Having recast these requirements in terms of a relationship between the current initial and final states, we close the window.

Window: 12

$$\begin{array}{l} \vdots \\ ! \; \boxed{\forall \sigma_1\, \sigma_2.\, I_d\, \sigma_1 \wedge (v_d\, \sigma_1 < v_d\, \breve{}\sigma) \supset (D(\sigma_1, \sigma_2) \sqsupseteq_3 O_d(\sigma_1, \sigma_2))} \\ ¡ \; \breve{}j \leq \breve{}\mathit{length} \\ ! \; \breve{}j \neq \breve{}\mathit{length} \\ ! \; \forall i.\, (i < \breve{}j) \supset (\grave{}\mathit{line}@i = \breve{}\mathit{line}@i) \\ ! \; \forall i.\, (\breve{}j \leq i < \grave{}j) \supset (\grave{}\mathit{line}@i = \breve{}\mathit{line}@(i+1)) \\ ! \; \forall i.\, (\grave{}j \leq i < \breve{}\mathit{length} + 1) \supset (\grave{}\mathit{line}@i = \breve{}\mathit{line}@i) \\ ! \; \grave{}j \leq \grave{}\mathit{length} \\ ¡ \; \grave{}\mathit{length} \leq \mathit{MaxChar} \\ ! \; \breve{}j < \grave{}j \\ ! \; (\grave{}\mathit{length}, \grave{}\mathit{current}, \grave{}\mathit{offset}) = (\breve{}\mathit{length}, \breve{}\mathit{current}, \breve{}\mathit{offset}) \end{array}$$

$$\sqsupseteq_3 \star \left| \begin{array}{ll} & (\mathit{length}', \mathit{current}', \mathit{offset}') = (\grave{}\mathit{length}, \grave{}\mathit{current}, \grave{}\mathit{offset}) \\ \wedge & \Big(\forall i.\, (i < \breve{}j) \supset (\mathit{line}'@i = \grave{}\mathit{line}@i)\Big) \\ \wedge & \Big(\forall i.\, (\breve{}j \leq i < \grave{}j) \supset (\mathit{line}'@i = \breve{}\mathit{line}@(i+1))\Big) \\ \wedge & \Big(\forall i.\, (\grave{}j \leq i < \grave{}\mathit{length}) \supset (\mathit{line}'@i = \breve{}\mathit{line}@(i+1))\Big) \end{array} \right|$$

We may simplify the other two terms describing the final contents of *line* above $\breve{}j$ similarly. The details of these simplifications are not given here, only their end result.

$$\sqsupseteq_3 \star \left| \begin{array}{ll} & (length', current', offset') = (\grave{}length, \grave{}current, \grave{}offset) \\ \wedge & \Big(\forall i.\, (i < \breve{}j) \supset (line'@i = \grave{}line@i)\Big) \\ \wedge & \Big(\forall i.\, (\breve{}j \leq i < \grave{}j) \supset (line'@i = \grave{}line@i)\Big) \\ \wedge & \Big(\forall i.\, (\grave{}j \leq i < \grave{}length) \supset (line'@i = \grave{}line@(i+1))\Big) \end{array} \right|$$

We now see that the requirements on the final contents of *line* from zero up to $\breve{}j$, and from $\breve{}j$ up to $\grave{}j$ are the same. Since $\breve{}j$ is less than $\grave{}j$, we may amalgamate these two requirements into a single term that describes the final contents of *line* from zero up to $\grave{}j$.

$$\sqsupseteq_3 \star \left| \begin{array}{ll} & (length', current', offset') = (\grave{}length, \grave{}current, \grave{}offset) \\ \wedge & \Big(\forall i.\, (i < \grave{}j) \supset (line'@i = \grave{}line@i)\Big) \\ \wedge & \Big(\forall i.\, (\grave{}j \leq i < \grave{}length) \supset (line'@i = \grave{}line@(i+1))\Big) \end{array} \right|$$

The focus is now an instance of the original requirement O_d on the current initial and final states.

$$\sqsupseteq_3 \star O_d(\grave{}\sigma, \sigma')$$

From our assumptions, we know that if the invariant holds and the variant has decreased, then we will be able to use a recursive call to implement the current focus. We open a subwindow on this assumption to see if we can derive the fact we require.

Window: 13

$$\begin{array}{l} \vdots \\ ¡\ \breve{}j \leq \breve{}length \\ !\ \grave{}j \leq \grave{}length \\ !\ \grave{}length \leq MaxChar \\ !\ \breve{}j < \grave{}j \\ !\ (\grave{}length, \grave{}current, \grave{}offset) = (\breve{}length, \breve{}current, \breve{}offset) \\ \subset \star \forall \sigma_1\, \sigma_2.\, I_d\, \sigma_1 \wedge (v_d\, \sigma_1 < v_d\, \breve{}\sigma) \supset (D(\sigma_1, \sigma_2) \sqsupseteq_3 O_d(\sigma_1, \sigma_2)) \end{array}$$

First, we instantiate σ_1 and σ_2 in the focus to $\grave{}\sigma$ and σ' respectively.

$$\subset \star I_d\, \grave{}\sigma \wedge (v_d\, \grave{}\sigma < v_d\, \breve{}\sigma) \supset (D(\grave{}\sigma, \sigma') \sqsupseteq_3 O_d(\grave{}\sigma, \sigma'))$$

Next, we expand the definition of I_d:

$$\begin{array}{l} \subset \star ((\grave{}j \leq \grave{}length) \wedge (\grave{}length \leq MaxChar) \wedge (v_d\, \grave{}\sigma < v_d\, \breve{}\sigma)) \supset \\ \qquad (D(\grave{}\sigma, \sigma') \sqsupseteq_3 O_d(\grave{}\sigma, \sigma')) \end{array}$$

Using the assumption $\grave{}j \leq \grave{}length$ and $\grave{}length \leq MaxChar$, we may simplify the focus as follows:

$$\subset\star\,(v_d\ \grave{}\sigma < v_d\ \breve{}\sigma) \supset (D(\grave{}\sigma, \sigma') \sqsupseteq_3 O_d(\grave{}\sigma, \sigma'))$$

Now we expand the definition of v_d:

$$\subset\star\,(|\grave{}length - \grave{}j| < |\breve{}length - \breve{}j|) \supset (D(\grave{}\sigma, \sigma') \sqsupseteq_3 O_d(\grave{}\sigma, \sigma'))$$

The assumptions $\grave{}j \leq \grave{}length$ and $\breve{}j \leq \breve{}length$ allow us to remove the absolute value functions.

$$\subset\star\,(\grave{}length - \grave{}j < \breve{}length - \breve{}j) \supset (D(\grave{}\sigma, \sigma') \sqsupseteq_3 O_d(\grave{}\sigma, \sigma'))$$

Using the assumption $(\grave{}length, \ldots) = (\breve{}length, \ldots)$ we replace $\breve{}length$ with $\grave{}length$ in the focus.

$$\subset\star\,(\grave{}length - \grave{}j < \grave{}length - \breve{}j) \supset (D(\grave{}\sigma, \sigma') \sqsupseteq_3 O_d(\grave{}\sigma, \sigma'))$$

Next we make some simple algebraic transformations.

$$\subset\star\,(-\grave{}j < -\breve{}j) \supset (D(\grave{}\sigma, \sigma') \sqsupseteq_3 O_d(\grave{}\sigma, \sigma'))$$
$$\subset\star\,(\breve{}j < \grave{}j) \supset (D(\grave{}\sigma, \sigma') \sqsupseteq_3 O_d(\grave{}\sigma, \sigma'))$$

Finally, using the assumption $\breve{}j < \grave{}j$, we can derive the result we want.

$$\subset\star\, D(\grave{}\sigma, \sigma') \sqsupseteq_3 O_d(\grave{}\sigma, \sigma')$$

Having proved that a recursive call to the label implements the original requirements, we close the window.

Window: 12

$$\begin{array}{l} \vdots \\ \mathbf{i}\ D(\grave{}\sigma, \sigma') \sqsupseteq_3 O_d(\grave{}\sigma, \sigma') \\ \sqsupseteq_3 \star\, O_d(\grave{}\sigma, \sigma') \end{array}$$

Using the lemma we just derived, we refine the focus into a recursive call to the label of the block.

$$\sqsupseteq_3 \star\, D(\grave{}\sigma, \sigma')$$

Having completed the implementation of this window, we close it and return to its parent.

Window: 11

$$
\begin{array}{l}
\vdots \\
¡\ `j \leq `length \\
¡\ `length \leq MaxChar \\
!\ `j \neq `length \\
\sqsupseteq_3 \star \boxed{\left|\begin{array}{ll}
 & \big(\forall i.\,(i < `j) \supset (line'@i = `line@i)\big) \\
\wedge & \big(\forall i.\,(`j \leq i < j') \supset (line'@i = `line@(i+1))\big) \\
\wedge & \big(\forall i.\,(j' \leq i < `length + 1) \supset (line'@i = `line@i)\big) \\
\wedge & j' \leq length' \\
\wedge & length' \leq MaxChar \\
\wedge & `j < j' \\
\wedge & (length', current', offset') = (`length, `current, `offset)
\end{array}\right|} ; \\
D(`\sigma, \sigma')
\end{array}
$$

We now open a window on the first, unimplemented, clause of the composition. This is the clause that was required to perform some of the work of the block before the recursive call to the label was made to complete the task.

Window: 12

$$
\begin{array}{l}
\vdots \\
¡\ `j \leq `length \\
¡\ `length \leq MaxChar \\
!\ `j \neq `length \\
\sqsupseteq_3 \star \left|\begin{array}{ll}
 & \big(\forall i.\,(i < `j) \supset (line'@i = `line@i)\big) \\
\wedge & \big(\forall i.\,(`j \leq i < j') \supset (line'@i = `line@(i+1))\big) \\
\wedge & \big(\forall i.\,(j' \leq i < `length + 1) \supset (line'@i = `line@i)\big) \\
\wedge & j' \leq length' \\
\wedge & length' \leq MaxChar \\
\wedge & `j < j' \\
\wedge & (length', current', offset') = (`length, `current, `offset)
\end{array}\right|
\end{array}
$$

The focus of this window is a requirement to increase the value of j, and to move down the elements of the array *line* in the positions between the initial and final values of j (while leaving the elements of the array below $`j$ and above j' unchanged).

As yet, we have not decided by how much j should be increased and, correspondingly, how big a block of elements we should move. It is easiest if we chose to the minimum amount of work necessary with each step. This can be achieved by increasing j by 1. As a result, we only have to move a block of size 1, a task that can be accomplished with an assignment statement.

We now strengthen the specification so that in addition to its other clauses, it also requires that j be increased by 1.

$$\sqsupseteq_3 \star \left\lfloor \begin{array}{l} j' = \grave{}j + 1 \wedge \\ \boxed{\begin{array}{ll} & \Big(\forall i.\,(i < \grave{}j) \supset (line'@i = \grave{}line@i)\Big) \\ \wedge & \Big(\forall i.\,(\grave{}j \leq i < j') \supset (line'@i = \grave{}line@(i+1))\Big) \\ \wedge & \Big(\forall i.\,(j' \leq i < \grave{}length + 1) \supset (line'@i = \grave{}line@i)\Big) \\ \wedge & j' \leq length' \\ \wedge & length' \leq MaxChar \\ \wedge & \grave{}j < j' \end{array}} \\ \wedge \quad (length', current', offset') = (\grave{}length, \grave{}current, \grave{}offset) \end{array} \right\rfloor$$

Now that we have chosen to increase j by 1, we should be able to make some simplifications to the other requirements. We open a subwindow on the other requirements to see what simplifications can be made.

Window: 13

$$\begin{array}{ll} & \vdots \\ & ¡\ \grave{}j \leq \grave{}length \\ & ¡\ \grave{}length \leq MaxChar \\ & !\ \grave{}j \neq \grave{}length \\ & !\ j' = \grave{}j + 1 \\ & !\ (length', current', offset') = (\grave{}length, \grave{}current, \grave{}offset) \\ \supset \star & \Big(\forall i.\,(i < \grave{}j) \supset (line'@i = \grave{}line@i)\Big) \\ \wedge & \Big(\forall i.\,(\grave{}j \leq i < j') \supset (line'@i = \grave{}line@(i+1))\Big) \\ \wedge & \Big(\forall i.\,(j' \leq i < \grave{}length + 1) \supset (line'@i = \grave{}line@i)\Big) \\ \wedge & j' \leq length' \\ \wedge & length' \leq MaxChar \\ \wedge & \grave{}j < j' \end{array}$$

Using the assumptions $j' = \grave{}j + 1$ and $(length', \ldots) = (\grave{}length, \ldots)$, we rewrite the focus.

$$\begin{array}{ll} \supset \star & \Big(\forall i.\,(i < \grave{}j) \supset (line'@i = \grave{}line@i)\Big) \\ \wedge & \Big(\forall i.\,(\grave{}j \leq i < \grave{}j + 1) \supset (line'@i = \grave{}line@(i+1))\Big) \\ \wedge & \Big(\forall i.\,(\grave{}j + 1 \leq i < \grave{}length + 1) \supset (line'@i = \grave{}line@i)\Big) \\ \wedge & \grave{}j + 1 \leq \grave{}length \\ \wedge & \grave{}length \leq MaxChar \\ \wedge & \grave{}j < \grave{}j + 1 \end{array}$$

Using the assumptions, we simplify the focus as follows.

$$\begin{array}{ll} \supset\star & \Big(\forall i.\,(i < \grave{}j) \supset (line'@i = \grave{}line@i)\Big) \\ \quad\wedge & \Big(\forall i.\,(\grave{}j \leq i < \grave{}j + 1) \supset (line'@i = \grave{}line@(i+1))\Big) \\ \quad\wedge & \Big(\forall i.\,(\grave{}j + 1 \leq i < \grave{}length + 1) \supset (line'@i = \grave{}line@i)\Big) \end{array}$$

The universal quantification describing the contents of the line between $\grave{}j$ and $\grave{}j+1$ ranges over a single point. This means that the quantification can be removed in favour of a description of the requirements at that particular point.

$$\begin{array}{ll} \supset\star & \Big(\forall i.\,(i < \grave{}j) \supset (line'@i = \grave{}line@i)\Big) \\ \quad\wedge & (line'@\grave{}j = \grave{}line@(\grave{}j + 1)) \\ \quad\wedge & \Big(\forall i.\,(\grave{}j + 1 \leq i < \grave{}length + 1) \supset (line'@i = \grave{}line@i)\Big) \end{array}$$

Having simplified the requirements, we close the window.

Window: 12

$$\begin{array}{l} \text{¡}\ \grave{}j \leq \grave{}length \\ \text{¡}\ \grave{}length \leq MaxChar \\ !\ \grave{}j \neq \grave{}length \\ \exists_3\star \left|\begin{array}{ll} & j' = \grave{}j + 1 \\ \wedge & \Big(\forall i.\,(i < \grave{}j) \supset (line'@i = \grave{}line@i)\Big) \\ \wedge & (line'@\grave{}j = \grave{}line@(\grave{}j + 1)) \\ \wedge & \Big(\forall i.\,(\grave{}j + 1 \leq i < \grave{}length + 1) \supset (line'@i = \grave{}line@i)\Big) \\ \wedge & (length', current', offset') = (\grave{}length, \grave{}current, \grave{}offset) \end{array}\right| \end{array}$$

These requirements may be implemented with an assignment statement. (The actual proof of this step requires use of the theorems †@*ii*_THM and †@*ij*_THM, from Appendix A, and the assumptions.)

$$\exists_3\star\,(j', line') := (\grave{}j + 1, \grave{}line\dagger(\grave{}j, \grave{}line@(\grave{}j + 1)))$$

We close this window now that its focus has been implemented.

Window: 11

$$\begin{array}{l} \vdots \\ \exists_3\star\,(j', line') := (\grave{}j + 1, \grave{}line\dagger(\grave{}j, \grave{}line@(\grave{}j + 1)))\,;\, D(\grave{}\sigma, \sigma') \end{array}$$

Both clauses of the composition have been implemented, so we may close this window too.

Window: 10

$$\begin{array}{l} \vdots \\ \sqsupseteq_3 \star \mathsf{if}\ \grave{}j = \grave{}length\ \ \mathsf{then\ skip} \\ \quad \mathsf{else}\ (j', line') := (\grave{}j + 1, \grave{}line \dagger (\grave{}j, \grave{}line @ (\grave{}j + 1))); D(\grave{}\sigma, \sigma') \end{array}$$

The focus of this window has been entirely implemented, so we may close it.

Window: 9

$$\begin{array}{l} \sqsupseteq_3 \star \mathsf{label}_{(v_d, O_d, I_d)}\ D. \\ \qquad \mathsf{if}\ \grave{}j = \grave{}length\ \ \mathsf{then\ skip} \\ \qquad \mathsf{else}\ (j', line') := (\grave{}j + 1, \grave{}line \dagger (\grave{}j, \grave{}line @ (\grave{}j + 1))); D(\grave{}\sigma, \sigma') \end{array}$$

We have now implemented the body of the recursive block. This means that we may complete the refinement of the block by replacing it with its true meaning. The effect of this is to drop the subscripts from the labelled block, we no longer need the information contained in them.

$$\begin{array}{l} \sqsupseteq_3 \star \mathsf{label}\ D.\ \mathsf{if}\ \grave{}j = \grave{}length\ \ \mathsf{then\ skip} \\ \qquad\qquad \mathsf{else}\ (j', line') := (\grave{}j + 1, \grave{}line \dagger (\grave{}j, \grave{}line @ (\grave{}j + 1))); D(\grave{}\sigma, \sigma') \end{array}$$

We close this window now that it is completely implemented.

Window: 8

$$\begin{array}{l} \sqsupseteq_3 \star \mathsf{var}\ j.\ j' := \grave{}current; \\ \qquad \mathsf{label}\ D.\ \mathsf{if}\ \grave{}j = \grave{}length\ \ \mathsf{then\ skip} \\ \qquad\qquad \mathsf{else}\ (j', line') := (\grave{}j + 1, \grave{}line \dagger (\grave{}j, \grave{}line @ (\grave{}j + 1))); D(\grave{}\sigma, \sigma') \end{array}$$

The implementation of this window is also complete, so we close it.

Window: 7

$$\begin{array}{l} \sqsupseteq_3 \star (length', current') := (\grave{}length - 1, \grave{}current - 1); \\ \qquad \mathsf{var}\ j.\ j' := \grave{}current; \\ \qquad\qquad \mathsf{label}\ D.\ \mathsf{if}\ \grave{}j = \grave{}length\ \ \mathsf{then\ skip} \\ \qquad\qquad\qquad \mathsf{else}\ (j', line') := (\grave{}j + 1, \grave{}line \dagger (\grave{}j, \grave{}line @ (\grave{}j + 1))); D(\grave{}\sigma, \sigma') \end{array}$$

We may close this window too.

Window: 6

$$\sqsupseteq_3 \star \left(\begin{array}{l} \lfloor l_2 = del\ l_1 \rfloor \\ \underline{\triangle} \left(\begin{array}{l} (length', current') := (\grave{}length - 1, \grave{}current - 1); \\ \mathsf{var}\, j. \\ \quad j' := \grave{}current; \\ \quad \mathsf{label}\, D. \mathsf{if}\ \grave{}j = \grave{}length\ \mathsf{then\ skip} \\ \qquad \mathsf{else}\, (j', line') := (\grave{}j + 1, \grave{}line \dagger (\grave{}j, \grave{}line@(\grave{}j + 1))); \\ \qquad\quad D(\grave{}\sigma, \sigma') \end{array} \right) \end{array} \right);$$
$$error' := \mathsf{F}$$

With the exception of the relationship between the abstract variables, this window is also completely implemented.

Once we have closed enough windows to bring the quantification of the abstract variables back into view, we will be able to see that they are no longer related to the concrete variables of the implementation. When this happens we will be able to drop the abstract variables from the specification. We therefore proceed by closing this window.

Window: 5

$$\sqsupseteq_3 \star \mathsf{if}\ \grave{}current = 0\ \mathsf{then}\ \lfloor l_2 = l_1 \rfloor \underline{\triangle} (error' := \mathsf{T})$$
$$\mathsf{else}\, (\lfloor l_2 = del\ l_1 \rfloor \underline{\triangle} [As\ in\ previous\ window]); error' := \mathsf{F}$$

Similarly, this window should also be closed.

Window: 4

$$\sqsupseteq_3 \star \mathsf{if}\ \grave{}key = \boxed{\mathsf{delete}}\,\mathsf{then}$$
$$\mathsf{if}\ \grave{}current = 0\ \mathsf{then}\ \lfloor l_2 = l_1 \rfloor \underline{\triangle} (error' := \mathsf{T})$$
$$\mathsf{else}\, (\lfloor l_2 = del\ l_1 \rfloor \underline{\triangle} [As\ in\ previous\ window]); error' := \mathsf{F}$$
$$\mathsf{else} \ldots$$

We close this window too.

Window: 3

$$\vdots$$
$$\sqsupseteq_3 \star \mathsf{if}\ \grave{}key = \boxed{\mathsf{delete}}\,\mathsf{then}$$
$$\mathsf{if}\ \grave{}current = 0\ \mathsf{then}\ \lfloor l_2 = l_1 \rfloor \underline{\triangle} (error' := \mathsf{T})$$
$$\mathsf{else}\, (\lfloor l_2 = del\ l_1 \rfloor \underline{\triangle} [As\ in\ previous\ window]); error' := \mathsf{F}$$
$$\mathsf{else} \ldots;$$
$$\mathsf{if}\ 0 \leq \grave{}current - \grave{}offset \leq MaxPos\ \mathsf{then\ skip}$$
$$\mathsf{else\ if}\ \grave{}current < \grave{}offset\ \mathsf{then}\ offset' := \grave{}current$$
$$\mathsf{else}\ offset' := \grave{}current - MaxPos;$$
$$PAINT(\grave{}line, \grave{}length, \grave{}current, line', length', current',$$
$$\grave{}error, error', \grave{}offset, screen', cursor', offset')$$

We also close this window.

Window: 2

$$
\begin{array}{l}
\vdots \\
\sqsupseteq_3 \star \mathit{GoodScreen}(\grave{}\mathit{screen}, \grave{}\mathit{cursor}, \grave{}\mathit{offset}, \grave{}\mathit{line}, \grave{}\mathit{length}, \grave{}\mathit{current}) \supset \\
\quad \left(\begin{array}{l}
\mathsf{if}\ \grave{}\mathit{key} = \boxed{\mathsf{delete}}\ \mathsf{then} \\
\quad \mathsf{if}\ \grave{}\mathit{current} = 0\ \mathsf{then}\ \lfloor l_2 = l_1 \rfloor \mathbin{\underline{\triangle}} (\mathit{error}' := \mathsf{T}) \\
\quad \mathsf{else}\ (\lfloor l_2 = \mathit{del}\ l_1 \rfloor \mathbin{\underline{\triangle}} [\mathit{As\ in\ previous\ window}]); \mathit{error}' := \mathsf{F} \\
\mathsf{else} \ldots ; [\mathit{As\ in\ previous\ window}]
\end{array}\right)
\end{array}
$$

The requirements of this window state that if the screen is initially a view of the line, then we should edit the line. Otherwise, no particular behaviour is required. This requirement can be easily satisfied by editing the line regardless of whether the screen is initially a view of the line or not.

$$
\begin{array}{l}
\sqsupseteq_3 \star \mathsf{if}\ \grave{}\mathit{key} = \boxed{\mathsf{delete}}\ \mathsf{then} \\
\quad \mathsf{if}\ \grave{}\mathit{current} = 0\ \mathsf{then}\ \lfloor l_2 = l_1 \rfloor \mathbin{\underline{\triangle}} (\mathit{error}' := \mathsf{T}) \\
\quad \mathsf{else}\ (\lfloor l_2 = \mathit{del}\ l_1 \rfloor \mathbin{\underline{\triangle}} [\mathit{As\ in\ previous\ window}]); \mathit{error}' := \mathsf{F} \\
\mathsf{else} \ldots ; [\mathit{As\ in\ previous\ window}]
\end{array}
$$

We may now close this window.

Window: 1

$$
\begin{array}{l}
\sqsupseteq_3 \star (\exists l_1 .\ \mathit{IsAbs}(l_1, \grave{}\mathit{line}, \grave{}\mathit{length}, \grave{}\mathit{current})) \supset \\
\quad \left(\begin{array}{l}
\mathsf{let}\ l_1 = (\varepsilon\, l_1 .\ \mathit{IsAbs}(l_1, \grave{}\mathit{line}, \grave{}\mathit{length}, \grave{}\mathit{current}))\ \mathsf{in} \\
\quad \underline{\exists} l_2 .\ \mathsf{if}\ \grave{}\mathit{key} = \boxed{\mathsf{delete}}\ \mathsf{then} \\
\qquad \mathsf{if}\ \grave{}\mathit{current} = 0\ \mathsf{then}\ \lfloor l_2 = l_1 \rfloor \mathbin{\underline{\triangle}} (\mathit{error}' := \mathsf{T}) \\
\qquad \mathsf{else}\ (\lfloor l_2 = \mathit{del}\ l_1 \rfloor \mathbin{\underline{\triangle}} [\mathit{As\ in\ previous\ window}]); \mathit{error}' := \mathsf{F} \\
\quad \mathsf{else} \ldots ; [\mathit{As\ in\ previous\ window}]
\end{array}\right)
\end{array}
$$

The focus of this window requires that we edit the line if there exists an abstract line corresponding to the initial concrete state of the editor. We can refine these requirements by editing the line regardless of whether there exists an abstract line corresponding to the initial concrete state of the editor.

$$
\begin{array}{l}
\sqsupseteq_3 \star \mathsf{let}\ l_1 = (\varepsilon\, l_1 .\ \mathit{IsAbs}(l_1, \grave{}\mathit{line}, \grave{}\mathit{length}, \grave{}\mathit{current}))\ \mathsf{in} \\
\quad \underline{\exists} l_2 .\ \mathsf{if}\ \grave{}\mathit{key} = \boxed{\mathsf{delete}}\ \mathsf{then} \\
\qquad \mathsf{if}\ \grave{}\mathit{current} = 0\ \mathsf{then}\ \lfloor l_2 = l_1 \rfloor \mathbin{\underline{\triangle}} (\mathit{error}' := \mathsf{T}) \\
\qquad \mathsf{else}\ (\lfloor l_2 = \mathit{del}\ l_1 \rfloor \mathbin{\underline{\triangle}} [\mathit{As\ in\ previous\ window}]); \mathit{error}' := \mathsf{F} \\
\quad \mathsf{else} \ldots ; [\mathit{As\ in\ previous\ window}]
\end{array}
$$

The abstract variable l_2 occurs only at two places in the focus of the current window. (If we had developed the other editing operations, there would be further similar occurrences of l_2 in each of their implementations, currently represented with an

ellipsis.) Each of the remaining occurrences of l_2 is in a different branch of a conditional statement. We can move the scope of the quantification of l_2 inwards, splitting it into two quantifications (one for each branch of the conditional), until the resulting quantifications just enclose each instance of l_2.

$$
\begin{array}{l}
\sqsupseteq_3 \star \mathsf{let}\ l_1 = (\mathcal{E}\, l_1.\, \mathit{IsAbs}(l_1, \grave{}\mathit{line}, \grave{}\mathit{length}, \grave{}\mathit{current}))\ \mathsf{in} \\
\quad \mathsf{if}\ \grave{}\mathit{key} = \boxed{\mathsf{delete}}\ \mathsf{then} \\
\quad\quad \mathsf{if}\ \grave{}\mathit{current} = 0\ \mathsf{then}\ (\underline{\exists} l_2.\, \lfloor l_2 = l_1 \rfloor) \mathbin{\underline{\wedge}} (\mathit{error}' := \mathsf{T}) \\
\quad\quad \mathsf{else}\ ((\underline{\exists} l_2.\, \lfloor l_2 = \mathit{del}\ l_1 \rfloor) \mathbin{\underline{\wedge}} \mathit{[As\ in\ previous\ window]}); \mathit{error}' := \mathsf{F} \\
\quad \mathsf{else} \ldots; \mathit{[As\ in\ previous\ window]}
\end{array}
$$

Both of the terms in which l_2 is existentially quantified are now obviously equal to true (T_3). Since for any three-valued predicate P, $\mathsf{T}_3 \mathbin{\underline{\wedge}} P$ is equal to P, we can remove the quantifications of l_2 from the focus.

We have now removed all reference to the final abstract state of the editor from the implementation.

$$
\begin{array}{l}
\sqsupseteq_3 \star \mathsf{let}\ l_1 = (\mathcal{E}\, l_1.\, \mathit{IsAbs}(l_1, \grave{}\mathit{line}, \grave{}\mathit{length}, \grave{}\mathit{current}))\ \mathsf{in} \\
\quad \mathsf{if}\ \grave{}\mathit{key} = \boxed{\mathsf{delete}}\ \mathsf{then} \\
\quad\quad \mathsf{if}\ \grave{}\mathit{current} = 0\ \mathsf{then}\ \mathit{error}' := \mathsf{T} \\
\quad\quad \mathsf{else}\ \mathit{[As\ in\ previous\ window]}; \mathit{error}' := \mathsf{F} \\
\quad \mathsf{else} \ldots; \mathit{[As\ in\ previous\ window]}
\end{array}
$$

There is no longer any reference to the initial abstract state of the editor l_1 in the body of the let construction. We may therefore remove the let construct from the implementation. In doing so we have removed all traces of the abstract representation of the line from the implementation of the editor.

This completes the derivation of the *EDIT* operation, so we show the resulting focus in its entirety.

$$
\begin{array}{l}
\sqsupseteq_3 \star \mathsf{if}\ \grave{}\mathit{key} = \boxed{\mathsf{delete}}\ \mathsf{then} \\
\quad \mathsf{if}\ \grave{}\mathit{current} = 0\ \mathsf{then}\ \mathit{error}' := \mathsf{T} \\
\quad \mathsf{else}\ (\mathit{length}', \mathit{current}') := (\grave{}\mathit{length} - 1, \grave{}\mathit{current} - 1); \\
\quad\quad \mathsf{var}\, j.\, j' := \grave{}\mathit{current}; \\
\quad\quad\quad \mathsf{label}\ D. \\
\quad\quad\quad\quad \mathsf{if}\ \grave{}j = \grave{}\mathit{length}\ \mathsf{then\ skip} \\
\quad\quad\quad\quad \mathsf{else}\ (j', \mathit{line}') := (\grave{}j + 1, \grave{}\mathit{line} \dagger (\grave{}j, \grave{}\mathit{line}@(\grave{}j + 1))); \\
\quad\quad\quad\quad\quad D(\grave{}\sigma, \sigma'); \\
\quad\quad \mathit{error}' := \mathsf{F} \\
\mathsf{else} \ldots; \\
\mathsf{if}\ 0 \leq \grave{}\mathit{current} - \grave{}\mathit{offset} \leq \mathit{MaxPos}\ \mathsf{then\ skip} \\
\mathsf{else\ if}\ \grave{}\mathit{current} < \grave{}\mathit{offset}\ \mathsf{then}\ \mathit{offset}' := \grave{}\mathit{current} \\
\mathsf{else}\ \mathit{offset}' := \grave{}\mathit{current} - \mathit{MaxPos}; \\
\mathit{PAINT}(\grave{}\mathit{line}, \grave{}\mathit{length}, \grave{}\mathit{current}, \mathit{line}', \mathit{length}', \mathit{current}', \\
\qquad \grave{}\mathit{error}, \mathit{error}', \grave{}\mathit{offset}, \mathit{screen}', \mathit{cursor}', \mathit{offset}')
\end{array}
$$

We have now, with the exception of the implementation of *PAINT*, completed the refinement of the *EDIT* operation.

B.4 The Refinement of *PAINT*

In this section we implement the predicate *PAINT* defined on page 133 of section B.2. We begin by making *PAINT* the focus of a window preserving refinement. Note that the state vector σ contains the following program variables: *current*, *cursor*, *error*, *length*, *line*, *offset* and *screen*.

Window: 1

$$\sqsupseteq_3 \star PAINT(`line, `length, `current, line', length', current', \\ `error, error', `offset, screen', cursor', offset')$$

We replace *PAINT* by its definition.

$$\sqsupseteq_3 \star SeeCurrent(`offset, `current) \supset \boxed{\left|\begin{array}{ll} & (offset', line', length', current', error') = \\ & \quad (`offset, `line, `length, `current, `error) \\ \wedge & IsView(screen', `line, `length, `offset) \\ \wedge & CursorCurrent(cursor', `offset, `current) \end{array}\right|}$$

Assuming that the variable *offset* has been set so that the current position falls within the bounds of the screen, the definition of *PAINT* requires that the screen correctly displays a view of the line and the cursor. If the screen has not been properly positioned on the line, then the definition of *PAINT* makes no requirements of the system.

We open a subwindow on the focus where we may assume that the screen has been correctly positioned.

Window: 2

$$! SeeCurrent(`offset, `current) \\ \sqsupseteq_3 \star \left|\begin{array}{ll} & (offset', line', length', current', error') = \\ & \quad (`offset, `line, `length, `current, `error) \\ \wedge & IsView(screen', `line, `length, `offset) \\ \wedge & CursorCurrent(cursor', `offset, `current) \end{array}\right|$$

The focus of the window requires an implementation which ensures that the screen displays a view of the line and the cursor is positioned so as to indicate the current

position. We decompose this task serially into a specification that requires us to first position the cursor and then ensure that the screen displays a view of the line.

$$\sqsupseteq_3 \star \boxed{\left\lfloor \begin{array}{ll} & CursorCurrent(cursor', \grave{}offset, \grave{}current) \\ \wedge & (offset', line', length', current', error') = \\ & \quad (\grave{}offset, \grave{}line, \grave{}length, \grave{}current, \grave{}error) \end{array} \right\rfloor}; \left\lfloor \begin{array}{ll} & cursor' = \grave{}cursor \\ \wedge & (offset', line', length', current', error') = \\ & \quad (\grave{}offset, \grave{}line, \grave{}length, \grave{}current, \grave{}error) \\ \wedge & IsView(screen', \grave{}line, \grave{}length, \grave{}offset) \end{array} \right\rfloor$$

We open a window on the first clause of the sequential composition and concentrate on the task of positioning the cursor.

Window: 3

$$\begin{array}{l} \vdots \\ ! \; SeeCurrent(\grave{}offset, \grave{}current) \\ \sqsupseteq_3 \star \left\lfloor \begin{array}{ll} & CursorCurrent(cursor', \grave{}offset, \grave{}current) \\ \wedge & (offset', line', length', current', error') = \\ & \quad (\grave{}offset, \grave{}line, \grave{}length, \grave{}current, \grave{}error) \end{array} \right\rfloor \end{array}$$

The focus of the current window requires us to position the cursor according to the predicate *CursorCurrent*. We may do this under the assumption *SeeCurrent*. In order to progress further we must first expand the definitions of *CursorCurrent* and *SeeCurrent* (thus creating a new lemma).

$$\begin{array}{l} ¡ \; 0 \leq \grave{}current - \grave{}offset \leq MaxPos \\ \sqsupseteq_3 \star \left\lfloor \begin{array}{ll} & cursor' = |\grave{}current - \grave{}offset| \\ \wedge & (offset', line', length', current', error') = \\ & \quad (\grave{}offset, \grave{}line, \grave{}length, \grave{}current, \grave{}error) \end{array} \right\rfloor \end{array}$$

We are required to place the cursor at the position $|\grave{}current - \grave{}offset|$. From the expanded definition of *SeeCurrent*, we can see that $0 \leq \grave{}current - \grave{}offset \leq MaxPos$. From this assumption we deduce that there is no need to take the absolute value of the expression $|\grave{}current - \grave{}offset|$.

$$\sqsupseteq_3 \star \left\lfloor \begin{array}{ll} & cursor' = \grave{}current - \grave{}offset \\ \wedge & (offset', line', length', current', error') = \\ & \quad (\grave{}offset, \grave{}line, \grave{}length, \grave{}current, \grave{}error) \end{array} \right\rfloor$$

The focus can now be easily implemented by assigning the value $\grave{}current - \grave{}offset$ to *cursor*.

$$\sqsupseteq_3 \star cursor' := \grave{}current - \grave{}offset$$

Having completed the implementation of this portion of the requirements, we close the window and return to the level above.

Window: 2

$$\vdots$$
$$\sqsupseteq_3 \star\ cursor' := \grave{}current - \grave{}offset;$$
$$\boxed{\left\lfloor \begin{array}{ll} & cursor' = \grave{}cursor \\ \wedge & (offset', line', length', current', error') = \\ & \quad (\grave{}offset, \grave{}line, \grave{}length, \grave{}current, \grave{}error) \\ \wedge & IsView(screen', \grave{}line, \grave{}length, \grave{}offset) \end{array} \right\rfloor}$$

We now open a subwindow on the second, and as yet unimplemented, clause of the sequential composition.

Window: 3

$$\vdots$$
$$\sqsupseteq_3 \star \left\lfloor \begin{array}{ll} & cursor' = \grave{}cursor \\ \wedge & (offset', line', length', current', error') = \\ & \quad (\grave{}offset, \grave{}line, \grave{}length, \grave{}current, \grave{}error) \\ \wedge & IsView(screen', \grave{}line, \grave{}length, \grave{}offset) \end{array} \right\rfloor$$

The focus of our window requires us to ensure that the screen is a view of the line at the current offset. We are also required not to alter the value of the variables other than *screen*.

First we make a minor simplification to the form of the requirement by coercing the two clauses requiring us to maintain the values of variables other than *screen* into a single clause.

$$\sqsupseteq_3 \star \left\lfloor \begin{array}{ll} & (offset', line', length', current', error', cursor') = \\ & \quad (\grave{}offset, \grave{}line, \grave{}length, \grave{}current, \grave{}error, \grave{}cursor) \\ \wedge & IsView(screen', \grave{}line, \grave{}length, \grave{}offset) \end{array} \right\rfloor$$

In order to see exactly what the focus of the current window requires, we expand the definition of *IsView*.

$$\sqsupseteq_3 \star \left\lfloor \begin{array}{ll} & (offset', line', length', current', error', cursor') = \\ & \quad (\grave{}offset, \grave{}line, \grave{}length, \grave{}current, \grave{}error, \grave{}cursor) \\ \wedge & \left(\forall n.\, (n \leq MaxPos) \supset \right. \\ & \quad \left. screen'@n = glyph\_at(\grave{}line, \grave{}length, n + \grave{}offset) \right) \end{array} \right\rfloor$$

We can now see that the requirements call for every position on the screen (from zero up to and including *MaxPos*) to contain a glyph representing the contents of the line

at the corresponding position. We will implement this requirement with a recursive block. First we introduce a local variable i. We will then increment i from zero up to $MaxPos + 1$ while ensuring all those positions on the screen below i correctly represent the corresponding positions of the line.

$$\sqsupseteq_3 \star \mathsf{var}\, i.$$

$$\boxed{\left\lfloor \begin{array}{ll} & (offset', line', length', current', error', cursor') = \\ & \quad (`offset, `line, `length, `current, `error, `cursor) \\ \wedge & \left(\forall n.\, (n \leq MaxPos) \supset \right. \\ & \qquad \left. screen'@n = glyph\_at(`line, `length, n + `offset)\right) \end{array} \right\rfloor}$$

We open a subwindow on the body of the block which introduced the variable i. Within the body of this block i is a member of the state, σ.

Window: 4

$$\vdots$$

$$\sqsupseteq_3 \star \left\lfloor \begin{array}{ll} & (offset', line', length', current', error', cursor') = \\ & \quad (`offset, `line, `length, `current, `error, `cursor) \\ \wedge & \left(\forall n.\, (n \leq MaxPos) \supset \right. \\ & \qquad \left. screen'@n = glyph\_at(`line, `length, n + `offset)\right) \end{array} \right\rfloor$$

In preparation for introducing a recursive implementation for the current focus, we define a predicate on states I_p, to be the invariant of the recursion.

$$\begin{array}{ll} \vdash^{\mathsf{def}} I_p\ \sigma = & \\ \qquad i \leq MaxPos + 1 & [I_p\_\mathsf{DEF}] \\ \quad \wedge \quad \left(\forall n.\, (n < i) \supset (screen@n = glyph\_at(line, length, n + offset))\right) & \end{array}$$

The invariant I_p holds whenever i is less than $MaxPos + 1$, and every position on the screen less than i contains a glyph representing the corresponding position of the line.

Before we can introduce the recursive implementation of the focus, we must first establish the invariant. Rather than choose any particular way to establish the invariant, we simply prefix the current requirement with a specification requiring that the invariant be established.

$$\sqsupseteq_3 \star \left\lfloor \begin{array}{ll} & I_p\ \sigma' \\ \wedge & (offset', line', length', current', error', cursor') = \\ & \quad (`offset, `line, `length, `current, `error, `cursor) \end{array} \right\rfloor ;$$

$$\boxed{\left\lfloor \begin{array}{ll} & (offset', line', length', current', error', cursor') = \\ & \quad (`offset, `line, `length, `current, `error, `cursor) \\ \wedge & \left(\forall n.\, (n \leq MaxPos) \supset \right. \\ & \qquad \left. screen'@n = glyph\_at(`line, `length, n + `offset)\right) \end{array} \right\rfloor}$$

We open a subwindow on our original requirement and proceed with its implementation. Later, after this subwindow is closed, we will return and implement the requirement to establish the invariant.

Window: 5

$$\begin{array}{l} \vdots \\ !\ I_p\ `\sigma \\ \sqsupseteq_3 \star \left\lfloor \begin{array}{ll} & (\mathit{offset}', \mathit{line}', \mathit{length}', \mathit{current}', \mathit{error}', \mathit{cursor}') = \\ & \quad (`\mathit{offset}, `\mathit{line}, `\mathit{length}, `\mathit{current}, `\mathit{error}, `\mathit{cursor}) \\ \wedge & \left(\begin{array}{l} \forall n.\, (n \leq \mathit{MaxPos}) \supset \\ \quad \mathit{screen}'@n = \mathit{glyph\_at}(`\mathit{line}, `\mathit{length}, n + `\mathit{offset}) \end{array}\right) \end{array} \right\rfloor \end{array}$$

In the context of this subwindow we may assume that the invariant holds. We may therefore introduce the recursive block.

Every recursive block must have a variant function which must be shown to decrease with respect to some well founded ordering upon each iteration. We use the $<$ ordering on the natural numbers. We now define the function v_p which is the variant for the recursive block.

$$\models^{\text{def}} v_p\ \sigma = |\mathit{MaxPos} + 1 - i| \qquad [v_p\text{\_DEF}]$$

The value of v_p gets smaller the closer i gets to its final value of $\mathit{MaxPos} + 1$.

During its development, a recursive block is annotated with the original specification it was introduced to implement. We therefore define the following three-valued predicate O_p as an abbreviation for the focus of the current window.

$$\begin{array}{l} \models^{\text{def}} O_p(\sigma_1, \sigma_2) = \\ \quad \left\lfloor \begin{array}{ll} & (\mathit{offset}_2, \mathit{line}_2, \mathit{length}_2, \mathit{current}_2, \mathit{error}_2, \mathit{cursor}_2) = \\ & \quad (\mathit{offset}_1, \mathit{line}_1, \mathit{length}_1, \mathit{current}_1, \mathit{error}_1, \mathit{cursor}_1) \\ \wedge & \left(\begin{array}{l} \forall n.\, (n \leq \mathit{MaxPos}) \supset \\ \quad \mathit{screen}'@n = \mathit{glyph\_at}(`\mathit{line}, `\mathit{length}, n + `\mathit{offset}) \end{array}\right) \end{array} \right\rfloor \end{array} \qquad [O_p\text{\_DEF}]$$

We now introduce the recursive block, choosing P (for 'paint') as the label for the block.

$$\begin{array}{l} \vdots \\ !\ I_p\ `\sigma \\ \sqsupseteq_3 \star \mathsf{label}_{(v_p, O_p, I_p)}\ P. \end{array}$$

$$\boxed{\left\lfloor \begin{array}{ll} & (\mathit{offset}', \mathit{line}', \mathit{length}', \mathit{current}', \mathit{error}', \mathit{cursor}') = \\ & \quad (`\mathit{offset}, `\mathit{line}, `\mathit{length}, `\mathit{current}, `\mathit{error}, `\mathit{cursor}) \\ \wedge & \left(\begin{array}{l} \forall n.\, (n \leq \mathit{MaxPos}) \supset \\ \quad \mathit{screen}'@n = \mathit{glyph\_at}(`\mathit{line}, `\mathit{length}, n + `\mathit{offset}) \end{array}\right) \end{array} \right\rfloor}$$

To continue the refinement we open a window inside the body of the labelled block.

Window: 6

$$
\begin{array}{l}
\vdots \\
! \; I_p \; `\sigma \\
! \; \forall \sigma_1 \, \sigma_2 . \, I_p \, \sigma_1 \wedge (v_p \, \sigma_1 < v_p \; `\sigma) \supset (P(\sigma_1, \sigma_2) \sqsupseteq_3 O_p(\sigma_1, \sigma_2)) \\
\sqsupseteq_3 \star \left\lfloor \begin{array}{ll} & (offset', line', length', current', error', cursor') = \\ & \quad (`offset, `line, `length, `current, `error, `cursor) \\ \wedge & \left(\begin{array}{l} \forall n . (n \leq MaxPos) \supset \\ \quad screen'@n = glyph\_at(`line, `length, n + `offset) \end{array} \right) \end{array} \right\rfloor
\end{array}
$$

Within the body of the block we may assume that the invariant holds. We may also assume that in any state where the invariant holds and the variant has decreased, a recursive call to the label P implements the original specification.

To see exactly what we may assume, we expand the definition of the invariant in the assumptions. The result of the expansion contains two conjuncts, which we split into two separate lemmas.

$$
\begin{array}{l}
¡ \; `i \leq MaxPos + 1 \\
¡ \; \forall n . (n < `i) \supset (`screen@n = glyph\_at(`line, `length, n + `offset)) \\
\sqsupseteq_3 \star \left\lfloor \begin{array}{ll} & (offset', line', length', current', error', cursor') = \\ & \quad (`offset, `line, `length, `current, `error, `cursor) \\ \wedge & \left(\begin{array}{l} \forall n . (n \leq MaxPos) \supset \\ \quad screen'@n = glyph\_at(`line, `length, n + `offset) \end{array} \right) \end{array} \right\rfloor
\end{array}
$$

It is now clear from the context that if $`i$ is equal to $MaxPos + 1$, then our goal has been achieved. We therefore introduce a conditional that allows us to consider this case separately.

$$
\begin{array}{l}
\sqsupseteq_3 \star \, \mathsf{if} \; `i = MaxPos + 1 \; \mathsf{then} \\
\qquad \boxed{\left\lfloor \begin{array}{ll} & (offset', line', length', current', error', cursor') = \\ & \quad (`offset, `line, `length, `current, `error, `cursor) \\ \wedge & \left(\begin{array}{l} \forall n . (n \leq MaxPos) \supset \\ \quad screen'@n = glyph\_at(`line, `length, n + `offset) \end{array} \right) \end{array} \right\rfloor} \\
\quad \mathsf{else} \\
\qquad \left\lfloor \begin{array}{ll} & (offset', line', length', current', error', cursor') = \\ & \quad (`offset, `line, `length, `current, `error, `cursor) \\ \wedge & \left(\begin{array}{l} \forall n . (n \leq MaxPos) \supset \\ \quad screen'@n = glyph\_at(`line, `length, n + `offset) \end{array} \right) \end{array} \right\rfloor
\end{array}
$$

First we open a window on the simple case and find an implementation for it.

Window: 7

$$
\begin{array}{l}
\vdots \\
\mathsf{i}\ \forall n.(n < \grave{}i) \supset (\grave{}screen@n = glyph\_at(\grave{}line, \grave{}length, n + \grave{}offset)) \\
!\ \grave{}i = MaxPos + 1 \\
\sqsupseteq_3 \star \left\lfloor \begin{array}{ll} & (offset', line', length', current', error', cursor') = \\ & \quad (\grave{}offset, \grave{}line, \grave{}length, \grave{}current, \grave{}error, \grave{}cursor) \\ \wedge & \left(\begin{array}{l} \forall n.(n \leq MaxPos) \supset \\ \quad screen'@n = glyph\_at(\grave{}line, \grave{}length, n + \grave{}offset) \end{array} \right) \end{array} \right\rfloor
\end{array}
$$

Here, in the context of the first branch of the conditional, we may assume that $\grave{}i$ is equal to $MaxPos + 1$. Along with the other assumptions, this fact allows us to deduce that the conditions that we wish the final state of this operation to meet are already satisfied in the initial state. Therefore, simply terminating in the same state as the initial state will satisfy the requirements Accordingly, the whole operation can be implemented with a skip statement.

$$\sqsupseteq_3 \star \mathsf{skip}$$

Having completed the implementation of this window, we close it and return to the parent window.

Window: 6

$$
\begin{array}{l}
\vdots \\
!\ \forall \sigma_1\, \sigma_2.\, I_p\ \sigma_1 \wedge (v_p\ \sigma_1 < v_p\ \grave{}\sigma) \supset (P(\sigma_1, \sigma_2) \sqsupseteq_3\ O_p(\sigma_1, \sigma_2)) \\
\mathsf{i}\ \grave{}i \leq MaxPos + 1 \\
\mathsf{i}\ \forall n.(n < \grave{}i) \supset (\grave{}screen@n = glyph\_at(\grave{}line, \grave{}length, n + \grave{}offset)) \\
\sqsupseteq_3 \star \mathsf{if}\ \grave{}i = MaxPos + 1\ \mathsf{then}\ \mathsf{skip} \\
\quad \mathsf{else} \\
\qquad \boxed{\left\lfloor \begin{array}{ll} & (offset', line', length', current', error', cursor') = \\ & \quad (\grave{}offset, \grave{}line, \grave{}length, \grave{}current, \grave{}error, \grave{}cursor) \\ \wedge & \left(\begin{array}{l} \forall n.(n \leq MaxPos) \supset \\ \quad screen'@n = glyph\_at(\grave{}line, \grave{}length, n + \grave{}offset) \end{array} \right) \end{array} \right\rfloor}
\end{array}
$$

We now open a subwindow on the second clause of the conditional and turn our attention to the difficult case, where $\grave{}i$ does not yet equal $MaxPos + 1$.

Window: 7

$$
\begin{array}{l}
\vdots \\
! \ \forall \sigma_1 \, \sigma_2 . \, I_p \, \sigma_1 \wedge (v_p \, \sigma_1 < v_p \, \grave{} \sigma) \supset (P(\sigma_1, \sigma_2) \sqsupseteq_3 O_p(\sigma_1, \sigma_2)) \\
¡ \ \grave{} i \leq MaxPos + 1 \\
¡ \ \forall n . (n < \grave{} i) \supset (\grave{} screen@n = glyph\_at(\grave{} line, \grave{} length, n + \grave{} offset)) \\
! \ \grave{} i \neq MaxPos + 1 \\
\sqsupseteq_3 \star \left\lfloor \begin{array}{ll} & (offset', line', length', current', error', cursor') = \\ & \quad (\grave{} offset, \grave{} line, \grave{} length, \grave{} current, \grave{} error, \grave{} cursor) \\ \wedge & \left(\begin{array}{l} \forall n . (n \leq MaxPos) \supset \\ \quad screen'@n = glyph\_at(\grave{} line, \grave{} length, n + \grave{} offset) \end{array} \right) \end{array} \right\rfloor
\end{array}
$$

To implement the focus of this window we must first decrease that variant while maintaining the invariant and then complete the implementation with a recursive call to the label. Rather than decide right away how to achieve this, we simply prefix the current focus with a requirement to decrease the variant while maintaining the invariant.

$$
\sqsupseteq_3 \star \left\lfloor \begin{array}{ll} & I_p \, \sigma' \wedge (v_p \, \sigma' < v_p \, \sigma') \\ \wedge & (offset', line', length', current', error', cursor') = \\ & \quad (\grave{} offset, \grave{} line, \grave{} length, \grave{} current, \grave{} error, \grave{} cursor) \end{array} \right\rfloor ;
$$

$$
\boxed{\left\lfloor \begin{array}{ll} & (offset', line', length', current', error', cursor') = \\ & \quad (\grave{} offset, \grave{} line, \grave{} length, \grave{} current, \grave{} error, \grave{} cursor) \\ \wedge & \left(\begin{array}{l} \forall n . (n \leq MaxPos) \supset \\ \quad screen'@n = glyph\_at(\grave{} line, \grave{} length, n + \grave{} offset) \end{array} \right) \end{array} \right\rfloor}
$$

We open a window on the second clause of the composition.

Window: 8

$$
\begin{array}{l}
\vdots \\
! \ \boxed{\forall \sigma_1 \, \sigma_2 . \, I_p \, \sigma_1 \wedge (v_p \, \sigma_1 < v_p \, \tilde{} \sigma) \supset (P(\sigma_1, \sigma_2) \sqsupseteq_3 O_p(\sigma_1, \sigma_2))} \\
! \ I_p \, \grave{} \sigma \\
! \ v_p \, \grave{} \sigma < v_p \, \tilde{} \sigma \\
\sqsupseteq_3 \star \left\lfloor \begin{array}{ll} & (offset', line', length', current', error', cursor') = \\ & \quad (\grave{} offset, \grave{} line, \grave{} length, \grave{} current, \grave{} error, \grave{} cursor) \\ \wedge & \left(\begin{array}{l} \forall n . (n \leq MaxPos) \supset \\ \quad screen'@n = glyph\_at(\grave{} line, \grave{} length, n + \grave{} offset) \end{array} \right) \end{array} \right\rfloor
\end{array}
$$

The current focus is an instance of the definition of O_p on the current initial and final states.

$$
\sqsupseteq_3 \star O_p(\grave{} \sigma, \sigma')
$$

We now open a subwindow on the assumption that describes under what conditions P can be used to implement O_p.

Window: 9

$$
\begin{array}{l}
\vdots \\
!\ I_p\ `\sigma \\
!\ v_p\ `\sigma < v_p\ ^{\backprime}\sigma \\
\subset\star\ \forall \sigma_1\ \sigma_2.\ I_p\ \sigma_1 \wedge (v_p\ \sigma_1 < v_p\ ^{\backprime}\sigma) \supset (P(\sigma_1, \sigma_2) \sqsupseteq_3 O_p(\sigma_1, \sigma_2))
\end{array}
$$

We instantiate the universally quantified states σ_1 and σ_2 to be the current initial and final states.

$$\subset\star\ I_p\ `\sigma \wedge (v_p\ `\sigma < v_p\ ^{\backprime}\sigma) \supset (P(\sigma_1, \sigma_2) \sqsupseteq_3 O_p(\sigma_1, \sigma_2))$$

Using the contextual assumptions we can then simplify the focus as follows.

$$\subset\star\ P(`\sigma, \sigma') \sqsupseteq_3 O_p(`\sigma, \sigma')$$

We have now know that in the current state we can implement the original specification with a recursive call to the label. We close the window and return to the parent where we can make use of this new fact.

Window: 8

$$
\begin{array}{l}
\vdots \\
¡\ P(`\sigma, \sigma') \sqsupseteq_3 O_p(`\sigma, \sigma') \\
\sqsupseteq_3\star\ O_p(`\sigma, \sigma')
\end{array}
$$

Using our newly derived lemma, we refine the focus from $O_p(`\sigma, \sigma')$ to $P(`\sigma, \sigma')$.

$$\sqsupseteq_3\star\ P(`\sigma, \sigma')$$

The focus of this window has now been implemented. We close the window and return to its parent.

Window: 7

$$
\begin{array}{l}
\vdots \\
¡\ `i \leq MaxPos + 1 \\
¡\ \forall n.\ (n < `i) \supset (`screen@n = glyph\_at(`line, `length, n + `offset)) \\
!\ `i \neq MaxPos + 1 \\
\sqsupseteq_3\star \boxed{\left\lfloor \begin{array}{ll} & I_p\ \sigma' \wedge (v_p\ \sigma' < v_p\ \sigma') \\ \wedge & (offset', line', length', current', error', cursor') = \\ & \quad (`offset, `line, `length, `current, `error, `cursor) \end{array} \right\rfloor}; \\
P(`\sigma, \sigma')
\end{array}
$$

In this window the clause that is required to decrease the variant while maintaining the invariant remains unimplemented. We open a subwindow on the unimplemented portion of the focus.

Window: 8

$$
\begin{array}{l}
\vdots \\
\mathbin{¡}\ \grave{}i \leq MaxPos + 1 \\
\mathbin{¡}\ \forall n.(n < \grave{}i) \supset (\grave{}screen@n = glyph\_at(\grave{}line, \grave{}length, n + \grave{}offset)) \\
!\ \grave{}i \neq MaxPos + 1 \\
\sqsupseteq_3 \star \left\lfloor \begin{array}{ll} & I_p\ \sigma' \wedge (v_p\ \sigma' < v_p\ \sigma') \\ \wedge & (offset', line', length', current', error', cursor') = \\ & \quad (\grave{}offset, \grave{}line, \grave{}length, \grave{}current, \grave{}error, \grave{}cursor) \end{array} \right\rfloor
\end{array}
$$

To see exactly what is required of this focus, we expand the definitions of the I_p and v_p.

$$
\sqsupseteq_3 \star \left\lfloor \begin{array}{ll} & i' \leq MaxPos + 1 \\ \wedge & \left(\begin{array}{l} \forall n.(n < i') \supset \\ \quad screen'@n = glyph\_at(line', length', n + offset') \end{array} \right) \\ \wedge & \boxed{|MaxPos + 1 - i'| < |MaxPos + 1 - \grave{}i|} \\ \wedge & (offset', line', length', current', error', cursor') = \\ & \quad (\grave{}offset, \grave{}line, \grave{}length, \grave{}current, \grave{}error, \grave{}cursor) \end{array} \right\rfloor
$$

We open a subwindow to simplify the clause that states that the difference between i and $MaxPos + 1$ should decrease.

Window: 9

$$
\begin{array}{l}
\vdots \\
!\ \grave{}i \leq MaxPos + 1 \\
!\ i' \leq MaxPos + 1 \\
\supset \star\ |MaxPos + 1 - i'| < |MaxPos + 1 - \grave{}i|
\end{array}
$$

In this context we can assume that i is less or equal to $MaxPos + 1$ both before and after the operation. This allows us to simplify the focus as follows.

$$\supset \star\ MaxPos + 1 - i' < MaxPos + 1 - \grave{}i$$

A few further algebraic simplifications allow us to reduce the focus to a requirement that i should increase.

$$
\begin{array}{l}
\supset \star\ -i' < -\grave{}i \\
\supset \star\ \grave{}i < i'
\end{array}
$$

We now close the window.

Window: 8

$$
\begin{array}{l}
\vdots \\
\mathbin{¡}\ `i \leq MaxPos + 1 \\
\mathbin{¡}\ \forall n.(n < `i) \supset (`screen@n = glyph\_at(`line, `length, n + `offset)) \\
!\ `i \neq MaxPos + 1 \\
\sqsupseteq_3 \star \left\lfloor \begin{array}{ll}
 & i' \leq MaxPos + 1 \\
\wedge & \boxed{\left(\forall n.(n < i') \supset \\ \qquad screen'@n = glyph\_at(line', length', n + offset')\right)} \\
\wedge & `i < i' \\
\wedge & (offset', line', length', current', error', cursor') = \\
 & \qquad (`offset, `line, `length, `current, `error, `cursor)
\end{array} \right\rfloor
\end{array}
$$

We now open a subwindow to simplify the requirement that after the operation every position on the screen less than i should contain a glyph representing the contents of the corresponding position of the line.

Window: 9

$$
\begin{array}{l}
\vdots \\
\mathbin{¡}\ \forall n.(n < `i) \supset (`screen@n = glyph\_at(`line, `length, n + `offset)) \\
!\ `i < i' \\
!\ (offset', line', length', \ldots) = (`offset, `line, `length, \ldots) \\
\supset \star \forall n.(n < i') \supset (screen'@n = glyph\_at(line', length', n + offset'))
\end{array}
$$

We simplify the focus using the assumption the operation must not alter the values of *offset*, *line* or *length*.

$$
\supset \star \forall n.(n < i') \supset (screen'@n = glyph\_at(`line, `length, n + `offset))
$$

The focus of the window states that for all n less than i', the screen at position n must reflect the contents of the line at the corresponding position. However, we can see from the assumptions that this is already the case for all n less that $`i$. Since we also know that $`i$ is less than i', we can partition the universal quantification into two quantifications—one ranging from zero up to $`i$ and the other from $`i$ to i'.

$$
\begin{array}{ll}
\supset \star & \left(\forall n.(n < `i) \supset \right. \\
 & \qquad \left. screen'@n = \boxed{glyph\_at(`line, `length, n + `offset)}\right) \\
\wedge & \left(\forall n.(`i \leq n < i') \supset \right. \\
 & \qquad \left. screen'@n = glyph\_at(`line, `length, n + `offset)\right)
\end{array}
$$

We open a subwindow inside the first quantification.

Window: 10

$$
\begin{array}{l}
\quad \vdots \\
\quad ¡\ \forall n.(n < \grave{}i) \supset (\grave{}screen@n = glyph\_at(\grave{}line, \grave{}length, n + \grave{}offset)) \\
\quad !\ n < \grave{}i \\
= \star\ glyph\_at(\grave{}line, \grave{}length, n + \grave{}offset)
\end{array}
$$

Using the assumptions we can simplify the focus as follows.

$$= \star\ \grave{}screen@n$$

We now close the window.

Window: 9

$$
\begin{array}{ll}
\quad \vdots & \\
\supset \star & (\forall n.(n < \grave{}i) \supset screen'@n = \grave{}screen@n) \\
\quad \wedge & \left(\begin{array}{l} \forall n.(\grave{}i \le n < i') \supset \\ \qquad screen'@n = glyph\_at(\grave{}line, \grave{}length, n + \grave{}offset) \end{array}\right)
\end{array}
$$

We have now simplified the requirement that the screen (below position i') must reflect the contents of the line by identifying that portion of the screen that already reflects the contents of the line. We now close this window.

Window: 8

$$
\begin{array}{l}
\quad \vdots \\
\quad !\ \grave{}i \le MaxPos + 1 \\
\quad !\ \grave{}i \ne MaxPos + 1 \\
\sqsupseteq_3 \star \left|\begin{array}{ll}
 & i' \le MaxPos + 1 \\
\wedge & (\forall n.(n < \grave{}i) \supset screen'@n = \grave{}screen@n) \\
\wedge & \left(\begin{array}{l} \forall n.(\grave{}i \le n < i') \supset \\ \qquad screen'@n = glyph\_at(\grave{}line, \grave{}length, n + \grave{}offset) \end{array}\right) \\
\wedge & \grave{}i < i' \\
\wedge & (offset', line', length', current', error', cursor') = \\
 & \quad (\grave{}offset, \grave{}line, \grave{}length, \grave{}current, \grave{}error, \grave{}cursor)
\end{array}\right|
\end{array}
$$

The focus of this window requires an implementation that increases the value of i (while keeping it below $MaxPos + 1$). We have not yet decided by how much we will

increase the value of i. We now decide that the value of i should be increased by one. We add this requirement to the others.

$$\sqsupseteq_3 \star \left\lfloor \begin{array}{l} i' = \grave{}i + 1 \wedge \\ \boxed{\begin{array}{ll} & i' \leq MaxPos + 1 \\ \wedge & (\forall n.\, (n < \grave{}i) \supset screen'@n = \grave{}screen@n) \\ \wedge & \left(\forall n.\, (\grave{}i \leq n < i') \supset \right. \\ & \qquad \left. screen'@n = glyph\_at(\grave{}line, \grave{}length, n + \grave{}offset)\right) \\ \wedge & \grave{}i < i' \end{array}} \\ \wedge \quad (offset', line', length', current', error', cursor') = \\ \qquad (\grave{}offset, \grave{}line, \grave{}length, \grave{}current, \grave{}error, \grave{}cursor) \end{array} \right\rfloor$$

We open a subwindow to re-examine the earlier requirements now that we have decided that i should be increased by one.

Window: 9

$$\begin{array}{ll} & \vdots \\ & ! \ \grave{}i \leq MaxPos + 1 \\ & ! \ \grave{}i \neq MaxPos + 1 \\ & ! \ i' = \grave{}i + 1 \\ \supset \star & \quad i' \leq MaxPos + 1 \\ & \wedge \quad (\forall n.\, (n < \grave{}i) \supset screen'@n = \grave{}screen@n) \\ & \wedge \quad \left(\forall n.\, (\grave{}i \leq n < i') \supset \right. \\ & \qquad \left. screen'@n = glyph\_at(\grave{}line, \grave{}length, n + \grave{}offset)\right) \\ & \wedge \quad \grave{}i < i' \end{array}$$

We rewrite the focus using the assumption that $i' = \grave{}i + 1$.

$$\begin{array}{ll} \supset \star & \quad \grave{}i + 1 \leq MaxPos + 1 \\ & \wedge \quad (\forall n.\, (n < \grave{}i) \supset screen'@n = \grave{}screen@n) \\ & \wedge \quad \left(\forall n.\, (\grave{}i \leq n < \grave{}i + 1) \supset \right. \\ & \qquad \left. screen'@n = glyph\_at(\grave{}line, \grave{}length, n + \grave{}offset)\right) \\ & \wedge \quad \grave{}i < \grave{}i + 1 \end{array}$$

In this context we can see that the requirements that $\grave{}i + 1 \leq MaxPos + 1$ and $\grave{}i < \grave{}i + 1$ are both true.

$$\begin{array}{ll} \supset \star & \quad (\forall n.\, (n < \grave{}i) \supset screen'@n = \grave{}screen@n) \\ & \wedge \quad \left(\forall n.\, (\grave{}i \leq n < \grave{}i + 1) \supset \right. \\ & \qquad \left. screen'@n = glyph\_at\ (\grave{}line, \grave{}length, n + \grave{}offset)\right) \end{array}$$

The quantified variable in the second clause of the focus is restricted so that it may take on only a single value—$`i$. We may therefore simplify the focus by re-expressing that clause solely in terms of i.

$$\begin{array}{ll} \supset \star & (\forall n.(n < `i) \supset screen'@n = `screen@n) \\ \quad \wedge & screen'@`i = glyph\_at(`line, `length, `i + `offset) \end{array}$$

We have now finished simplifying the requirements, so we close this window and return to its parent.

Window: 8

$$\begin{array}{l} \vdots \\ ! \ `i \leq MaxPos + 1 \\ ! \ `i \neq MaxPos + 1 \\ \sqsupseteq_3 \star \left| \begin{array}{ll} & i' = `i + 1 \\ \wedge & (\forall n.(n < `i) \supset screen'@n = `screen@n) \\ \wedge & screen'@`i = glyph\_at(`line, `length, `i + `offset) \\ \wedge & (offset', line', length', current', error', cursor') = \\ & \quad (`offset, `line, `length, `current, `error, `cursor) \end{array} \right| \end{array}$$

At this point we could sequentially decompose the focus into a requirement to first update the screen at i, and to *then* increment i. However, it seems likely that these two operations could be accomplished simultaneously with a parallel assignment.

We therefore proceed to expand the definition of *glyph_at* to see what, precisely, we are required to place on the screen at position i.

$$\sqsupseteq_3 \star \left| \begin{array}{ll} & i' = `i + 1 \\ \wedge & (\forall n.(n < `i) \supset screen'@n = `screen@n) \\ \wedge & screen'@`i = \left(\begin{array}{ll} `i + `offset < `length & \\ \quad \Rightarrow & char2glyph\ `line@(`i + `offset) \\ \blacksquare & \cdot \end{array} \right) \\ \wedge & (offset', line', length', current', error', cursor') = \\ & \quad (`offset, `line, `length, `current, `error, `cursor) \end{array} \right|$$

The definition of *glyph_at* is conditional on the value of $`i + `offset$. If $`i + `offset$ is less than $`length$, it returns the glyph representing the contents of the line at $`i + `offset$. Otherwise, it returns the empty glyph. We therefore introduce a condi-

tional statement with which we can consider these two cases separately.

$$\sqsupseteq_3 \star \textsf{if } \grave{}i + \grave{}offset < \grave{}length \textsf{ then}$$

$$\boxed{\left|\begin{array}{ll} & i' = \grave{}i + 1 \\ \wedge & (\forall n.\,(n < \grave{}i) \supset screen'@n = \grave{}screen@n) \\ \wedge & screen'@\grave{}i = \left(\begin{array}{l} \grave{}i + \grave{}offset < \grave{}length \\ \quad \Rightarrow \quad char2glyph\ \grave{}line@(\grave{}i + \grave{}offset) \\ \quad \mid \quad \cdot \end{array}\right) \\ \wedge & (offset', line', length', current', error', cursor') = \\ & \quad (\grave{}offset, \grave{}line, \grave{}length, \grave{}current, \grave{}error, \grave{}cursor) \end{array}\right|}$$

$$\textsf{else}$$

$$\left|\begin{array}{ll} & i' = \grave{}i + 1 \\ \wedge & (\forall n.\,(n < \grave{}i) \supset screen'@n = \grave{}screen@n) \\ \wedge & screen'@\grave{}i = \left(\begin{array}{l} \grave{}i + \grave{}offset < \grave{}length \\ \quad \Rightarrow \quad char2glyph\ \grave{}line@(\grave{}i + \grave{}offset) \\ \quad \mid \quad \cdot \end{array}\right) \\ \wedge & (offset', line', length', current', error', cursor') = \\ & \quad (\grave{}offset, \grave{}line, \grave{}length, \grave{}current, \grave{}error, \grave{}cursor) \end{array}\right|$$

We open a window on the first case of the conditional. Using the resulting context we will be able to simplify the requirements on the contents of the screen at $\grave{}i$.

Window: 9

$$\begin{array}{l} \vdots \\ !\ \grave{}i \leq MaxPos + 1 \\ !\ \grave{}i \neq MaxPos + 1 \\ !\ \grave{}i + \grave{}offset < \grave{}length \end{array}$$

$$\sqsupseteq_3 \star \left|\begin{array}{ll} & i' = \grave{}i + 1 \\ \wedge & (\forall n.\,(n < \grave{}i) \supset screen'@n = \grave{}screen@n) \\ \wedge & screen'@\grave{}i = \left(\begin{array}{l} \grave{}i + \grave{}offset < \grave{}length \\ \quad \Rightarrow \quad char2glyph\ \grave{}line@(\grave{}i + \grave{}offset) \\ \quad \mid \quad \cdot \end{array}\right) \\ \wedge & (offset', line', length', current', error', cursor') = \\ & \quad (\grave{}offset, \grave{}line, \grave{}length, \grave{}current, \grave{}error, \grave{}cursor) \end{array}\right|$$

Using our contextual assumption $\grave{}i + \grave{}offset < \grave{}length$, we can see that the guard of the conditional describing the required value of $screen'$ at $\grave{}i$ is equal to true. We may therefore simplify the focus as follows, by reducing the conditional to its first

clause.

$$\sqsupseteq_3 \star \left| \begin{array}{ll} & i' = \grave{}i + 1 \\ \wedge & (\forall n.\, (n < \grave{}i) \supset screen'@n = \grave{}screen@n) \\ \wedge & screen'@\grave{}i = char2glyph\ \grave{}line@(\grave{}i + \grave{}offset) \\ \wedge & (offset', line', length', current', error', cursor') = \\ & \quad (\grave{}offset, \grave{}line, \grave{}length, \grave{}current, \grave{}error, \grave{}cursor) \end{array} \right|$$

We can now see that the focus is refined by incrementing i and assigning $screen'$ at $\grave{}i$ the glyph representing the contents of $\grave{}line$ at $\grave{}i + \grave{}offset$ while leaving all other variables (including the other elements of $screen$) unchanged. (A formal proof of this step would require use of the theorems $\dagger@ii\_\mathsf{THM}$ and $\dagger@ij\_\mathsf{THM}$ from Appendix A about updating arrays.)

Accordingly, we implement these requirements with the following parallel assignment.

$$\sqsupseteq_3 \star (i', screen') := (\grave{}i + 1, \grave{}screen\dagger(\grave{}i, char2glyph\ \grave{}line@(\grave{}i + \grave{}offset)))$$

This branch of the conditional has now been now implemented. We close this window and return to the parent, where the other branch of the conditional remains to be implemented.

Window: 8

$$\begin{array}{l} \vdots \\ !\ \grave{}i \leq MaxPos + 1 \\ !\ \grave{}i \neq MaxPos + 1 \\ \sqsupseteq_3 \star \mathsf{if}\ \grave{}i + \grave{}offset < \grave{}length\ \mathsf{then} \\ \qquad (i', screen') := (\grave{}i + 1, \grave{}screen\dagger(\grave{}i, char2glyph\ \grave{}line@(\grave{}i + \grave{}offset))) \\ \quad \mathsf{else} \\ \qquad \left| \begin{array}{ll} & i' = \grave{}i + 1 \\ \wedge & (\forall n.\, (n < \grave{}i) \supset screen'@n = \grave{}screen@n) \\ \wedge & screen'@\grave{}i = \left(\begin{array}{ll} \grave{}i + \grave{}offset < \grave{}length & \\ \quad \Rightarrow & char2glyph\ \grave{}line@(\grave{}i + \grave{}offset) \\ \mathbf{|} & \cdot \end{array} \right) \\ \wedge & (offset', line', length', current', error', cursor') = \\ & \quad (\grave{}offset, \grave{}line, \grave{}length, \grave{}current, \grave{}error, \grave{}cursor) \end{array} \right| \end{array}$$

The refinement of the second branch of the conditional is virtually identical to that of the first. In this case the conditional expression may be reduced to its second case and the resulting requirements refined into another parallel assignment. We therefore omit the details of this refinement and present only the result.

$$\begin{array}{l} \sqsupseteq_3 \star \mathsf{if}\ \grave{}i + \grave{}offset < \grave{}length\ \mathsf{then} \\ \qquad (i', screen') := (\grave{}i + 1, \grave{}screen\dagger(\grave{}i, char2glyph\ \grave{}line@(\grave{}i + \grave{}offset))) \\ \quad \mathsf{else}\ (i', screen') := (\grave{}i + 1, \grave{}screen\dagger(\grave{}i, \cdot)) \end{array}$$

The focus of this window is completely implemented, so we close it.

Window: 7

$$
\begin{array}{l}
\quad\vdots \\
\sqsupseteq_3 \star \mathsf{if}\ \grave{}i + \grave{}\mathit{offset} < \grave{}\mathit{length}\ \mathsf{then} \\
\qquad (i', \mathit{screen}') := (\grave{}i + 1, \grave{}\mathit{screen}\dagger(\grave{}i, \mathit{char2glyph}\ \grave{}\mathit{line}@(\grave{}i + \grave{}\mathit{offset}))) \\
\quad \mathsf{else}\ (i', \mathit{screen}') := (\grave{}i + 1, \grave{}\mathit{screen}\dagger(\grave{}i, \cdot)); \\
\quad P(\grave{}\sigma, \sigma')
\end{array}
$$

This window is now also completely implemented, so we close it too.

Window: 6

$$
\begin{array}{l}
\quad\vdots \\
\sqsupseteq_3 \star \mathsf{if}\ \grave{}i = \mathit{MaxPos} + 1\ \mathsf{then\ skip} \\
\quad \mathsf{else\ if}\ \grave{}i + \grave{}\mathit{offset} < \grave{}\mathit{length}\ \mathsf{then} \\
\qquad (i', \mathit{screen}') := (\grave{}i + 1, \grave{}\mathit{screen}\dagger(\grave{}i, \mathit{char2glyph}\ \grave{}\mathit{line}@(\grave{}i + \grave{}\mathit{offset}))) \\
\quad \mathsf{else}\ (i', \mathit{screen}') := (\grave{}i + 1, \grave{}\mathit{screen}\dagger(\grave{}i, \cdot)); \\
\quad P(\grave{}\sigma, \sigma')
\end{array}
$$

Again, the implementation of this window is complete, so we can close it.

Window: 5

$$
\begin{array}{l}
\quad\vdots \\
\sqsupseteq_3 \star \mathsf{label}_{(v_p, O_p, I_p)}\ P. \\
\qquad \mathsf{if}\ \grave{}i = \mathit{MaxPos} + 1\ \mathsf{then\ skip} \\
\qquad \mathsf{else} \\
\qquad\quad \mathsf{if}\ \grave{}i + \grave{}\mathit{offset} < \grave{}\mathit{length}\ \mathsf{then} \\
\qquad\qquad (i', \mathit{screen}') := (\grave{}i + 1, \grave{}\mathit{screen}\dagger(\grave{}i, \mathit{char2glyph}\ \grave{}\mathit{line}@(\grave{}i + \grave{}\mathit{offset}))) \\
\qquad\quad \mathsf{else}\ (i', \mathit{screen}') := (\grave{}i + 1, \grave{}\mathit{screen}\dagger(\grave{}i, \cdot)); \\
\qquad\quad P(\grave{}\sigma, \sigma')
\end{array}
$$

The labelled block that is the focus of this window is now completely implemented. That is, the body of the bock is made up entirely from executable constructs. This means that we can replace the block with its true meaning—i.e. we may drop the subscripts from the label construct.

$$
\begin{array}{l}
\quad\vdots \\
\sqsupseteq_3 \star \mathsf{label}\ P. \\
\qquad \mathsf{if}\ \grave{}i = \mathit{MaxPos} + 1\ \mathsf{then\ skip} \\
\qquad \mathsf{else} \\
\qquad\quad \mathsf{if}\ \grave{}i + \grave{}\mathit{offset} < \grave{}\mathit{length}\ \mathsf{then} \\
\qquad\qquad (i', \mathit{screen}') := (\grave{}i + 1, \grave{}\mathit{screen}\dagger(\grave{}i, \mathit{char2glyph}\ \grave{}\mathit{line}@(\grave{}i + \grave{}\mathit{offset}))) \\
\qquad\quad \mathsf{else}\ (i', \mathit{screen}') := (\grave{}i + 1, \grave{}\mathit{screen}\dagger(\grave{}i, \cdot)); \\
\qquad\quad P(\grave{}\sigma, \sigma')
\end{array}
$$

We have now finished the implementation of this window, so we close it.

Window: 4

$$
\begin{array}{l}
\vdots \\
\sqsupseteq_3 \star \left\lfloor \begin{array}{ll} & I_p\ \sigma' \\ \wedge & (\mathit{offset}', \mathit{line}', \mathit{length}', \mathit{current}', \mathit{error}', \mathit{cursor}') = \\ & \quad (\grave{}\mathit{offset}, \grave{}\mathit{line}, \grave{}\mathit{length}, \grave{}\mathit{current}, \grave{}\mathit{error}, \grave{}\mathit{cursor}) \end{array} \right\rfloor ; \\
\quad \mathsf{label}\ P. \\
\quad\quad \mathsf{if}\ \grave{}i = \mathit{MaxPos} + 1\ \mathsf{then\ skip} \\
\quad\quad \mathsf{else} \\
\quad\quad\quad \mathsf{if}\ \grave{}i + \grave{}\mathit{offset} < \grave{}\mathit{length}\ \mathsf{then} \\
\quad\quad\quad\quad (i', \mathit{screen}') := (\grave{}i + 1, \grave{}\mathit{screen} \dagger (\grave{}i, \mathit{char2glyph}\ \grave{}\mathit{line} @ (\grave{}i + \grave{}\mathit{offset}))) \\
\quad\quad\quad \mathsf{else}\ (i', \mathit{screen}') := (\grave{}i + 1, \grave{}\mathit{screen} \dagger (\grave{}i, \cdot\,)); \\
\quad\quad\quad P(\grave{}\sigma, \sigma')
\end{array}
$$

In the focus of the current window the initialisation of the recursive block remains unimplemented. We therefore open a subwindow on the requirements for the initialisation.

Window: 5

$$
\begin{array}{l}
\vdots \\
\sqsupseteq_3 \star \left\lfloor \begin{array}{ll} & I_p\ \sigma' \\ \wedge & (\mathit{offset}', \mathit{line}', \mathit{length}', \mathit{current}', \mathit{error}', \mathit{cursor}') = \\ & \quad (\grave{}\mathit{offset}, \grave{}\mathit{line}, \grave{}\mathit{length}, \grave{}\mathit{current}, \grave{}\mathit{error}, \grave{}\mathit{cursor}) \end{array} \right\rfloor
\end{array}
$$

We expand the definition of I_p to see precisely what is required of the implementation.

$$
\sqsupseteq_3 \star \left\lfloor \begin{array}{ll} & i' \leq \mathit{MaxPos} + 1 \\ \wedge & \left(\begin{array}{l} \forall n.\, (n < i') \supset \\ \qquad \mathit{screen}' @ n = \mathit{glyph\_at}(\mathit{line}', \mathit{length}', n + \mathit{offset}') \end{array} \right) \\ \wedge & (\mathit{offset}', \mathit{line}', \mathit{length}', \mathit{current}', \mathit{error}', \mathit{cursor}') = \\ & \quad (\grave{}\mathit{offset}, \grave{}\mathit{line}, \grave{}\mathit{length}, \grave{}\mathit{current}, \grave{}\mathit{error}, \grave{}\mathit{cursor}) \end{array} \right\rfloor
$$

We can see that these requirements are trivially satisfied by assigning i the value zero.

$$
\sqsupseteq_3 \star i' := 0
$$

The initialisation of the labelled block has now been completely implemented. We may therefore close this window and return to its parent.

Window: 4

$$
\begin{array}{l}
\quad \vdots \\
\sqsupseteq_3 \star i' := 0; \\
\quad \mathsf{label}\ P. \\
\qquad \mathsf{if}\ \grave{}i = MaxPos + 1\ \mathsf{then\ skip} \\
\qquad \mathsf{else} \\
\qquad\quad \mathsf{if}\ \grave{}i + \grave{}offset < \grave{}length\ \mathsf{then} \\
\qquad\qquad (i', screen') := (\grave{}i + 1, \grave{}screen \dagger (\grave{}i, char2glyph\ \grave{}line @ (\grave{}i + \grave{}offset))) \\
\qquad\quad \mathsf{else}\ (i', screen') := (\grave{}i + 1, \grave{}screen \dagger (\grave{}i, \cdot)); \\
\qquad\quad P(\grave{}\sigma, \sigma')
\end{array}
$$

Now that the initialisation is implemented, the implementation of this window is also complete. We close this window too and return to its parent, which, as a result, is also complete.

Window: 3

$$
\begin{array}{l}
\quad \vdots \\
\sqsupseteq_3 \star \mathsf{var}\ i.\ i' := 0; \textit{[As in previous window]}
\end{array}
$$

This window is also complete, and so we close it.

Window: 2

$$
\begin{array}{l}
\quad \vdots \\
\sqsupseteq_3 \star cursor' := \grave{}current - \grave{}offset; \mathsf{var}\ i. \textit{[As in previous window]}
\end{array}
$$

This window may also be closed.

Window: 1

$$
\begin{array}{l}
\sqsupseteq_3 \star SeeCurrent(\grave{}offset, \grave{}current) \supset \\
\qquad\quad cursor' := \grave{}current - \grave{}offset; \textit{[As in previous window]}
\end{array}
$$

Returning to the top level of the window stack, we see that have completed the implementation of the original specification for the case where the predicate *SeeCurrent* is true. Fortunately the requirements call for no particular behaviour when the predicate *SeeCurrent* is false. Therefore, the implementation we have arrived at will do perfectly well for both cases, which means that the conditional expression may be dropped.

This step completes the implementation of the *PAINT* operation. We therefore display the resulting focus in its entirety.

$$
\begin{array}{l}
\sqsupseteq_3 \star\, cursor' := \grave{}current - \grave{}offset; \\
\quad \mathsf{var}\, i.\, i' := 0; \\
\quad\quad \mathsf{label}\, P.\, \mathsf{if}\, \grave{}i = MaxPos + 1\ \mathsf{then\ skip} \\
\quad\quad\quad \mathsf{else\ if}\, \grave{}i + \grave{}offset < \grave{}length\ \mathsf{then} \\
\quad\quad\quad\quad (i', screen') := \\
\quad\quad\quad\quad\quad (\grave{}i + 1, \grave{}screen \dagger (\grave{}i, char2glyph\ \grave{}line @ (\grave{}i + \grave{}offset))) \\
\quad\quad\quad\quad \mathsf{else}\, (i', screen') := (\grave{}i + 1, \grave{}screen \dagger (\grave{}i, \cdot)); \\
\quad\quad\quad\quad P(\grave{}\sigma, \sigma')
\end{array}
$$

The implementation of the description *PAINT* is now complete.

B.5 Comments

Comments on the case study, including the detailed presentation given in this appendix, can be found in section 6.10 (the conclusion of the chapter about the case study).

References

[Ale85] Heather Alexander. Specifying and prototyping a line editor: A comparison between Z and **me too**. Internal Report RGP-24, Department of Computing Science, University of Stirling, Stirling FK9 4LA, Scotland, November 1985.

[Ale88] Heather Alexander. Comments on 'Formal specification of user interfaces: A comparison and evaluation of four axiomatic approaches'. *IEEE Transactions on Software Engineering*, 14(4):438–439, April 1988.

[Bac81] Ralph-Johan R. Back. On correct refinement of programs. *Journal of Computer and System Sciences*, 23(1):49–68, February 1981.

[Bac88] Ralph-Johan R. Back. A calculus of refinements for program derivations. *Acta Informatica*, 25(6):593–624, 1988.

[BBP93] Dines Bjørner, Manfred Broy, and Igor V. Pottosin, editors. *Proceedings of the International Conference on Formal Methods in Programming and Their Applications*, volume 735 of *Lecture Notes in Computer Science*, Academgorodok, Novosibirsk, Russia, 28 June – 3 July 1993. Springer-Verlag, Berlin.

[BCJ84] Howard Barringer, J. H. Cheng, and Cliff B. Jones. A logic covering undefinedness in program proofs. *Acta Informatica*, 21(3):251–269, 1984.

[BD77] Rod M. Burstall and John Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, January 1977.

[Bek84] Hans Bekič. Definable operations in general algebras, and the theory of automata and flowcharts. In Cliff B. Jones, editor, *Programming Languages and Their Definition*, volume 177 of *Lecture Notes in Computer Science*, pages 30–55. Springer-Verlag, Berlin, 1984.

[BKT94] Yves Bertot, Gilles Kahn, and Laurent Théry. Proof by pointing. In Takayasu Ito, Masami Hagiya, Albert R. Meyer, and John Mitchell, editors, *Proceedings of the Symposium on Theoretical Aspects of Computer*

Science, Tohoku University, Sendai, Japan, 19–21 April 1994. Information Processing Society of Japan.

[BM79] Robert S. Boyer and J Strother Moore. *A Computational Logic*. ACM Monograph Series. Academic Press, New York, 1979.

[BM88] Robert S. Boyer and J Strother Moore. *A Computational Logic Handbook*. Academic Press, New York, 1988.

[BMPP89] Friedrich Ludwig Bauer, Bernhard Möller, Helmut Partsch, and Peter Pepper. Formal program construction by transformations—computer-aided, intuition-guided programming. *IEEE Transactions on Software Engineering*, 15(2):165–180, February 1989.

[Boch38] D. A. Bochvar. Ob odnom trexznachnom ischislenii i ego primenenii k analizu paradoksov klassicheskogo rasshirennogo funktsional'nogo ischisleniya [On a three-valued calculus and its application to the analysis of contradictions of expanded functional calculus]. *Matematicheskii sbornik*, 4(46):287–308, 1938.

[BR91] Stephen P. Bear and Tony W. Rush. Rigorous software engineering: A method for preventing software defects. *Hewlett-Packard Journal*, 42(5):24–31, December 1991.

[BvW90] Ralph-Johan R. Back and Joakim von Wright. Refinement concepts formalised in higher-order logic. *Formal Aspects of Computing*, 2(3):247–272, July 1990.

[BW91] Manfred Broy and Martin Wirsing, editors. *Methods of Programming: Selected Papers on the CIP-Project*, volume 544 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1991.

[Chi85] Uli H. Chi. Formal specification of user interfaces: A comparison and evaluation of four axiomatic approaches. *IEEE Transactions on Software Engineering*, 11(8):671–685, August 1985.

[CIP85] CIP Language Group. *The Munich Project CIP—Volume I: The Wide Spectrum Language CIP-L*, volume 183 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1985.

[CIP87] CIP System Group. *The Munich Project CIP—Volume II: The Program Transformation System CIP-S*, volume 292 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1987.

[Cor93] Francisco Corella. What holds in a context? *Journal of Automated Reasoning*, 10(2):79–93, February 1993.

[DF88] Edsger W. Dijkstra and Wim H. J. Feijen. *A Method of Programming*. Addison-Wesley, Workingham, England, 1988.

[Dij68] Edsger W. Dijkstra. A constructive approach to the problem of program correctness. *BIT*, 8:174–186, 1968.

[Dij76] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice Hall Series in Automatic Computation. Prentice Hall, New York, 1976.

[Flo66] Robert W. Floyd. Assigning meaning to programs. In Jacob T. Schwartz, editor, *Mathematical Aspects of Computer Science: Proceedings of the 19th Symposium in Applied Mathematics*, pages 19–32, Providence, United States, 1966. AMS.

[GM93] Michael J. C. Gordon and Thomas F. Melham, editors. *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press, Cambridge, England, 1993.

[GMW79] Michael J. C. Gordon, Robin Milner, and Chistopher P. Wadsworth. *Edinburgh LCF*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1979.

[GNU92] Lindsay Groves, Raymond Nickson, and Mark Utting. A tactic driven refinement tool. In Jones et al. [JDS92], pages 272–297.

[Gor85] Michael J. C. Gordon. Why higher-order logic is a good formalism for specifying and verifying hardware. In George J. Milne and P. A. Subrahmanyam, editors, *Proceedings of the 1985 Edinburgh Workshop on VLSI Design: Formal Aspects of VLSI Design*, pages 153–177, Edinburgh, Scotland, 1985. North Holland, Amsterdam.

[Gor88] Michael J. C. Gordon. Mechanizing programming logics in higher-order logic. In Graham M. Birtwistle and P. A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving (Proceedings of the Workshop on Hardware Verification)*, pages 387–439, Banff, Canada, 1988. Springer-Verlag, Berlin.

[Gor93] Andrew Donald Gordon. Transformation design using a proof assistant. Internal report of the HOL-ELLA project, University of Cambridge, Computer Laboratory, New Museums Site, Pembroke Street, Cambridge CB2 3QG, England, 1993.

[Gra92] Andrew Gravell. Constructive refinement of first-order specifications. In Jones et al. [JDS92], pages 181–210.

[Gru91a] Jim Grundy. The HOL window library. In SRI [SRI91].

[Gru91b] Jim Grundy. Report on m-EVES. ERL Research Report ERL-0545-RR, Information Technology Division, Electronics Research Laboratory, Defence Science & Technology Organisation, Building 171 Laboratories Area, PO Box 1500, Salisbury SA 5108, Australia, March 1991.

[Gru91c] Jim Grundy. Window inference in the HOL system. In Windley et al. [WALJ91], pages 177–189.

[Gru92] Jim Grundy. A window inference tool for refinement. In Jones et al. [JDS92], pages 230–254.

[Gru93a] Jim Grundy. Predicative programming—a survey. In Bjørner et al. [BBP93], pages 8–25.

[Gru93b] Jim Grundy. A three-valued logic for refinement. In Bjørner et al. [BBP93], pages 26–42.

[Hay87] Ian J. Hayes. *Specification Case Studies*. Prentice Hall International Series in Computer Science. Prentice Hall International, London, 1987.

[HD85] F. Keith Hanna and Neil Daeche. Specification and verification using higher-order logic. In Cees-Jan Koomen and Toru Moto-oka, editors, *Proceedings of the 7th International Conference on Computer Hardware Description Languages and Their Applications*, pages 418–433, Tokyo, August 1985. IFIP, North Holland, Amsterdam.

[Heh84] Eric C. R. Hehner. Predicative programming—part 1. *Communications of the ACM*, 27(2):134–143, February 1984.

[Heh88] Eric C. R. Hehner. Termination conventions and comparative semantics. *Acta Informatica*, 25(1):1–14, 1988.

[Heh89] Eric C. R. Hehner. Termination is timing. In Jan L. A. van da Snepscheut, editor, *Mathematics of Program Construction: 375th Anniversary of the Groningen University International Conference*, volume 375 of *Lecture Notes in Computer Science*, pages 36–47, University of Groningen, Groningen, The Netherlands, June 1989. Springer-Verlag, Berlin.

[Heh90] Eric C. R. Hehner. A practical theory of programming. *Science of Computer Programming*, 14(2-3):133–158, 1990.

[HGM86] Eric C. R. Hehner, Lorene E. Gupta, and Andrew J. Malton. Predicative methodology. *Acta Informatica*, 23(5):487–505, 1986.

[HH86] C. A. R. Hoare and Jifeng He. The weakest prespecification. *Fundamenta Informaticae*, 9:51–84,217–251, 1986.

[HH87] C. A. R. Hoare and Jifeng He. The weakest prespecification. *Information Processing Letters*, 24(2):127–132, January 1987.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580,583, October 1969.

[Hoa84] C. A. R. Hoare. Programs are predicates. *Philosophical Transactions of the Royal Society of London—Series A: Mathematical and Physical Sciences*, 312(1522):475–490, October 1984.

[HPP92] Anne Haxthausen, Jan Storbank Pedersen, and Søren Prehn. RAISE overview. Part RAISE/CRI/DOC/9/V6, Computer Resources International, Bregnerødvej 144, PO Box 173, DK-3460 Birkerød, Denmark, August 1992.

[JDS92] Cliff B. Jones, B. Tim Denvir, and Roger C. F. Shaw, editors. *Proceedings of the 5th Refinement Workshop*, Workshops in Computing, Lloyd's Register, London, 8–10 January 1992. BCS-FACS, Springer-Verlag, London.

[JM93] Bart Jacobs and Tom Melham. Translating dependent type theory into higher order logic. In M. Bezem and J. F. Groote, editors, *Typed Lambda Calculi and Applications: Proceedings of the International Conference*, volume 664 of *Lecture Notes in Computer Science*, pages 209–229, Utrecht, The Netherlands, 16–18 March 1993. Springer-Verlag, Berlin.

[Jon90] Cliff B. Jones. *Systematic Software Development Using VDM*. Prentice Hall International Series in Computer Science. Prentice Hall International, London, 2 edition, 1990.

[Kau88] Matt Kaufmann. A user's manual for an interactive enhancement to the Boyer-Moore theorem prover. Technical Report 19, Computational Logic Incorporated, Suite 290, 1717 West 6th Street, Austin TX 78703, United States, May 1988.

[KB86] Bernd Krieg-Brückner. Integration of program construction and verification: the PROSPECTRA methodology. In A. Nico Habermann and Ugo Montanari, editors, *System Development in Ada: Proceedings of the Workshop on Software Factories and Ada*, volume 275 of *Lecture Notes in Computer Science*, pages 173–194, Capri, Italy, 26–30

May 1986. Consorzio per le Ricerche e le Applicazioni di Informatica, Springer-Verlag.

[KB91] Bernd Krieg-Brückner. Transformational meta program development. In Broy and Wirsing [BW91], pages 19–33.

[Kle38] S. C. Kleene. On notation for ordinal numbers. *The Journal of Symbolic Logic*, 3(4):150–155, December 1938.

[Lar93] Mats Larsson. A transformational approach to formal digital system design. Licentiate thesis, Institutionen för Datavetenskap, Universitetet och Tekniska Högskolan i Linköping, S-581 83 Linköping, Sweden, May 1993.

[Luk20a] Jan Łukasiewicz. O logice trójwartościowej [On three-valued logic]. *Ruch Filozoficzny*, 5:170–171, 1920. translated in [McC67].

[Luk20b] Jan Łukasiewicz. O pojęciu możliwości [On the notion of possibility]. *Ruch Filozoficzny*, 5:169–170, 1920. translated in [McC67].

[McC67] Storrs McCall, editor. *Polish Logic 1920–1939*. Oxford University at the Clarendon Press, Oxford, England, 1967.

[MDG86] Ali Mili, Jules Desharnais, and Jean Raymond Gagné. Formal models of stepwise refinement of programs. *ACM Computing Surveys*, 18(3):231–276, September 1986.

[Mil83] Ali Mili. A relational approach to the design of deterministic programs. *Acta Informatica*, 20(4):315–328, 1983.

[Mon88] Leonard G. Monk. Inference rules using local contexts. *Journal of Automated Reasoning*, 4(4):445–462, December 1988.

[Mor87] Joseph M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*, 9(3):287–306, December 1987.

[Mor88a] Carroll C. Morgan. Data refinement by miracles. *Information Processing Letters*, 26(5):243–246, January 1988.

[Mor88b] Carroll C. Morgan. The specification statement. *ACM Transactions on Programming Languages and Systems*, 10(3):403–419, July 1988.

[MR87] Carroll C. Morgan and Ken A. Robinson. Specification statements and refinement. *IBM Journal of Research and Development*, 31(5):546–555, September 1987.

[MXYQ86] Ali Mili, Wang Xiao-Yang, and Yu Qing. Specification methodology: An integrated relational approach. *Software—Practice and Experience*, 16(11):1003–1030, November 1986.

[Naf92] Maurice Naftalin. A model of the refinement process. In Jones et al. [JDS92], pages 211–229.

[Nau66] Peter Naur. Proof of algorithms by general snapshots. *BIT*, 6:310–316, 1966.

[Nei90] D. S. Neilson. From Z to C: Illustration of a rigorous development method. Technical Monograph PRG-101, Oxford University Computing Laboratory, Programming Research Group, 8–11 Keble Road, Oxford OX1 3QD, England, 1990.

[Nel89] Greg Nelson. A generalization of Dijkstra's calculus. *ACM Transactions on Programming Languages and Systems*, 11(4):517–561, October 1989.

[Ngu91] Thanh Tung Nguyen. A relational model of demonic nondeterministic programs. *International Journal of Foundations of Computer Science*, 2(2):101–131, June 1991.

[NHWG89] Morgens Nielsen, Klaus Havelund, Kim Ritter Wagner, and Chris George. The RAISE language, method and tools. *Formal Aspects of Computing*, 1(1):85–114, January–March 1989.

[ORA90] Odyssey Reseach Associates, Ottawa, Canada. *m-EVES Collected Papers*, 1990.

[Par79] David Park. On the semantics of fair parallelism. In Dines Bjørner, editor, *Abstract Software Specifications: Proceedings of the Winter School*, volume 86 of *Lecture Notes in Computer Science*, pages 504–526, Copenhagen, 22 January – 2 February 1979. Springer-Verlag, Berlin.

[Par85] David Lorge Parnas. Technical correspondence: Predicative programming. *Communications of the ACM*, 28(5):354–356, May 1985.

[Pau87] Lawrence C. Paulson. *Logic and Computation: Interactive Proof with Cambridge LCF*, volume 2 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, Cambridge, England, 1987.

[Pau93] Lawrence C. Paulson. The Isabelle reference manual. Technical Report 283, University of Cambridge, Computer Laboratory, New Museums Site, Pembroke Street, Cambridge CB2 3QG, England, February 1993.

[Pre87] Søren Prehn. From VDM to RAISE. In Dine Bjørner, Cliff B. Jones, Mícheál Mac an Airchinnigh, and Erich J. Neuhold, editors, *VDM—A Formal Method at Work: Proceedings of the VDM-Europe Symposium*, volume 252 of *Lecture Notes in Computer Science*, pages 141–149, Brussels, Belgium, 23–26 March 1987. Springer-Verlag, Berlin.

[RS93] Peter J. Robinson and John Staples. Formalising the hierarchical structure of practical mathematical reasoning. *Journal of Logic and Computation*, 3(1):47–61, February 1993.

[RTP90] Rex Tompson & Partners, Surry, England. *MALPAS Users Guide*, 1990.

[Sek92] Emil Sekerinski. A calculus for predicative programming. In Richard S. Bird, Carroll C. Morgan, and Jim C. P. Woodcock, editors, *Mathematics of Program Construction: Proceedings of the 2nd International Conference*, volume 669 of *Lecture Notes in Computer Science*, pages 302–332, St. Catherine's College, Oxford University, Oxford, England, 29 June – 3 July 1992. Springer-Verlag, Berlin.

[Smi92] Alf Smith. Proof of theorems in Z. Report 92006, Defence Research Agency, St Andrews Road, Great Malvern, Worcstershire WR14 3PS, England, March 1992.

[SRI91] SRI International, Cambridge Research Center, Millers Yard, Mill Lane, Cambridge CB2 1RQ, England. *The HOL System*, 2.01 edition, July 1991.

[SS71] Dana S. Scott and Christopher Strachey. Towards a mathematical semantics for computer languages. In Jerome Fox, editor, *Proceeding of the Symposium on Computers and Automata*, volume 21 of *Microwave Research Institute Symposia Series*, pages 19–46, Brooklyn, United States, 1971. Polytechnic University, Microwave Research Institute, Wiley-Interscience.

[Suf82] Bernard Sufrin. Formal specification of a display-oriented text editor. *Science of Computer Programming*, 1:157–202, 1982.

[TBK92] Laurent Théry, Yves Bertot, and Gilles Kahn. Real theorem provers deserve real user-interfaces. Rapports de Recherche: Programme 2 — N° 1684, INRIA, Domaine de Volceau, Rocquencourt, BP 105, F-78153 Le Chesnay Cedex, France, May 1992.

[Thé93] Laurent Théry. An X-windows interface for the window inference system. In Mike Gordon, editor, The Final Report on DSTO Grant: Improved Interface for HOL, Internal Report.University of Cambridge, Computer

Laboratory, New Museums Site, Pembroke Street, Cambridge CB2 3QG, England, 1993.

[TRS90] Tong Gao Tang, Peter J. Robinson, and John Staples. The demonstration proof editor Demo2. Technical Report 175, Key Centre for Software Technology, Department of Computer Science, University of Queensland, QLD 4072, Australia, December 1990.

[Vic90] Trevor Vickers. An overview of a refinement editor. In *Proceedings of the 5th Australian Software Engineering Conference*, pages 39–44, Sydney, 22–25 May 1990. IREE.

[vW94] Joakim von Wright. Program refinement by theorem prover. In David Till and Roger C. F. Shaw, editors, *Proceedings of the 6th Refinement Workshop*, Workshops in Computing, City University, London, 5–7 January 1994. BCS-FACS, Springer-Verlag, London.

[vWHLL93] Joakim von Wright, J. Hekanaho, P. Loustarinen, and P. Långbaka. Mechanising some advanced refinement concepts. *Formal Methods in System Design*, 3(1/2), August 1993.

[vWS91] Joakim von Wright and K. Sere. Program transformations and refinements in HOL. In Windley et al. [WALJ91], pages 231–239.

[WALJ91] Phillip J. Windley, Myla Archer, Karl N. Levitt, and Jeffrey J. Joyce, editors. *Proceedings of the International Tutorial and Workshop on the HOL Theorem Proving System and its Applications*, University of California at Davis, 28–30 August 1991. ACM SIGDA / IEEE Computer Society, IEEE Computer Society Press, Los Alamitos, United States.

[Wir71] Nicklaus Wirth. Program development by stepwise refinement. *Communications of the ACM*, 14(4):221–227, April 1971.

[You87] William D. Young. GVE user's manual: Examples of proof commands. Technical Report 8, Computational Logic Incorporated, Suite 290, 1717 West 6th Street, Austin TX 78703, United States, 1987.

Index

Page numbers are given in bold type where they refer to a definition or primary source of information about the item being indexed. Page numbers are given in slanted type where they refer to an illustrative example. Page numbers followed by 'n' are references to footnotes. Page numbers followed by 'f' are references to figures.

$\lfloor \_ \rfloor$, $\lceil \_ \rceil$, **56**
$\mathbb{M}$, **57**, *64*
@, **125**
†, **125**
†@ii_THM, †@ij_THM, **126**
$\downarrow$, **124**
$\bot$, **32**, **53**
$\bot_3$, **56**
, , **123**
$\supset$, **124**
$\supset\downarrow$1_THM, $\supset\downarrow$2_THM, **125**
$\supset\!\!)$, **57**, *58*
=, **56**
$\equiv$, **56**
$\underline{\supset}$, **56**
$\supset\sqsubseteq_3$ _THM, **61**
$\not\vdash$, **24**
$\underline{\sqsupset}$, **56**
$\underline{\vee}$, **56**
$\triangleright$, **124**
$\triangleright\downarrow$0_THM, $\triangleright\downarrow i$_THM, **125**
$\langle\,\rangle$, **124**
!, **12**
?, **24**
$\not?$, **25**
$\underline{?}$, **24**
i, **22**
$\cdot$, **98**
_ $\Rightarrow$ _ | _, **56**n
$\subset$, **16**n
#, **124**
#0_THM, **125**
#$\supset$_THM, **125**
#$\triangleright$_THM, #$\triangleleft$_THM, **125**
#$\langle\,\rangle$_THM, **124**
$\neg$#0$\triangleright$_THM, $\neg$#0$\triangleleft$_THM, **125**
|, **36**, **39**, **43**, 44, **46**, 50, **63**
;, **36**
:=, **36**, **39**, **43**, **46**, **63**, 72–73
;, **39–40**, **43**, **47**, 50, **64**, 74–79
$\underline{\exists}$, $\underline{\forall}$, **57**
$[\![\_]\!]$, **58**, 60
$[\![\_]\!]_p$, **41**
$[\![\_]\!]_r$, **37**
$[\![\_]\!]_t$, **45**
$\triangleleft$, **124**
$\triangleleft\downarrow i$_THM, $\triangleleft\downarrow$#_THM, **125**
$\sqcup$, **100**
$\sqsubseteq$, **33**
$\sqsubseteq_3$, $\sqsubseteq_3\sqsubseteq_D$ _THM, **61**
$\sqsubseteq_D$, **60**
$\sqsubseteq_p$, $\sqsubseteq_p$ _THM, **41–42**
$\sqsubseteq_r$, $\sqsubseteq_r$ _THM, **38**
$\sqsubseteq_t$, $\sqsubseteq_t$ _THM, **45**

abort, **34**, 37, **38**, 40, **42**, **46**, 49, **59**
abstraction, **101**, *see also IsAbs*
applicable
  suppositions, *see* suppositions
  theorems, *see* theorems
assignment, *see* :=
assumptions, **12**, 17, 20–22

Back, Ralph-Johan R., 5, 29
behaviour, **33**
Bochvar, D. A., 55
BOOL3, **56**
Boyer-Moore theorem prover, 28
Burstall, Rodney Martineau, 6, 7
but_last, **124**

$\mathcal{C}$, **24**
Centaur, 29
chance, **34**, **38**, 40, **42**, **45**, 49, 54, **59**
chaos, **34**, 37, **38**, 41, **42**, **45**, 49, **59**
CHAR, **100**
char2glyph, **100**
choice, *see* |
CIP, 7
clear, *clear*_THM, **62**
closing, *see* window
composition, *see* ;
conditional
  expression, *see* _ ⇒ _ | _
  statement, *see* if _ then _ else _
conjectures, 12, **24**, *25–27*, *see also* ?
  used, **24**, *see also* ⁉ and ⁇
context, 11, **22**, 23, **24**, 94
  windows on, **23–24**, *25*
Corrella, Francisco, 28
current, **101**, 105, 106
cursor, **99**, 104
CursorCurrent, **105**

Darlington, John, 6–7
del, **102**
Demo2, 12, 28
depth, *see* window, depth
description, ***57–58***
  interpretation, 58, *see also* $[\![\_]\!]$
  refinement, **60**, *61*, *see also* $\sqsubseteq_D$, $\sqsubseteq_3$
Dijkstra, Edsger Wybe, 2, 3, 5, 31, 37

EDIT, 101, **107**, 108, 109, 113, 135
EDIT 1, **102–103**
EDIT 2, **104**
EDIT 3, **106**
empty glyph, 104, *see also* ·
empty_line, **102**
$\mathcal{E}$, **107**n
Error, **103**
error, **99**
extended total model, **53**, 55

F_3, **56**
fix, **40**n, 48
$\overline{\text{fix}}$, **44**
fixed point, **40**n
  greatest, 44, 51, *see also* $\overline{\text{fix}}$
  least, **40**n, 44, 51, *see also* fix
Floyd, Robert W., 1
focus, *see* window
fst, **123**
fst, snd_THM, **123**
fst_THM, **123**

Generic-PC, 97, **98**f, 99
GLYPH, **98**, 99–100
glyph, **98**
glyph_at, **104**
GoodScreen, **105**, 109
Gordon, Andrew Donald, 28, 122
Gordon, Michael J. C., 41
Gravell, Andrew, 37
guarded-command language, **3**, 5, 37
GVE, 2

head, **124**
Hehner, Eric C. R., 6, 31, 44, 53, 122
Hilbert's epsilon, *see* $\mathcal{E}$
Hoare logic, 2, 41, 62

Hoare, Charles Anthony Richard, 1, 6, 31, 44, 53
HOL, 11–12, 14, 15, 28
  expressions, **19**
  window interface, 20, 21, 27, 30
HP-SL, 31, 37

I_d, **111**, **157**
if _ then _ else _, **36**, **39**, **43**, **46**, **63**, 73
I_m, **86**
INIT, 101, **106**, 108–109, 127
INIT 1, **102**, 108
INIT 2, **104**, 108
INIT 3, **105**
ins, **102**
interface language, **2**, 4
invariant, 84–85
I_p, **114**, **175**
is_char_key, **100**
Isabelle, 12
IsAbs, **101**, 107
IsView, **105**

Jones, Clifford Bryn, 4, 56, 120

Kaufmann, Matt, 28
KEY, **99**
key, **99–100**
*KEY*_THM, **103**
key2char, **100**
Kleene, S. C., 55

$\mathcal{L}$, **22**, 23–25, *26*
label, **36**, **40**, **44**, **48**, 51, 64, 80–85
$\mathsf{label}_{(v,O)}$, **81**
$\mathsf{label}_{(v,O,I)}$, **84**
Larsson, Mats, 28, 122
last, **124**
left, **102**
lemmas, **22**, 25
  creating, **23**, *26*
  windows on, *see* context
length, **100**, 101, 106

LINE, **101**
line, **100**, 101
local variable, *see* var
Logic of Computable Functions, 4, 56
Logic of Partial Functions, 4, 56
Łukasiewicz, Jan, 55

m-EVES, 2
magic, 34, *see also* miracle
MALPAS, 2
max, **125**
MaxChar, 100
MaxLength, **100**
MaxPos, **98**, 104
meets, **33**
Mili, Ali, 6, 31, 37
MinMove, **105–106**
miracle, **34**, 37, **38**, **42**, **46**, 49, **59**
ML, 15
Monk, Leonard G., 28
monotonic, **40**n, 48, 64, 81
Morgan, Carroll C., 5
Morris, Joseph M., 5

natural deduction, *see* reasoning
Naur, Peter, 2
Nguyen, Thanh Tung, 37
nondeterminism, 7, **33**, 41, *see also* |
  angelic, **33**, 39–40, 47
  demonic, **33**, 37, 50–51
nontermination, **32**, 37, 41, 50, 66, 81

O_d, **157**
offset, **104**, 109
O_m, **87**
O_p, **176**
opening, *see* window
outcomes, **32**, 33–34

PAINT, 108, **109**, 110, 113, 115, **133**, 137, 172
Park, David, 41
partial correctness, **31**n, 41, 69

partial model, 41–44, 69
  interpretation, **41**, *see also* $[\![\_]\!]_p$
  refinement, **41**, 48–49, *see also* $\sqsubseteq_p$
path, 18
PC-NQTHM, 28
position, *see* path
power domain, 3
predicates, **35**
  interpretation of, **35**, *see also* $[\![\_]\!]$
predicative programming, **6**
preorder, **12**n, 14, 19
program calculation, **2**
program refinement, **5**
proof, *see* reasoning
PROSPECTRA, 7

RAISE, 4–6
reasoning
  goal directed, 12, 30, 119
  natural deduction, 12, 14, 119
  tactical, 30
  transformational, 11, 69
  window inference, *13–14*
recursive block, *see* label
refinement, ***33–34***, *70–71*, *see also* $\sqsubseteq$
refinement calculus, **5**
relational model, 37–41, 54
  interpretation, **37**, *see also* $[\![\_]\!]_r$
  refinement, **38**, 48–49, *see also* $\sqsubseteq_r$
right, **102**
rigorous, **3**
Robinson, Peter J., 12, 27

Scott, Dana S., 4, 56, 120
screen, 98–99, 104
SeeCurrent, **105**
Sekerinski, Emil, 37
sequential composition, *see* ;
σ, $\grave{}\sigma$, σ', **32**, 86, 107
Σ, **32**, 36
$\Sigma_\perp$, **32**
skip, **36**, **39**, **42**, **46**, **63**, 71–72
Smith, Alf, 29
snd, **123**
snd_THM, **123**
space character, *see* $\sqcup$
specification, **33**
  algebraic, **7**
  model oriented, **7**
Staples, John, 12, 27
state, **32**, *see also* σ
  final, **32**, 35, *see also* σ'
  initial, **32**, 33–35, *see also* $\grave{}\sigma$
state space, *see* Σ
step-wise refinement, **3**, 5
stronger relations, *see* window rules
substitution, **16**n, **20**n
Sufrin, Bernard A., 101
suppositions, **24**, *see also* ?-
  applicable, **24**

T_3, **56**
tactics, *see* reasoning
tail, **124**
Tarski-Knaster theorem, **40**n
termination, *see* nontermination
theorems
  applicable, **22**, 25
Théry, Laurent, 29, 121
timed model, 53
total correctness, 31, 54
total model, 44–48, 54
  interpretation, **45**, *see also* $[\![\_]\!]_t$
  refinement, **45**, 48–49, *see also* $\sqsubseteq_t$
transformational programming, **6**, 7

var, **36**, **40**, **43**, **48**, 51, **64**, 79–80
variant, **81**
v_d, **112**, **157**
VDM, 3–4, 62
verification, **2**, 30, 93, 119
view, **104**, 105
v_m, **87**
von Wright, Peter Joakim, 29, 121
v_p, **114**, **176**

weakest precondition, **3**, 4–5, 121
wide-spectrum language, **4**, 5, 6
window
  closing, **12**, *13*, *16*, *18*
  depth, **85**, 127
  focus of, **12**, 17
  opening, **12**, *13*, *16*, *17*
  relation of, **12**, 17, 18
  theorem held by, **14**, 24, 25
  transformation, 12, 15
    justification of, 15
window inference, 11–30
window rules, **15–17**
  basic, **19–20**
  choosing, *17*, **21**
  composing, **18–19**
  stronger, **19**
  substituting, 18
window stack, 12
  creating, 14, *17*
Wirth, Nicklaus, 3

Z, 29, 31, 37