A Relational Basis for Program Construction by Parts

by
Marc Frappier

A thesis
presented to the School of Graduate Studies and Research
in partial fulfillment of the requirements
for the degree of Ph.D. in Computer Science

University of Ottawa
Ottawa, Ontario, Canada
May 1995

À Félicien et Hermance

Abstract

Program construction by parts consists in tackling a complex specification one component at a time, developing a partially defined solution for each component, then combining the partial solutions into a global solution for the aggregate specification. This method is desirable whenever the specification at hand is too complex to be grasped in all its detail. It is feasible whenever the specification at hand is structured as an aggregate of clearly defined subspecifications —where each subspecification represents a simple functional requirement.

Our approach is based on relational specifications, whereby a specification is described by a binary relation. The set of relational specifications is naturally ordered by the refinement ordering, which provides a lattice-like structure. The join of two specifications S and S' is the specification that carries all the functional features of S and all the functional features of S'. Complex specifications are naturally structured as the join of simpler subspecifications.

On this basis, we have defined a language where specifications and programs can be represented. This language can represent specifications (structured as joins), programs (structured by means of traditional Pascal-like control structures) and intermediate designs (by means of a mixture of specification constructs and program constructs). Also, we have provided a set of rules for transforming a specification into a program, by changing a representation in this language from a specification structuring into a program structuring. The transformations we propose are correctness preserving in the sense that they map one representation into a more refined representation.

Contents

List of Figures

Acknowledgements

I wish to express my gratitude to following friends and colleagues. To professor Ali Mili, my thesis supervisor, who dedicatedly and patiently introduced me to the domain of formal program derivation; his guidance, advices and encouragements are inestimable. To Jules Desharnais, who helped me find my way in the world of relation algebra and formal specifications, and provided valuable comments on this research work. To the members of my advisory committee, Evangelos Kranakis, Luigi Logrippo and Phil Scott, for gracefully accepting to review this work, and for introducing me to algebraic specifications and logic. To Francis Bordeleau, as a fellow doctoral student, for sharing my frustrations and my hopes, and, as a friend, for making these three years in Ottawa a memorable experience. To Alain Beaudry, for reminding me that life has many facets, and work is just one of them. To the late Vianney Côté, Jean Goulet and Richard St-Denis, for once telling me that I should do a Ph. D. To Dominique Mongrain, for believing in me at times when I forgot to, and for explaining to me the meaning of the word determination. To Marie and Isabelle, for helping me to never forget that the greatest treasure is your family.

Funding for this research was provided by NSERC (Natural Sciences and Engineering Research Council) of Canada, by FCAR (Fonds pour la Formation de Chercheurs et l'Aide à la Recherche) of Québec, and by the School of Graduate Studies and Research of the University of Ottawa.

Part I

Background

Chapter 1

Introduction

"Le second, de diviser chacune des difficultés que j'examinerais, en autant de parcelles qu'il se pourrait, et qu'il serait requis pour les mieux résoudre.

Le troisième, de conduire par ordre mes pensées, en commençant par les objets les plus simples et les plus aisés à connaître, pour monter peu à peu, comme par degrés, jusques à la connaissance des plus composés; et supposant même de l'ordre entre ceux qui ne se précèdent point naturellement les uns les autres."

René Descartes, 1596-1650
Discours de la Méthode, June 8th, 1637

1.1 Constructing Programs from Specifications

Despite decades of research, the routine construction of programs from specifications under acceptable conditions of quality and productivity remains to a large extent beyond our reach. To be sure, we have achieved great progress in understanding the mechanics of program correctness and programming language semantics. Also, some of this understanding of program correctness has been put to bear on the problem of constructing correct programs from formal specifications.

But there is more to program construction than the ideas of program correctness and programming language semantics. If we analyze the process of constructing a program from a specification, we find two classes of decisions: decisions that are driven by *correctness preservation* concerns, which are mostly clerical in nature; decisions that are determined at the *programmer's discretion*, which are essentially creative in nature [33, 61]. Ideas of program correctness can help us tackle decisions in the former class, but are of little use with decisions of the latter class. Unfortunately, because they require creativity, decisions in the latter class are most worthy of investigation for possible assistance or automation. As a result, our understanding of program correctness and programming language semantics, although very beneficial for program verification, has led to little advances in program construction. Most typically, rules for program construction are rules of program correctness which have been reworded.

Another predicament of the current state of the art in program construction is the problem of scale: the advances in programming language semantics and program correctness have not addressed the issue of handling large, complex software components. To draw an analogy with hardware design, we observe that the hardware industry achieved a great deal of gains in quality and productivity by moving from an emphasis on logical gates and connectives (*logic design*) to an emphasis on standard reusable logical devices (*VLSI design*). The research in program construction is still in a *logic design* mind set. To be sure, the construction of programs from specifications is so information-intensive, and requires so much precision, that attention to semantic issues and correctness issues is necessary. Yet at the same time it appears that the software industry will not achieve the desired level of quality and productivity unless it scales up its program construction methods in such a way that they can deal with large scale programs and systems without sacrificing precision.

1.2 Some General Prescriptions

In the previous section we have identified two primary shortcomings in the current state of the art in program construction: first, the fact that existing proposals provide assistance with clerical decisions, when most help is needed with the creative decisions; second, the fact that existing proposals deal with a level of detail that is not likely to foster the expected gains in productivity and quality. We submit that gains in productivity and quality can be achieved by a combination of the following techniques, which are not unrelated: *software reuse*, *program merging*, and *program construction by parts*.

Software reuse consists in developing standard software components with a view to reusing them in several host software systems. This technology offers great potential in terms of productivity because a programmer may *produce* arbitrarily large quantities of code by a simple retrieval operation. It also offers great potential in terms of quality because a component that is written for the purpose of being reused repeatedly will be validated with more care than a component that is due to be used only once; also, all the previous uses of the component constitute as many opportunities for the component to be tested and validated. Software reuse relies heavily on two key technologies: program specification, which deals with describing properties of programs at an arbitrary (esp. arbitrarily high) level of abstraction; program composition, which deals with combining existing software components to produce an aggregate system with predefined functional properties.

Program merging consists in taking two programs and composing them to obtain a compound system that has all the functional properties of each component. This technique is typically applied when the two components in question are extensions of a single version of a software system: e.g., version v_0 extends the original product v with feature f_0; version v_1 extends the original product v with feature f_1; we want the merged system to extend v with features f_0 and f_1. Program merging relies heavily on program specification technology, as well as the ability to perform the structural analysis (to identify what program parts must be merged) and the functional analysis (to identify how to merge program parts) of programs.

Program construction by parts consists in tackling a complex specification one component at a time, developing a partially defined solution for each component, then combining the partial solutions into a global solution for the aggregate specification. Program construction by parts is desirable whenever the specification at hand is too complex to be grasped in all its detail, and feasible whenever the specification at hand is structured as an aggregate of clearly defined subspecifications —where each subspecification represents a simple functional requirement. Program construction by parts relies heavily on the ability to represent arbitrarily nondeterministic specifications, the ability to represent arbitrarily specified programs, and the ability to combine two partially specified programs into a program that has the functional features of the two components.

These three techniques have a number of features in common, including their emphasis on arbitrarily abstract, arbitrarily nondeterministic specifications, as well as their concern for combining programs to produce larger programs. We feel that there is a lot to gain in exchanging solutions from these techniques, and that together they offer some potential for improving the state of the art in software analysis, development and maintenance.

1.3 Contributions

In this thesis, we define an approach for program construction by parts. Our approach is based on relational specifications, whereby a specification is described by a binary relation, and it is defined by the following premises:

- The set of relational specifications is naturally ordered by the refinement ordering, whose interpretation is the following: a specification S is a refinement of a specification S' if and only if any program correct with respect to S is also correct with respect to S'.
- The refinement ordering confers the set of specifications a lattice-like structure, where joins and meets can be defined in a formula-based fashion. Furthermore, the join of two specifications S and S' is the specification that carries all the functional features of S and all the functional features of S'.
- Complex specifications are naturally structured as the join of simpler subspecifications, where each subspecification captures one aspect of the problem at hand.
- Specifications are arbitrarily nondeterministic (unbounded nondeterminacy), and termination is implicit in a specification (total correctness and demonic semantics).

On the basis of these premises, we propose a program construction paradigm where specifications, structured as the join of subspecifications, are refined into programs in the following steps:

- solve subspecifications;
- merge solutions to subspecifications to derive a program satisfying the entire specification.

The first step is carried out in the traditional style of stepwise refinement. However, the laws involved are sometimes different from those usually found in refinement calculi (e.g., [41, 68, 83]). The second step is carried out using laws not found in a traditional refinement calculus. The main benefit of our approach is that it distributes the complexity of solving a specification over a set of smaller specifications.

To support this program construction paradigm, we have defined a language where specifications and programs can be represented. This language can represent specifications (structured as joins), programs (structured by means of traditional Pascal-like control structures) and intermediate designs (by means of a mixture of specification constructs and program constructs). Also, we have derived a set of rules for transforming a specification into a program, by changing a representation in this language from a specification structuring into a program structuring. The transformations we propose are correctness preserving, in the sense that they map one representation into a more refined representation. In most cases, they are optimal, in the sense that they provide the least refined decompositions, hence the easiest specifications to refine.

We have found that our paradigm also applies to the modification of programs, by observing that a modification may often be expressed as the join of the existing component with the specifications of the new features to add. Whether it is program adaptation, program reuse or version merging, the transformation rules we propose allow the designer to identify which sections of the existing code are preserved, and which sections need to be adapted.

It is important to establish the scope and limits of our research. Our objective is to study the *algorithmic refinement* of sequential imperative programs. We assume given a set of data types and type constructors which are properly axiomatized, and we work with the basic constructs of structured programming: sequence, alternation and iteration. Other aspects of program construction which are orthogonal to algorithmic refinement are not considered: data refinement, procedural abstraction and object-oriented extensions.

1.4 Outline

This thesis is divided in three parts. The first part provides the background and foundations for our research. Chapter 2 discusses existing paradigms of program construction. We propose a brief historical survey on program construction, then we review the most recent methods proposed. Chapter 3 introduces the mathematical notions supporting our research. We describe in turn the relevant concepts in order theory, Boolean algebra and relation algebra.

The second part presents our logic for program construction by parts. Chapter 4 introduces *Utica*, our language for program construction by parts. We propose a relational semantics for the language. Chapter 5 presents the transformation rules allowing the refinement of a Utica specification structured as a join into a join-free specification. In Chapter 6, we illustrate our program construction paradigm by a simple problem.

The third part extends our logic for program construction to make it suitable for program modification. Chapter 7 describes how the program construction by parts paradigm can apply to software adaptation, software reuse and software merging. Chapter 8 provides the mathematical foundations supporting program modification; a number of transformation rules for structure coercion, the essential refinement mechanism used in program modification, are presented. Chapter 9 is a case study which fully deploys our paradigm to solve more complex problems of construction and modification. Finally, we conclude by summarizing the contributions of this thesis, discussing their significance, and identifying future research directions.

Chapter 2

Program Construction Paradigms

Our objective in this chapter is to survey different approaches to program construction. We restrict ourselves to the class of imperative sequential programs. We first introduce the basic terminology. Then, we present a historical perspective of program construction and we outline some of the most recent methods for program construction. Finally, we discuss some existing approaches to program construction by parts.

2.1 Terminology and Notation

We consider Pascal-like program variables, and are interested in discussing specifications and programs that manipulate these variables. In this context, a *state* is a function that maps program variables to values; one can picture a state as a snapshot of the program variables. The *initial* state is the state before the beginning of a program execution. The *final* state is the state at the end of a program execution. A *specification* describes the expected final states (or output values) for a given initial state (input values) of a program. A specification is *nondeterministic* if it allows more than one final state for some initial state. We consider programs as a specific class of specifications. A program is a specification which is constructed only from statements that can be executed on a computer or from statements that can be compiled into executable statements.

We use the following logical operators which we list below by level of precedence, from the highest to the lowest. Let x be a variable, let p be a logical expression and let e be an expression:

1. $\neg$, $p[x\backslash e]$
2. $\wedge$
3. $\vee$
4. $\Rightarrow$, $\Leftarrow$
5. $\Leftrightarrow$
6. $\forall x : p$, $\exists x : p$
7. $\triangleq$

These logical operators have a lower precedence than any other operator. We let $p[x\backslash e]$ denote the syntactic replacement of free occurrences of x by expression e in statement p. The scope of quantifiers $\forall$ and $\exists$ extends as far as possible on the right. We use parentheses "(" and ")" to modify the scope or the precedence.

Throughout this document, we adhere to the following conventions.

- We use the symbol "." for function application and for composition of functions. Let A, B, C be sets. If f is a function of type $A \rightarrow B$ and g is a function of type $B \rightarrow C$, then $g.f$ is a function of type $A \rightarrow C$. For $a \in A$, we have $f.a \in B$ if $f.a$ is defined. The operator "." is associative, so we may write an expression like $g.f.a$ without parentheses.
- A function $f : A \rightarrow B$ is *total* if and only if $f.a$ is defined for all $a \in A$.
- The *identity* function of a set A is the function $i_A : A \rightarrow A$ such that $i_A.a = a$ for all $a \in A$.
- The *prerestriction* of a function f to a set A is defined by $f.i_A$.
- We denote by ϵ the unique function of type $\emptyset \rightarrow A$ for an arbitrary set A.
- Let A, B, C, D be sets such that $A \cap C = \emptyset$. Let $f : A \rightarrow B$ and $g : C \rightarrow D$ be two functions. The *union* $f \cup g$ is a function of type $A \cup C \rightarrow B \cup D$ defined as

  $$(f \cup g).a = \text{ if } a \in A \text{ then } f.a \text{ else } g.a \ .$$

- We use f, g, h for functions.
- We use i, j, k for indices.
- We use x, y, z for general variables (program variables or individual variables in a theory); variables may be primed (e.g., x').
- We use w to denote a finite list $x_1, \ldots, x_n$ of variables.
- We use w' for a finite list of primed variables.
- We use p, q for logical expressions or program statements.
- We use t for a logical expression containing no primed variable.
- We let $p[w\backslash E]$, where E is a finite list $e_1, \ldots, e_n$ of expressions, denote

  $$p[x_1\backslash e_1, \ldots, x_n\backslash e_n] \ .$$

- We let $w = w'$ denote $x_1 = x'_1 \wedge \ldots \wedge x_n = x'_n$.
- We let $\forall w : p$ denote $\forall x_1 : \ldots \forall x_n : p$.
- We may drop the colon between two consecutive quantifiers.

2.2 Overview of Related Work

The work presented in this thesis belongs to the world of state-based specifications for sequential imperative programming languages. In this domain, one may distinguish two schools of program construction on the basis of the structure of the specification. In the precondition-postcondition school, one specifies the behavior of a program using a pair of predicates, a predicate for the initial states and a predicate for the final states. In the functional-relational-predicative school, a single object describes the relationship between initial states and final states, either a function, a binary relation or a predicate. We briefly describe each school in turn, tracing their development in history.

2.2.1 The Assertion-Based Approach

The origins of the precondition-postcondition school go back to the late sixties. The initial works focused more on *program verification* than on program construction per se. In [42], Hoare introduced the concept of a specification as a pair consisting of a precondition and a postcondition. The interpretation of such a specification is as follows: if a program starts in a state satisfying the precondition and if the program *terminates*, then the final state must satisfy the postcondition. Hoare defines an axiomatic system for proving that a program is partially correct with respect to a specification: this system does not include rules for proving that a program terminates. Dijkstra extended Hoare's work by including termination in specifications [25, 28]. He defined the notion of program statements as *predicate transformers* and the *weakest precondition calculus*. Dijkstra's work was extended by Back [5, 6], Morris [69], Morgan [67, 68] and Nelson [74]. A notable extension is the inclusion of a *specification statement* in the calculus, which can be freely mixed with program statements in the development of a program. A logical conclusion of introducing the specification statement is the *miraculous statement*, a statement that cannot be satisfied by any executable program.

2.2.2 The Functional-Relational-Predicative Approach

One of the first contributions in the functional-relational-predicative school is found in the work of Mills [64]. He adopted the notion of a specification as a function between initial states and final states. A function f is interpreted as follows: if a program starts in a state s in the domain of f, then it must terminate in the state $f.s$. Mills' work highlighted the close correspondence between operators in the algebra of functions and constructs for structuring programs. He formalized the notion of stepwise refinement using equality: a specification refines another if their functions are the same. Unlike the precondition-postcondition approach, Mills' logic had no provision for nondeterminacy.

Mili [57] extended Mills' work to binary relations to allow for nondeterminacy. A relation R in Mili's framework has the following meaning: if a computation starts in a state s in the domain of relation R, then it must terminate in a state s' such that $(s, s') \in R$. If a computation starts in a state not in the domain of R, then any outcome is acceptable, including nontermination. Mili also formalized the notion of refinement to make it more general than Mills' notion of refinement. De Bakker [18, 19] uses relations, but with the intent of defining the semantics of nondeterministic program schemes, and of proving properties about them.

Hehner [37], extending his work with Hoare on a predicative semantics of CSP [36], proposed to use a single predicate with unprimed and primed free variables (which we denote by w and w' respectively) to designate the input and output values of a computation. Only *total* predicates are accepted as specifications. A predicate p is total if and only if, for every initial state s, there exists a final state s' such that $p[w\backslash s, w'\backslash s']$ holds. Nontermination is represented by relating an initial state to all final states.

In [44, 45], Hoare *et al* use binary relations in a way analogous to Hehner's predicative approach, except that they represent termination in a different manner: a specification is a total relation; the set of states of a program is extended by a fictitious state representing nontermination. A relation in this context has the following interpretation: if a program starts in the first state of a pair in the relation then it may end in the second state of the pair. A computation that may not terminate for an initial state is specified by relating the initial state to all possible final states, including of course the fictitious state.

Hehner proposed a new version of the predicative approach in [40, 41] by modeling termination with a time variable. Sekerinski [83] also proposed a new version of the predicative approach by taking an interpretation equivalent to Mills and Mili: if a computation starts in a state s for which the predicate p is defined (i.e., there exists a state s' such that $p[w\backslash s, w'\backslash s']$ holds), then it must terminate in a state s' such that $p[w\backslash s, w'\backslash s']$ holds. Nontermination is represented by undefinedness.

R.M. Dijkstra proposed a relational semantics of programs in [29]. He also uses a fictitious state $\perp$ to represent nontermination, but in a different manner than Hoare *et al* [45]. The relation R associated to a program p is defined as follows: $(s, s') \in R$ if and only if there exists a computation under the control of p originating from s and terminating in s'; $(s, \perp) \in R$ if and only if there exists a nonterminating computation under the control of p originating from s. In addition, the fictitious state $\perp$ is related only to himself. By contrast, Hoare *et al* represent a nonterminating initial state by relating it to all states.

The works of Jones [47] and Parnas [77] are very close to the relational-predicative approach. A specification is a pair consisting of a relation (or a predicate with primed and unprimed variables) and a *termination* set (or a predicate with unprimed variables only). The relation indicates the set of initial-final state pairs. The termination set represents the initial states in the relation for which a program must terminate. Abrial uses a distinct notation based on generalized substitutions (a form of extended assignment statements) and abstract machines [1, 2, 3]. For a good review of the different formalisms used in program specification and construction, the reader may consult [39, 49].

2.3 Refinement Calculi

Since its introduction by Wirth in 1971 [89], the stepwise refinement of specifications into programs has been formalized by a number of researchers into so-called *refinement calculi* [2, 5, 41, 68, 83]. Some of these calculi are now applied with success on industrial-size problems [17].

A refinement calculus consists of a formal language, a refinement relation and a set of transformation rules. The *formal language* embodies specification and programming constructs which can be freely mixed during the development of a program. Programs are considered to be an executable subset of specifications. The *refinement relation* is an ordering on specifications: a specification p is refined by a specification q if and only if any program correct with respect to q is correct with respect to p. A *transformation rule* maps a specification p into a specification q such that p is refined by q. Thus, transformations preserve correctness. A specification is refined into a program by successive applications of transformation rules. The development of a program p_n from a specification p_0 is a sequence of refinements:

$$p_0 \sqsubseteq p_1 \sqsubseteq \ldots \sqsubseteq p_n \ .$$

We read $p \sqsubseteq q$ as "p is refined by q". The transitivity of the ordering $\sqsubseteq$ implies that $p_0 \sqsubseteq p_n$. The resulting program p_n is correct by construction. It is the programmer's responsibility to select transformation rules, and to satisfy proof obligations associated with a rule.

In a refinement step, a specification p is usually decomposed into specifications p_1 and p_2 such that $p \sqsubseteq p_1 \Phi p_2$, where Φ is an operator of the language. In turn, p_1 and p_2 are decomposed until executable elementary statements are obtained. To foster separation of concerns, a refinement calculus must allow the designer to refine p_1 without any regards to p or p_2, and, similarly, to refine p_2 without any regards to p or p_1; in addition, correctness must be preserved in the process. A sufficient condition to achieve this goal is to have *monotonic* specification operators* with respect to the refinement relation.

2.3.1 Refinement of Predicate Pairs

The specification language in the refinement calculus of Morgan [68] is Dijkstra's language of guarded commands [25] extended with a specification statement. Similar calculi were proposed by Back [5, 6], Morris [69] and Nelson [74]. The language has an axiomatic semantics based on predicate transformers.

2.3.1.1 Specifications

A specification statement is a structure

$$w : [\mathit{pre}, \mathit{post}] \ .$$

The frame w is a list of variables that a program is allowed to modify during execution. Variables not mentioned in w must be preserved by the execution of a program. Predicates *pre* and *post* are called the specification's *precondition* and *postcondition* respectively. The interpretation of a specification is the following: if a program starts in a state satisfying the precondition then it must terminate in a state satisfying the postcondition. The free variables in the predicates *pre* and *post* are the program variables. Assume in the following example that variables x, y are of type **Integer**:

$$y : [x \geq 0, y^2 = x] \ .$$

A program correct with respect to the specification above must preserve the value of x and set y to the square root of x if $x \geq 0$. If the initial value of x is less than 0, the program may do anything, including not terminating. This specification is nondeterministic, because for a given x there exist several values (2) of y satisfying the postcondition.

Some specification statements cannot be refined into an executable program, because they require a condition on the final state which cannot be satisfied for some initial state. For example, there is no final state satisfying the postcondition in the following specification if the initial value of x is less than 0:

$$y : [\mathbf{true}, y^2 = x] \ .$$

Such specifications are said to be *infeasible* (or *miraculous*). Another specification which is clearly infeasible is the *magic* specification:

$$w : [\mathbf{true}, \mathbf{false}] \ .$$

*Monotonicity is formally defined in Chapter 3

Special variables are sometimes required to specify a computation. For instance, consider a requirement where the final value of y must be the square root of the initial value of y. The postcondition must refer to the initial value of y, but variable y in the postcondition stands for the final value only. This issue has been known for a long time. A solution is to introduce an additional variable, called a *logical constant*, which is not considered as a regular program variable. It is implicitly existentially quantified. Its value is preserved by the execution of the program, since it is not in the frame and it is not a program variable. Using the logical constant Y, the requirement we just considered is specified as:

$$y : [y \geq 0 \wedge y = Y, y^2 = Y] \ .$$

The other constructs of Morgan's language are the guarded commands with new constructs to declare local variables and logical constants. Let t be a logical expression, let p, q be statements, let x be a variable, let w be a list of variables and let T be a type. The following list describes the syntax of the main constructs.

```
*multiple assignment*
  $x_1, \ldots, x_n := e_1, \ldots, e_n$
*sequential composition*
  $p; q$
*alternation*
  **if**
    $t_1 \rightarrow p_1,$
    $\ldots$
    $t_n \rightarrow p_n,$
  **fi**
*iteration*
  **do**
    $t_1 \rightarrow p_1,$
    $\ldots$
    $t_n \rightarrow p_n,$
  **od**
*local variable*
  $|[\textbf{var}\ w : T \bullet p]|$
*logical constant*
  $|[\textbf{con}\ w : T \bullet p]|$ .
```

The language also supports recursion, procedures and modules. An **if** statement is defined for the set of states satisfying *guards* $t_1, \ldots, t_n$.

2.3.1.2 Correctness

The notion of refinement is defined in terms of the notion of *correctness*. We define correctness in this section, and then we introduce the notion of refinement in the next section.

A program p satisfies (or is correct with respect to) a specification $w : [pre, post]$ if the execution of p in a state satisfying *pre* terminates in a state satisfying *post*. To determine correctness, one can proceed as follows:

1. compute from p and *post* the *weakest*[†] precondition pre' such that the execution of p from a state satisfying pre' terminates in a state satisfying *post*;

[†]The predicate satisfied by the largest set of states.

2. if $pre \Rightarrow pre'$ holds, that is, if a state satisfying *pre* also satisfies pre', then conclude that the program satisfies the specification; otherwise it does not.

Dijkstra [25] axiomatized the calculation of the weakest precondition for the language of guarded commands using the *wp* operator. The expression:

$$wp(p, post)$$

stands for the weakest precondition for statement p and postcondition *post*. The definition of *wp* is compositional, that is, the weakest precondition of a compound statement is defined in terms of the weakest preconditions of the components. Morgan formally defined the semantics of the specification statement using the *wp* operator:

$$wp(w : [pre, post], post') \mathrel{\hat{=}} pre \wedge (\forall w : post \Rightarrow post') \ .$$

2.3.1.3 Refinement

We mentioned in Chapter 1 that a specification p is refined by a specification q if any program correct with respect to q is also correct with respect to p. In the weakest precondition semantics, refinement is defined as follows:

$$p \sqsubseteq q \mathrel{\hat{=}} \forall post : wp(p, post) \Rightarrow wp(q, post) \ .$$

This definition provides that if p terminates in a state satisfying *post* for an initial state s, then q also terminates for s in a state satisfying *post*. In addition, q may terminate in a state satisfying *post* for some initial state in which p does not. In other words, q is stronger than p, because it can establish the same postcondition for a larger set of states. All the compound constructs in Morgan's language are monotonic with respect to the refinement relation. The specification *magic* refines any other specification.

$$\begin{array}{ll} & p \sqsubseteq w : [\mathbf{true}, \mathbf{false}] \\ \Leftrightarrow & \qquad \{ \text{ definition of } \sqsubseteq \} \\ & \forall post : wp(p, post) \Rightarrow wp(w : [\mathbf{true}, \mathbf{false}], post) \\ \Leftrightarrow & \qquad \{ \text{ semantics of } magic \} \\ & \forall post : wp(p, post) \Rightarrow \mathbf{true} \\ \Leftrightarrow & \qquad \{ \text{ propositional calculus laws } \} \\ & \mathbf{true} \ . \end{array}$$

A refinement sequence may well lead to an infeasible specification like *magic*, so the programmer must be careful in applying transformation rules. The least refined specification is *abort*, which is defined as $w : [\mathbf{false}, \mathbf{true}]$. This specification does not need to terminate from any initial state. It is refined by any specification. Similarly, a specification of the form $w : [\mathbf{false}, post]$ is refined by any specification.

2.3.2 Refinement of Predicates with Timing Requirements

The refinement calculus of Hehner [40, 41] differs significantly in terms of the specification model from Morgan's: Hehner uses a single predicate on primed and unprimed variables; termination is explicitly stated in a specification using a time variable.

2.3.2.1 Specifications

A specification is a predicate p with free unprimed variables w and primed variables w'. List w represents the input values of a computation, and list w' represents the output values of a computation. A computation satisfies a specification p if for any input s it delivers an output s' such that $p[w\backslash s, w'\backslash s']$ holds. However, a computation may take an *infinite* time to deliver its result[‡]. One can use a time variable to specify that a result must be delivered within a definite time. The type of the time variable is any number system extended with ∞ as the greatest element. Assume in the following specification that variables x, y are of type **Integer**:

$$x \geq 0 \Rightarrow y'^2 = x \ \ .$$

A computation satisfying the specification above must deliver a final value of y equal to the square root of the initial value of x. The final value of x is arbitrary since the specification does not mention it. The computation of y may take an infinite time, because there is no time variable in this specification.

Let t and t' be time variables representing respectively the starting time and ending time of a computation . A specification p is *implementable* if and only if

$$\forall w \exists w' : p \land t \leq t' \ \ .$$

A specification is implementable if it allows an output value for any input value, and if it does not decrease time. The following specification is not implementable:

$$y'^2 = x \ \ .$$

For $x < 0$, there exists no value for y' such that $y'^2 = x$. Curiously, the following specification is also not implementable:

$$(x \geq 0 \Rightarrow y'^2 = x) \land t' < \infty \ \ .$$

For $t = \infty$, there exists no t' such that $t \leq t' \land t' < \infty$. The only value of t' satisfying the first conjunct is ∞, but $t' = \infty$ does not satisfy the second conjunct since ∞ is the greatest element. The following specification is implementable and it requires the computation of the square root in a finite time (exactly one unit of time)

$$(x \geq 0 \Rightarrow y'^2 = x) \land t' = t + 1 \ \ .$$

The other constructs in the language are conveniently defined in terms of specifications. Let t be a logical expression, let p, q be statements, let x be a variable, and let w be a list of variables. The next list introduces the other operators with their semantics following the "$\triangleq$" symbol.

preserve the state
$\quad \mathbf{ok} \triangleq w = w'$
assignment
$\quad x := e \triangleq \mathbf{ok}[x\backslash e]$
sequential composition
$\quad p; q \triangleq \exists w'' p[w'\backslash w''] \land q[w\backslash w'']$
alternation
$\quad \mathbf{if}\ t\ \mathbf{then}\ p\ \mathbf{else}\ q \ \triangleq t \land p \lor \neg t \land q$
local variable
$\quad \mathbf{var}\ x : T; p \triangleq \exists x, x' : x \in T \land x' \in T \land p$

[‡]Hence the usage of the verb "deliver" instead of the verb "terminate" in the description of Hehner's calculus.

The language allows recursion and it contains other constructs for concurrency and communication. Of course, one can compose all these statements with the logical operators $\neg, \wedge, \vee, \Rightarrow$ and $\Leftrightarrow$. We will discuss recursion after we have introduced refinement in the next section.

2.3.2.2 Refinement

Refinement in Hehner's calculus is simple and attractive: refinement is logical implication. A specification p is refined by a specification q (noted[§] $p \sqsubseteq q$) if and only if:

$$\forall w, w' : p \Leftarrow q \ \ .$$

This definition provides that a computation satisfying q delivers a final state satisfying p. In other words, any computation satisfying q will also satisfy p, provided that p and q are implementable. Note that an implementable specification can be refined by an unimplementable specification, like in Morgan's calculus. For instance, the specification **false** refines any other specification since the following holds for any q:

$$\forall w, w' : q \Leftarrow \mathbf{false} \ \ .$$

Obviously, **false** is not implementable. The specification **true** is refined by any other specification. Finally, every operator of the language is monotonic with respect to refinement.

2.3.2.3 Recursion and Timing

It is possible to express recursion by having the same specification appearing on both sides of $\sqsubseteq$. In the following refinement

$$p \sqsubseteq q; p \ \ ,$$

the symbol $\sqsubseteq$ acts like a procedure definition symbol, and specification p occurring on the right-hand side of $\sqsubseteq$ acts like a recursive call. The next example is a recursive implementation of the factorial specification. Let p be the specification $x \geq 0 \Rightarrow y' = x!$ in the following recursion:

```
$p \sqsubseteq$
    if $x = 0$
        then $y := 1$
        else $p$  .
```

One can easily prove that what appears on the right-hand side of $\sqsubseteq$ implies the left-hand side. Therefore, the recursion is a refinement of the factorial specification, and it is also a program. Unfortunately, this program does not terminate for $x \neq 0$. Our specification p does not mention the time variable, so this program is still a refinement. Hehner identifies two ways of handling time: real time and recursive time. The former is used when one is interested in real-time programming. The latter is used when one is interested only in proving termination or in calculating the algorithmic complexity of an implementation. To obtain the real execution time of a program, one substitutes for each statement p (assignment, alternation or recursive call) the sequence $t := t + a$; p, where a is the actual execution time of p. In recursive time, one simply inserts $t := t + 1$ before each recursive call.

One must be cautious in specifying time. We saw earlier that adding the conjunct $t' < \infty$ made a specification not implementable. The reader may be tempted by adding instead the conjunct

[§]Hehner uses a larger $\Leftarrow$ as refinement symbol. We use $\sqsubseteq$ to maintain some consistency in the presentation of the various refinement calculi.

$$t < \infty \Rightarrow t' < \infty \ .$$

This conjunct states that if execution starts before infinity, then it must terminate in a finite time. However, this is still unsatisfactory. The factorial specification, adjusted in this manner, admits for refinement a non-terminating recursion. Let p be the following specification:

$$x \geq 0 \Rightarrow y' = x! \wedge (t < \infty \Rightarrow t' < \infty) \ .$$

The reader may check that the following non-terminating recursion is a valid refinement:

```
$p \sqsubseteq$
    **if** $x = 0$
        **then** $y := 1$
        **else** $t := t + 1;\ p$   .
```

As this example suggests, a designer must find an upper bound for execution time in order to prove termination; the predicate

$$t < \infty \Rightarrow t' < \infty$$

is insufficient.

2.3.3 Refinement of Predicates with Implicit Termination

Sekerinski [83] uses predicates like Hehner for specification, but with a different interpretation: termination is implicitly represented by undefinedness. This interpretation is similar to the ones of Mills and Mili, except that the latter uses functions or relations instead of predicates.

2.3.3.1 Specifications

A specification is a predicate p with free unprimed variables w and free primed variables w'. List w represents the input values of a computation, and list w' represents the output values of a computation. A specification is interpreted as follows: if a computation starts in a state s for which the predicate p is defined (i.e., there exists a state s' such that $p[w\backslash s, w'\backslash s']$ holds), then it must terminate in a state s' such that $p[w\backslash s, w'\backslash s']$ holds. If a computation starts in a state for which the predicate is not defined, then any result is acceptable, including nontermination. In this calculus, any predicate is a specification. Assume in the following specification that variables x, y are of type **Integer**:

$$y'^2 = x \ .$$

The specification above is equivalent to the specification

$$x, y : [x \geq 0 \wedge x = X, y^2 = X]$$

in Morgan's calculus, and to the specification

$$x \geq 0 \Rightarrow y'^2 = x \wedge t' \leq t + b(x)$$

in Hehner's calculus, where $b(x)$ is some time bound for the computation of the square root. Sekerinski's specification is more compact, because termination and the precondition are implicit in the specification. The set of states for which a computation is required to terminate (precondition or domain) may be derived from the specification. The domain Δp of a specification p is given by $\exists w' : p$. For example, the domain of the square root specification is $x \geq 0$.

The other operators in the language of Sekerinski, with their semantics following the "$\triangleq$" symbol, are:

preserve the state

$\textbf{skip} \triangleq w = w'$

assignment

$x := e \triangleq \textbf{skip}[x\backslash e]$

sequential composition

$p; q \triangleq (p \triangleright (\Delta q)) \wedge (p \circ q)$, where

$p \triangleright t \triangleq \forall w' : p \Rightarrow t[w'\backslash w]$,

$(p \circ q) \triangleq \exists w'' : p[w'\backslash w''] \wedge q[w\backslash w'']$

alternation

$\textbf{if } t \textbf{ then } p \textbf{ else } q \textbf{ end} \triangleq t \wedge p \vee \neg t \wedge q$

iteration

$\textbf{while } t \textbf{ do } p \textbf{ end} \triangleq \ldots$

nondeterministic choice

$p \sqcap q \triangleq \Delta p \wedge \Delta q \wedge (p \vee q)$

local variable

$\textbf{var } x : p \triangleq (\forall x : \Delta p) \wedge (\exists x, x' : p)$

The definition of the **while** statement is not provided, because it requires foundations too complex to describe at this point. A nondeterministic choice $p \sqcap q$ behaves like p or like q. It is defined on the intersections of the domains of p and q. A computation satisfies a nondeterministic choice if, when started in a state in the domain of p and q, it terminates in a state satisfying p or q. It is equivalent to $p \vee q$ in Hehner's calculus. Note that the definition of operators are more complex than Hehner's definition. This is due to the fact that termination is implicit in a specification.

2.3.3.2 Refinement

A specification p is refined by a specification q (noted $p \sqsubseteq q$) if and only if

$$\forall w, w' : \Delta p \Rightarrow \Delta q \wedge (q \Rightarrow p) \ .$$

This definition provides that a computation satisfying q satisfies p as well. Indeed, if p is defined for an initial state, then q is also defined ($\Delta p \Rightarrow \Delta q$), and on the domain of p any final state satisfying q also satisfies p ($\Delta p \Rightarrow (q \Rightarrow p)$). Again, one may observe that the definition of refinement is more complex than Hehner's definition.

The least refined specification is **false**, contrary to Hehner's calculus where it is the most refined specification. There is no most refined specification in Sekerinski's calculus. The specification **true** refines any specification with unprimed variables only, and it is refined by any total specification. The maximal specifications (i.e., specifications which admit only logically equivalent specifications as refinements) are total deterministic specifications.

2.4 Program Construction by Parts

An essential premise of program construction by parts is the use of a join operator to structure specifications. Several authors have advocated that the join operator is a most useful tool in the derivation of complex specifications [13, 40, 45]: the join operator fosters separation of concerns by allowing the specifier to consider one aspect at a time, independently of other aspects. We emphasize the word "independent", because a program satisfying the join of subspecifications must satisfy each subspecification individually. Hoare *et al* in [45] also argue that join increases the clarity of specifications.

Hoare *et al* [45] identified some basic properties of join like idempotence, commutativity, associativity and absorption. They also mention distributivity over usual programming constructs like sequential composition, alternation, iteration and nondeterministic choice, but for a very limited case of join (*directed set* of subspecifications with bounded nondeterminacy). Their body of laws falls short of providing means for transforming join-structured specifications into executable programs.

The early work of Hehner [37] also mentions the use of join as a specification operator. His subsequent work [40, 41] includes interesting laws of refinement for join-structured specifications. Hehner calls these laws "refinement by parts"; hence we use the expression "construction by parts" to denote our approach to program construction. In Hehner's context, as we have seen in Section 2.3.2, specifications are predicates, termination is prescribed using a time variable, and join is logical conjunction.

Morgan and Gardiner [31] use a join statement for data refinement and for the definition of logical constants. They do not use join as a specification structuring device nor do they study the refinement of join-structured specifications.

Finally, the work of von Wright also mentions a join operator. In [88], he reconstructs the refinement calculus of Back from elementary primitives. The emphasis of this work is more on the definition of a language than on the definition of a refinement calculus. Program construction by parts is not studied as such. The semantics of the language is given by predicate transformers, and miraculous specifications are allowed.

2.5 Discussion and Assessment

Our objective is to define a calculus that adequately supports the join operator by providing transformations rules for the refinement of join-structured specifications. In the sequel, we highlight the differences between our calculus and the others described in the previous sections.

Our work differs from that of Hoare *et al* [45] by taking a different interpretation of relations (nontermination represented by undefinedness, versus nontermination represented by relating an initial state to every final state, including the fictitious state). We extend their work by allowing unbounded nondeterminacy in any specification, by generalizing laws of the join operator to arbitrary sets of specifications, instead of directed sets of specifications, and by studying the transformation of join-structured specifications.

Our work differs from the work of Hehner by using a different formalism (the algebra of relations, versus the calculus of predicates), by taking a distinct interpretation of specifications (implicit termination represented by definedness versus explicit termination represented by a time variable), and by admitting any relation as a specification instead of restricting to total specifications. We extend his work by proposing additional laws for program construction by parts.

Our work differs from the work of von Wright [88], Morgan and Gardiner [31] by using a different semantics (denotational semantics based on relations, versus axiomatic semantics based on predicate transformers), by not allowing miraculous specifications, and by studying the transformation of join-structured specifications.

Chapter 3

Mathematical Foundations

This chapter is devoted to the mathematical foundations of program construction by parts. We present relevant results for building our refinement calculus. In Section 3.1, we introduce ordered sets, which are used for the formal definition of refinement, of the join operator and of iterative constructs. In Section 3.2, we present Boolean algebras, which are a prerequisite for the presentation of relation algebra, in Section 3.3. Section 3.4 introduces the refinement ordering on relations. Finally, the last section is devoted to the *prerestriction ordering*, which is used to prove properties about iterative constructs.

3.1 Ordered Sets

Our definitions are taken from [16, 82].

3.1.1 Ordering

An *ordering* on a set X is a binary relation $\leq$ (*less than*[*]) on X that is reflexive, transitive and antisymmetric. An *ordered set* is a pair $(X, \leq)$. We may refer to X as an ordered set if there is no ambiguity on the ordering. Let x and y be elements of an ordered set. We may write $x < y$ (*strictly less than*) if $x \leq y$ and $x \neq y$. We may use $x \leq y$ and $y \geq x$ interchangeably.

3.1.2 Lower and Upper Bounds

Let X be an ordered set. A *lower bound* of $Y \subseteq X$ is an element $x \in X$ such that, for all $y \in Y$, $x \leq y$. The set of lower bounds of Y is denoted by Y^l. The *greatest lower bound* of Y, if it exists, is an element $x \in Y^l$ such that, for all $x' \in Y^l$, $x' \leq x$. The notions of *upper bound* and *least upper bound* are defined dually: x is an upper bound of Y if and only if $y \leq x$ for all $y \in Y$; the set of upper bounds of Y is denoted by Y^u; $x \in Y^u$ is the least upper bound of Y if and only if $x \leq x'$ for all $x' \in Y^u$. The greatest lower bound and least upper bound of Y are denoted by $\bigwedge Y$ and $\bigvee Y$ respectively. When $x = \bigwedge Y$ and $x \in Y$, we say that x is the *least* element of Y. Dually, if $x = \bigvee Y$ and $x \in Y$, then we say that x is the *greatest* element of Y.

[*]The reader may be accustomed to the denomination *less than or equal to*. We favor *less than* for its shorter form.

3.1.3 Special Classes of Ordered Sets

A *lattice* is an ordered set X such that the greatest lower bound and the least upper bound of any *non-empty finite* subset Y of X exist. A lattice is said to be *complete* if and only if the greatest lower bound and the least upper bound of any subset Y of X exist. In a complete lattice, there exist a least element and a greatest element. A $\wedge$-*semilattice* is an ordered set X where the greatest lower bound of any non-empty finite subset exists. Dually, a $\vee$-*semilattice* is an ordered set X where the least upper bound of any non-empty finite subset exists. A $\wedge$-semilattice ($\vee$-semilattice) is *complete* if the greatest lower bound (least upper bound) exists for any non-empty subset. A complete $\wedge$-semilattice ($\vee$-semilattice) has a least (greatest) element. A *chain* is an ordered set X such that, for any $x, x' \in X$, $x \leq x'$ or $x' \leq x$. A *CPO* (complete partially ordered set) is an ordered set X where any chain Y of X has a least upper bound. A CPO has a least element since the empty chain must have a least upper bound.

3.1.4 Fixpoint Theory

Let X be an ordered set and $f : X \to X$ be a function. A *fixpoint* of f is an element $x \in X$ such that $x = f.x$. The *least fixpoint* of f, if it exists, is the least element of the set of fixpoints of f. The *greatest fixpoint* of f, if it exists, is the greatest element of the set of fixpoints of f. We say that f is *monotonic* with respect to $\leq$ if and only if $x \leq x' \Rightarrow f.x \leq f.x'$; we say that f is *antitonic* with respect to $\leq$ if and only if $x \leq x' \Rightarrow f.x' \leq f.x$. Fixpoints are used to define the semantics of iterative constructs. The next three theorems provide the existence of fixpoints for monotonic functions in three different kinds of ordered sets.

1 Theorem. Knaster-Tarski Fixpoint Theorem [87]. *Let X be a complete lattice and let $f : X \to X$ be a monotonic function. Then f has a least fixpoint and a greatest fixpoint given by*

$$\bigwedge\{x \in X | f.x \leq x\} \text{ and } \bigvee\{x \in X | x \leq f.x\} \text{ respectively.} \qquad \Box$$

The next theorem states the existence of a least fixpoint for an object weaker than the complete lattice of the previous theorem, but with an additional condition on the monotonic function.

2 Theorem. *Let X be a complete $\wedge$-semilattice and $f : X \to X$ be a monotonic function. If $\bigwedge\{x \in X | f.x \leq x\}$ exists, then it is the least fixpoint of f.* $\Box$

The next theorem states the existence of a least fixpoint for another object weaker than a complete lattice. There is no necessary condition on the monotonic function, but the theorem does not provide a characterization of the least fixpoint.

3 Theorem. Bourbaki [14], Marchowsky [54]. *Let X be a CPO and $f : X \to X$ be a monotonic function. Then f has a least fixpoint.* $\Box$

3.2 Boolean Algebra

In the previous section, we provided an order-theoretic definition of a lattice. A lattice can also be seen as an algebraic object. Boolean algebras form a special class of lattices.

4 Definition. A lattice is a structure $(X, \vee, \wedge)$ over a non-empty set X. The operations $\vee$ and $\wedge$ are called *join* and *meet* respectively. Let x, y and z be elements of X. The following laws are satisfied.

$$\begin{array}{llll} \text{(a)} & (x \vee y) \vee z = x \vee (y \vee z) & \text{(d)} & (x \wedge y) \wedge z = x \wedge (y \wedge z) \\ \text{(b)} & x \vee y = y \vee x & \text{(e)} & x \wedge y = y \wedge x \\ \text{(c)} & x \vee (x \wedge y) = x & \text{(f)} & x \wedge (x \vee y) = x \ . \qquad \square \end{array}$$

5 Definition. In a lattice, we define the binary relation $\subseteq$ as

$$x \subseteq y \Leftrightarrow x \wedge y = x$$

or, equivalently,

$$x \subseteq y \Leftrightarrow x \vee y = y \ .$$

The relation $\subseteq$ is reflexive, transitive and antisymmetric. Whence, it is an *ordering.* If $x \subseteq y$ and $x \neq y$, then we write $x \subset y$. In a lattice, the greatest lower bound of $\{x, y\}$ is given by $x \wedge y$, and the least upper bound is given by $x \vee y$. □

6 Definition. A lattice is said to be *distributive* if and only if the following laws are satisfied.

$$\begin{array}{ll} \text{(a)} & x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z) \\ \text{(b)} & x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z) \ . \qquad \square \end{array}$$

7 Definition. An element $\emptyset$ of a lattice X is called *zero* (or least element) if, for any other element x of the lattice, the following hold:

(a) $\emptyset \vee x = x$ or, equivalently, $\emptyset \subseteq x$.

An element L of a lattice X is called *universal* (or greatest element) if, for any other element x of the lattice, the following hold:

(b) $L \wedge x = x$ or, equivalently, $x \subseteq L$. □

8 Definition. An element a of a lattice X with zero element $\emptyset$ is an *atom* if $a \neq \emptyset$ and there exists no element $x \in X$ such that $\emptyset \subset x \subset a$. The lattice X is *atomic* if for every $x \in X$, $x \neq \emptyset$, there exists an atom a such that $a \subseteq x$. □

9 Definition. In a lattice with zero element $\emptyset$ and universal element L, an element c is said to be the *complement* of an element x if

(a) $x \vee c = L$ (b) $x \wedge c = \emptyset$.

A lattice where there exists at least one complement for each element is called *complementary.* □

In a distributive lattice, complements are unique. In a complementary, distributive lattice, we denote the complement of an element x by $\overline{x}$.

10 Definition. A *Boolean lattice* is a structure $\mathcal{B} = (X, \vee, \wedge, \overline{\ }, \emptyset, L)$ where $(X, \vee, \wedge)$ is a complementary, distributive lattice. Complementation, zero element and universal element are denoted by $\overline{\ }$, $\emptyset$ and L respectively. A Boolean lattice is also called a *Boolean algebra.* □

11 Definition. A lattice $(X, \vee, \wedge)$ is said to be *complete* if and only if the join and meet of arbitrary subsets of X exist. $\square$

The following properties hold in a complete atomic Boolean algebra.

12

(a)	$x \vee x = x \wedge x = x$	(b)	$x \subseteq y \;\Rightarrow\; x \vee z \subseteq y \vee z$
(c)	$x \subseteq y \;\Rightarrow\; x \wedge z \subseteq y \wedge z$	(d)	$x_i \subseteq x_1 \vee x_2$ $(i \in 1..2)$
(e)	$x_1 \wedge x_2 \subseteq x_i$ $(i \in 1..2)$	(f)	$x_1 \subseteq u$ and $x_2 \subseteq u \;\Leftrightarrow\; x_1 \vee x_2 \subseteq u$
(g)	$u \subseteq x_1$ and $u \subseteq x_2 \;\Leftrightarrow\; u \subseteq x_1 \wedge x_2$	(h)	$x \subseteq y \;\Leftrightarrow\; L \subseteq \overline{x} \vee y$
(i)	$x \subseteq y \;\Leftrightarrow\; x \wedge \overline{y} \subseteq \emptyset$	(j)	$x \subseteq y \vee z \;\Leftrightarrow\; x \wedge \overline{y} \subseteq z$
(k)	$x \subseteq y \vee z \;\Leftrightarrow\; \overline{y} \subseteq \overline{x} \vee z$	(l)	$x \subseteq y \vee z \;\Leftrightarrow\; L \subseteq \overline{x} \vee y \vee z$
(m)	$\overline{\overline{x}} = x$	(n)	$x \subseteq y \;\Rightarrow\; \overline{y} \subseteq \overline{x}$
(o)	$\overline{\bigvee_j x_j} = \bigwedge_j \overline{x_j}$	(p)	$\overline{\bigwedge_j x_j} = \bigvee_j \overline{x_j}$
(q)	$x \wedge \bigvee_j y_j = \bigvee_j x \wedge y_j$	(r)	$x \vee \bigwedge_j y_j = \bigwedge_j x \vee y_j$.

3.3 Relation Algebra

Due to their simplicity and their ability to represent nondeterminacy, relations have been used by a number of researchers for representing the specification of software requirements [7, 11, 18, 48, 61, 66, 84, 90]: To represent a specification, one provides a relation containing the input/output pairs that the user considers correct. We consider two axiomatizations of relations, which correspond to two distinct levels of abstraction: the *(abstract) relation algebra* and the *algebra of (concrete) relations*. The first definition, which views the set of relations as an algebraic structure, will be useful primarily for mathematical calculations (definitions, propositions, proofs, etc.); the second definition, which views relations as sets of pairs, will be useful primarily for defining the semantics of programming languages and for illustrative purposes.

3.3.1 Abstract Relation Algebra

Homogeneous relation algebras have been presented first in [86]; their axiomatization is carried out in [15]. Our definition comes from [82], as do most of the other relational notions presented in this section. An interesting presentation of relation algebras can also be found in [8]. See also [52, 79] for a historical perspective.

13 Definition. A *homogeneous relation algebra* is a structure $(\mathcal{R}, \cup, \cap, ^{-}, ^{\frown}, \circ, \emptyset, L)$ over a non-empty set $\mathcal{R}$ of elements, called *relations*. The following conditions are satisfied.

1. $(\mathcal{R}, \cup, \cap, ^{-}, \emptyset, L)$ is a complete atomic Boolean algebra.
2. For every relation R there exists a *converse* relation $\widehat{R}$ (we will write $(R)^{\frown}$ rather than $\widehat{(R)}$ for parenthesized expressions).
3. *Composition*, denoted by $\circ$, is an associative operation with an identity element, which is denoted by I.
4. The Schröder rule is satisfied: $P \circ Q \subseteq R \Leftrightarrow \widehat{P} \circ \overline{R} \subseteq \overline{Q} \Leftrightarrow \overline{R} \circ \widehat{Q} \subseteq \overline{P}$.
5. $L \circ R \circ L = L$ holds for every $R \neq \emptyset$ (Tarski rule). $\square$

The precedence of the relational operators, from highest to lowest, is the following: $^{-}$ and $^{\frown}$ bind equally, followed by $\circ$, followed by $\cap$ and finally by $\cup$. The scope of $\bigcup_i$ and $\bigcap_i$ goes to the right as far as is possible. From now on, the composition operator symbol $\circ$ will be omitted (that is, we write QR for $Q \circ R$).

3.3.2 Algebra of Concrete Relations

A (homogeneous) relation R on a set S is a subset of the Cartesian product $S \times S$. We denote the *universal relation* $(S \times S)$ by L, the *empty relation* by $\emptyset$ and the *identity relation* $(\{(s,s')|s' = s\})$ by I. Also, the complement $\overline{R}$ is the set difference $L \backslash R$; the relational product RR' is defined by $\{(s,s')|\exists t : (s,t) \in R \wedge (t,s') \in R'\}$; the i^{th} relative power of relation R is denoted by R^i and represents the product of R by itself i times (with $R^0 \triangleq I$); finally, the converse $\widehat{R}$ is $\{(s,s')|(s',s) \in R\}$.

We admit without proof that the algebra of concrete relations, provided with the set-theoretic operations of union, intersection and complement, and the relational operations of product and converse, defines a *relation algebra* in the sense of Definition 13 above [82]. Hence we may apply results of relation algebras to algebras of concrete relations.

We may use the following operators on concrete relations, especially for providing set-theoretic interpretations of abstract relational operators and formulas. The domain $dom(R)$ of a relation R is the set $\{s|\exists s' : (s,s') \in R\}$; the range $rng(R)$ of a relation R is the set $\{s'|\exists s : (s,s') \in R\}$; the images of element s by relation R is denoted by[†] $R.s$ and defined by $\{s'|(s,s') \in R\}$.

3.3.3 Properties of Relations

14 Definition. A relation R is said to be *reflexive* if and only if $I \subseteq R$; a relation is said to be *transitive* if and only if $RR \subseteq R$; a relation R is said to be *deterministic* if and only if $\widehat{R}R \subseteq I$; a relation R is said to be *injective* if and only if $\widehat{R}$ is deterministic; a relation R is said to be *total* if and only if $L = RL$ (equivalently, $I \subseteq R\widehat{R}$); a relation R is said to be *surjective* if and only if $\widehat{R}$ is total. Also, we say that a relation t is a *left vector* if and only if $t = tL$; the converse $\widehat{t}$ of a vector t is called a *right vector*. We say that a relation c is a *rectangle* if and only if it is the intersection of a left vector by a right vector. The *transitive closure* of relation R is denoted by R^+ and defined by $\bigcup_{i>0} R^i$; the *reflexive transitive closure* (or simply *closure* for short) of relation R is denoted by R^* and is given by $R^+ \cup I$ or, equivalently, $\bigcup_{i \geq 0} R^i$. $\square$

These definitions are taken from [34]. In an algebra of concrete relations over a set S, a left vector is a relation of the form $T \times S$, where $T \subseteq S$. A rectangle is a relation of the form $T \times T'$, where $T, T' \subseteq S$. For example, if

$$S \triangleq \{0,1,2\}$$

and

$$T \triangleq \{0,1\} \ ,$$

then

$$t \triangleq T \times S = \{(0,0),(0,1),(0,2),(1,0),(1,1),(1,2)\}$$

is a left vector. The set of left vectors constitutes a complete Boolean subalgebra of the Boolean algebra $(\mathcal{R}, \cup, \cap, ^{-}, \emptyset, L)$ (see [82], where this result is proved).

Given a left vector t and a relation R, the relation $t \cap R$ is said to be the *prerestriction* of R to t and $\widehat{t} \cap R$ is said to be the *postrestriction* of R to t. For example, with

[†]This notation is an extension of the function application operator "." to relations. Note that if F is the graph of function f, then $F.x = \{f.s\}$.

$$R \triangleq \{(0,1),(0,2),(2,1)\}$$

and the left vector t of the previous paragraph, the prerestriction and postrestriction of R to t are

$$\{(0,1),(0,2)\} \text{ and } \{(0,1),(2,1)\} \ ,$$

respectively. In addition, for any R, the relations RL and $\widehat{R}L$ are left vectors characterizing the *domain* and *range* of R, respectively. For the relation R above, these are

$$RL = \{(0,0),(0,1),(0,2),(2,0),(2,1),(2,2)\}$$

and

$$\widehat{R}L = \{(1,0),(1,1),(1,2),(2,0),(2,1),(2,2)\} \ .$$

From the definition of a relation algebra, the usual rules of the calculus of relations can be derived (see, e.g., [15, 82]). We recall a few of them. Let P, Q, R be relations, let t, u be left vectors and let j range over an arbitrary index set J. Then,

15	(a)	$P \bigcap_j Q_j \subseteq \bigcap_j PQ_j$	(b)	$(\bigcap_j P_j)Q \subseteq \bigcap_j P_jQ$
	(c)	$P \bigcup_j Q_j = \bigcup_j PQ_j$	(d)	$(\bigcup_j P_j)Q = \bigcup_j P_jQ$
	(e)	$Q \subseteq R \Rightarrow PQ \subseteq PR$	(f)	$P \subseteq Q \Rightarrow PR \subseteq QR$
	(g)	$Q \subseteq R \Leftrightarrow \widehat{Q} \subseteq \widehat{R}$	(h)	$(Q \cup R)\widehat{\ } = \widehat{Q} \cup \widehat{R}$
	(i)	$(Q \cap R)\widehat{\ } = \widehat{Q} \cap \widehat{R}$	(j)	$(QR)\widehat{\ } = \widehat{R}\widehat{Q}$
	(k)	$\widehat{\widehat{R}} = R$	(l)	$\widehat{\overline{R}} = \overline{\widehat{R}}$
	(m)	$LL = L$	(n)	$tt = t$
	(o)	$(t \cap u)L = t \cap u$	(p)	$\overline{t}L = \overline{t}$
	(q)	$(P \cap t)R = PR \cap t$	(r)	$P(Q \cap t) = (P \cap \widehat{t})Q$
	(s)	$P(Q \cap \widehat{t}) = PQ \cap \widehat{t}$	(t)	$QLR = QL \cap LR$
	(u)	$\widehat{P}L \subseteq t \Rightarrow PQ = P(Q \cap t)$	(v)	$\widehat{P}L \subseteq QL \Rightarrow PL = PQL$
	(w)	$\widehat{L} = L \ .$		

16	(a)	$R^+ = R \ \Leftrightarrow \ RR \subseteq R$	(b)	$R^+ = RR^* = R^*R$
	(c)	$R^* = R^*R^*$	(d)	$R^* = (R^*)^*$
	(e)	$Q^*R^* \subseteq (Q \cup R)^*$	(f)	$(Q \cup R)^* = Q^*(RQ^*)^* = (Q^*R)^*Q^*$
	(g)	$(R \cap I)^* = I$	(h)	$(Q \cup R \cap I)^* = Q^* \ .$

17 (a) Q deterministic $\Rightarrow Q(\bigcap_j R_j) = \bigcap_j QR_j$
(b) Q injective $\Rightarrow (\bigcap_j R_j)Q = \bigcap_j R_jQ$
(c) P deterministic $\Rightarrow (Q \cap R\widehat{P})P = QP \cap R$
(d) P injective $\Rightarrow P(\widehat{P}Q \cap R) = Q \cap PR$
(e) Q total $\Leftrightarrow \overline{QR} \subseteq Q\overline{R}$
(f) Q surjective $\Leftrightarrow \overline{RQ} \subseteq \overline{R}Q$
(g) Q deterministic $\Rightarrow Q\overline{R} = QL \cap \overline{QR}$
(h) Q injective $\Rightarrow \overline{R}Q = LQ \cap \overline{RQ}$
(i) Q deterministic $\Rightarrow Q\overline{R} \cup \overline{QL} = \overline{QR}$
(j) Q injective $\Rightarrow \overline{R}Q \cup \overline{LQ} = \overline{RQ}$.

Finally, the Dedekind identity holds (it is a substitute for the Schröder identity).

18 $PQ \cap R \subseteq (P \cap R\widehat{Q})(Q \cap \widehat{P}R)$.

There exists a notion of *division* in relation algebra which exhibits many properties similar to the notion of division in arithmetic. Since the relational product is not commutative, there are two division-like operators.

19 Definition. The *left residue* of relation R by relation Q is denoted by R/Q and is defined by $\overline{\overline{R}\widehat{Q}}$. The *right residue* of relation R by relation Q is denoted[‡] by $Q\backslash R$ and is defined by $\overline{\widehat{Q}\overline{R}}$. □

Operators / and \ have the same precedence, and it is the same precedence as $\circ$. The formal definitions of / and \ in the algebra of concrete relations are more intuitive:

$$R/Q \triangleq \{(s,s')|Q.s' \subseteq R.s\} \ ,$$

and

$$Q\backslash R \triangleq \{(s,s')|\widehat{Q}.s \subseteq \widehat{R}.s'\} \ .$$

The following laws illustrate the similarities between relational division and arithmetic division. The first four laws provide that left residue and right residue are the greatest solutions wrt $\subseteq$ to the inequations $XQ \subseteq R$ and $QX \subseteq R$ respectively. These laws follow easily from the definitions [44, 82].

20 (a) $PQ \subseteq R \Leftrightarrow P \subseteq R/Q$ (b) $PQ \subseteq R \Leftrightarrow Q \subseteq P\backslash R$
(c) $(R/Q)Q \subseteq R$ (d) $Q(Q\backslash R) \subseteq R$
(e) $(P/Q)(Q/R) \subseteq P/R$ (f) $(P\backslash Q)(Q\backslash R) \subseteq P\backslash R$
(g) $(R/Q)\hat{} = Q\hat{}\backslash R\hat{}$ (h) $(R\backslash Q)\hat{} = Q\hat{}/R\hat{}$

We now introduce a property of relations which is useful in proving termination of iterative statements.

21 Definition. A relation R is said to be *progressively finite* [82] if and only if for any left vector t, $t \subseteq Rt \Rightarrow t = \emptyset$ (equivalently, we also say that $\widehat{R}$ is well-founded). □

A concrete relation R is progressively finite if and only if there exists no infinite sequence

$$s_0, s_1, \ldots, s_i, \ldots$$

such that $(s_i, s_{i+1}) \in R$ for all i. To see that the definition above states this fact, assume that we have an infinite sequence by relation R. Let T be the set

$$\{s_1, s_2, \ldots, s_i, \ldots\} \ .$$

Let $t \triangleq T \times L$ and let $t' = (T \cup \{s_0\}) \times L$. Then, we have $Rt \supseteq t' \supseteq t$. On the other hand, if there is a $t \neq \emptyset$ such that $t \subseteq Rt$, then one can construct an infinite sequence from t. Indeed, it follows that any element in the domain T of t is related by R to at least one element of T; hence the sequence (s_i) with s_0 being any element in T, and $s_{i+1} \in R.s_i$, for $i > 0$, is infinite.

Progressively finite relations satisfy the following laws.

22 (a) R progressively finite $\Rightarrow R^*\overline{RL} = L$
(b) R progressively finite $\Rightarrow R \cap Q$ progressively finite .

[‡]Operator \ is already used for substitution, but there is no confusion since a substitution is enclosed between square brackets.

PROOF. For a proof of (a), see [82]. For (b), we have

$$\begin{array}{ll}
 & t \subseteq (R \cap Q)t \\
\Rightarrow & \quad \{ \text{ 15(b) } \} \\
 & t \subseteq Rt \cap Qt \\
\Rightarrow & \quad \{ \text{ 12(g) } \} \\
 & t \subseteq Rt \wedge t \subseteq Qt \\
\Rightarrow & \quad \{ \text{ hypothesis } R \text{ progressively finite } \} \\
 & t = \emptyset \ . \qquad \Box
\end{array}$$

3.3.4 Heterogeneous Relations

So far, we have been dealing with homogeneous relations, which are subsets of a Cartesian product $S \times S$. One can generalize the concept of a relation as a subset of the Cartesian product of two different sets $S \times S'$. These relations are said to be *heterogeneous*. We need heterogeneous relations for defining the semantics of local variables.

Operations in a heterogeneous relation algebra are partially defined, because the underlying sets of the operands must be compatible in order for the operation to make sense. The following definition is taken from [82].

23 Definition. A *relation algebra* is a structure $(\mathcal{R}, \cup, \cap, ^-, ^\frown, \circ)$ over a nonempty set $\mathcal{R}$ of elements, called *relations*. The following conditions are satisfied.

1. Every relation R belongs to a subset $\mathcal{B}_R$ of $\mathcal{R}$ such that $(\mathcal{B}_R, \cup, \cap, ^-, \emptyset, L)$ is a complete atomistic Boolean algebra.
2. For every relation R there exists a *converse* relation $\widehat{R}$ (we will write $(R)^\frown$ rather than $\widehat{(R)}$ for parenthesized expressions). If $\widehat{R} \in \mathcal{B}_R$, then R is said to be *homogeneous*.
3. Given two relations Q, R belonging to suitable Boolean algebras $\mathcal{B}_Q$ and $\mathcal{B}_R$ respectively, an associative *composition* $Q \circ R$ is defined. There exist right and left identities for every set $\mathcal{B}_R$ of relations. The existence of a composition $Q \circ R$ implies that $P \circ R$ is defined for all relations $P \in \mathcal{B}_Q$. Moreover, the compositions $\widehat{R} \circ R$ and $R \circ \widehat{R}$ are always defined.
4. The Schröder rule $P \circ Q \subseteq R \Leftrightarrow \widehat{P} \circ \overline{R} \subseteq \overline{Q} \Leftrightarrow \overline{R} \circ \widehat{Q} \subseteq \overline{P}$ holds whenever one of the three expressions is defined.
5. $L \circ R \circ L = L$ holds for every $R \neq \emptyset$ (Tarski rule). $\Box$

For simplicity, the universal, zero, and identity elements are all denoted by $L, \emptyset, I$ (respectively), except that we may occasionally use subscripts to designate a base set. Heterogeneous relations obey laws very similar to homogeneous relations. Laws of 15 hold in a heterogeneous relation algebra if the operations are defined and if the universal, zero and identity elements are adjusted with the appropriate subscript.

3.4 The Refinement Ordering

3.4.1 Ordering Properties

Given a set S, we define an ordering between relations on S, under the name *refinement ordering*. This ordering, which we prescribe in the definition below, compares the amount of input/output information that a relational specification carries.

24 Definition. Relation R is said to be *refined* by relation R', denoted by $R \sqsubseteq R'$, if and only if

$$RL \subseteq R'L \wedge RL \cap R' \subseteq R \ . \qquad \Box$$

The set-theoretic interpretation of this definition is the following:

$$dom(R) \subseteq dom(R') \wedge \forall s : s \in dom(R) \Rightarrow R'.s \subseteq R.s \ .$$

As an illustration of this definition, we give below a pair of relations R and R' such that R is refined by R', leaving it to the reader to ponder in what sense R' carries more input/output information than R'.

$$R \triangleq \{(0,1),(0,2),(0,3),(1,2),(1,3),(1,4)\} \ ,$$

$$R' \triangleq \{(0,1),(0,2),(1,2),(1,3),(2,3),(2,4)\} \ .$$

We leave it to the interested reader to check that the relation $\sqsubseteq$ defined above (among relations) is reflexive, transitive and antisymmetric; hence it is a partial ordering.

The definition and proposition below provide useful information on our refinement ordering; in particular, they highlight its importance for our purposes. We consider Pascal-like variable declarations and we let S be the set that they define, and p be a Pascal program on S. The *functional abstraction* of program p is the function denoted by $[p]$ and defined as the set of pairs (s, s') of S such that if p starts execution in state s then it terminates in state s'.

25 Definition. Program p is said to be *correct* with respect to specification R if and only if $[p]$ refines R. $\Box$

Despite its difference of form, this definition is equivalent to traditional definitions of *total correctness* [26, 33, 53]. The following proposition sheds more light on the refinement ordering, and its importance for our purposes.

26 Proposition. *Given relations R and R' such that R refines R', any program correct with respect to R is correct with respect to R'.* $\Box$

This proposition stems immediately from the transitivity of the refinement ordering. This proposition provides, e.g., that a programmer who is given specification R' may transform it into specification R, assured in the knowledge that any solution to R is solution to R'; we say that R is a *refinement* of R', or that R *refines* R' —hence the name given to this ordering.

3.4.2 Lattice Properties

Given that the refinement relation has the properties of an ordering, a legitimate question is to ask whether it is a lattice. Investigations by Mili et al [23, 13, 60] provide a number of results in this regard, which we content ourselves in this paper with presenting (without proof) and illustrating.

27 Proposition. *Any two relations R and R' have a greatest lower bound, defined by the following formula:*

$$R \sqcap R' \triangleq RL \cap R' \cup R'L \cap R \ . \qquad \Box$$

Operation $\sqcap$ is called *demonic meet*. Close inspection of this formula leads us to the following interpretation [13]: the demonic meet of two relations carries the information that is common to both components. The more R and R' have information in common, the greater (with respect to the refinement ordering) their demonic meet. Incidentally, this formula is known to other authors as the *binary choice* [83], the *demonic choice* [9] or the *demonic join* [11]. As an example, consider the following relations.

$$R_1 \triangleq \{(0,0)\}$$

$$R_2 \triangleq \{(1,1)\}$$

$$R_3 \triangleq \{(0,1),(1,0)\}$$

Computing the meets of these relations, we find the following.

$$R_1 \sqcap R_2 = \emptyset$$

$$R_1 \sqcap R_3 = \{(0,0),(0,1)\}$$

$$R_2 \sqcap R_3 = \{(1,0),(1,1)\}$$

28 Proposition. *Any two relations R and R' that satisfy the following condition,*

$$RL \cap R'L = (R \cap R')L \ ,$$

have a least upper bound, which is given by the formula

$$R \sqcup R' \triangleq \overline{R'L} \cap R \cup \overline{RL} \cap R' \cup R \cap R' \ . \qquad \Box$$

Operation $\sqcup$ is called *demonic join*. Close inspection of the condition of existence of least upper bound leads to the following interpretation [13]: two relations R and R' satisfy this condition if and only if they do not contradict each other, in the following sense: given s in the domain of R and R', the image sets of s by R and R' are not disjoint (they may be different sets, but must have elements in common). In light of this interpretation, we refer to the condition of existence of a least upper bound as the *consistency condition*, and we abbreviate it by: $cs(R,R')$. Note that because

$$(R \cap R')L \subseteq RL \cap R'L$$

is a tautology, the consistency condition is equivalent to

$$RL \cap R'L \subseteq (R \cap R')L \ .$$

In programming terms, the consistency condition between two relations R and R' is the condition under which R and R' admit a common refinement, i.e., they can be refined by a common program. As an example, consider the following relations.

$$R_1 \triangleq \{(0,0)\}$$

$$R_2 \triangleq \{(1,1)\}$$

$$R_3 \triangleq \{(0,1),(1,0)\}$$

$$R_4 \triangleq \{(0,0),(0,1),(1,0),(1,1)\}$$

Relations R_1 and R_2 are consistent: their domains are disjoint; hence they have no common input for which their outputs do not agree. Relations R_1 and R_3 are inconsistent: for input 0, their outputs cannot agree on at least one value. Similarly, R_2 and R_3 are inconsistent. Relation R_4 is consistent with relation R_1, because for input 0 they agree on output 0. Similarly, relation R_4 is consistent with relations R_2 and R_3. Computing the joins of consistent relations, we find the following.

$$R_1 \sqcup R_2 = \{(0,0),(1,1)\}$$

$$R_1 \sqcup R_4 = \{(0,0),(1,0),(1,1)\}$$

$$R_2 \sqcup R_4 = \{(0,0),(0,1),(1,1)\}$$

$$R_3 \sqcup R_4 = \{(0,1),(1,0)\} = R_3$$

It stems from the definition of the refinement ordering (24) that the empty relation is the least element of our ordering, i.e., any relation refines the empty relation. On the other hand, we could not identify a relation that refines all other relations.

Given two relations R and R' that satisfy the consistency condition, their demonic join can be interpreted as follows [13]: the demonic join of R and R' carries all the information in R and all the information in R'; it can be thought of as the *sum* (in terms of input output information) of R and R'. The demonic join and demonic meet take simple forms under two distinct conditions, which we explore below.

29 Corollary. *If R and R' satisfy the following condition,*

$$R'L \cap R = RL \cap R' \ ,$$

then R and R' satisfy the consistency condition; furthermore,

$$R \sqcup R' = R \cup R'$$

and

$$R \sqcap R' = R \cap R' \ . \qquad \square$$

The proof of this corollary is quite straightforward, as it stems from the definitions; note that the condition of this corollary is trivially met whenever $RL \cap R'L = \emptyset$ (R and R' have disjoint domains) or when R and R' are left vectors.

30 Corollary. *If R and R' satisfy the following condition,*

$$RL = R'L = (R \cap R')L \ ,$$

then R and R' satisfy the consistency condition; furthermore,

$$R \sqcup R' = R \cap R'$$

and

$$R \sqcap R' = R \cup R' \ . \qquad \square$$

The proof of this corollary is quite straightforward, as it stems from the definitions; note that the condition of this corollary is trivially met whenever R and R' are two non-disjoint right vectors. We cite one last result, which was proved in [23], and which will be useful for us in providing the existence of infinite demonic meets.

31 Proposition. *The set of relations ordered by $\sqsubseteq$ forms a complete $\sqcap$-semilattice.* $\square$

A natural way to illustrate a lattice structure is to show a cube-like graph that highlights, for each pair of elements, their least upper bound and their greatest lower bound. This is what we do in the figure below. We let space S be defined by $S \triangleq \{a, b, c, d\}$ and let relations R_0, R_1, R_2, R_3, R_4, R_5, R_6, R_7 be defined as follows.

$$\begin{aligned}
&R_0 \triangleq \{(d,a),(d,b),(d,c),(d,d)\}\\
&R_1 \triangleq \{(a,a),(d,a),(d,c),(d,d)\}\\
&R_2 \triangleq \{(b,b),(d,b),(d,c),(d,d)\}\\
&R_3 \triangleq \{(c,c),(d,a),(d,b),(d,d)\}\\
&R_4 \triangleq \{(a,a),(b,b),(d,c),(d,d)\}\\
&R_5 \triangleq \{(a,a),(c,c),(d,a),(d,d)\}\\
&R_6 \triangleq \{(b,b),(c,c),(d,b),(d,d)\}\\
&R_7 \triangleq \{(a,a),(b,b),(c,c),(d,d)\}
\end{aligned}$$

We leave it to the reader to ponder in what sense this figure reflects the amount of information carried by specifications (higher specifications carry more information) as well as the lattice operations between specifications. Hence, e.g., we prove below that specification R_6 refines specification R_2.

$$\begin{aligned}
&R_6L = \{(s,s')|s = b \vee s = c \vee s = d\} \ ,\\
&R_2L = \{(s,s')|s = b \vee s = d\} \ .
\end{aligned}$$

Hence

$$R_2L \subseteq R_6L \ .$$

On the other hand, we consider $R_2L \cap R_6$ and show it to be included in R_2.

$$\begin{aligned}
&\quad R_2L \cap R_6\\
&=\\
&\quad \{(b,b),(d,b),(d,d)\}\\
&\subseteq\\
&\quad \{(b,b),(d,b),(d,c),(d,d)\}\\
&=\\
&\quad R_2 \ .
\end{aligned}$$

Hence R_6 refines R_2. In addition, we want to show that the least upper bound of R_1 and R_3 is indeed R_5, as shown in figure 3.1.

$$\begin{aligned}
&R_1L = \{(s,s')|s = a \vee s = d\} \ ,\\
&R_3L = \{(s,s')|s = c \vee s = d\} \ .
\end{aligned}$$

Then

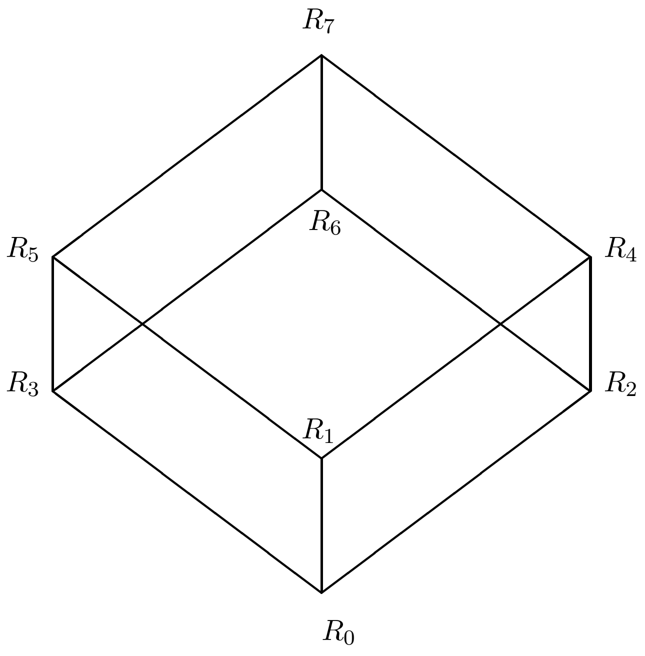


Figure 3.1: A Sample Lattice of Specifications

$$
\begin{array}{ll}
 & R_1 \sqcup R_3 \\
= & \\
 & \overline{R_3 L} \cap R_1 \cup \overline{R_1 L} \cap R_3 \cup (R_1 \cap R_3) \\
= & \\
 & \{(a,a)\} \cup \{(c,c) \cup \{(d,a),(d,d)\} \\
= & \\
 & R_5 \enspace .
\end{array}
$$

We leave it to the reader to ponder in what sense specification R_5 can be said to carry the sum of input output information of R_1 and R_3. We now show that R_1 is the greatest lower bound of R_4 and R_5

$$
\begin{array}{ll}
 & R_4 \sqcap R_5 \\
= & \\
 & R_4 L \cap R_5 \cup R_4 \cap R_5 L \\
= & \\
 & \{(a,a),(d,a),(d,d)\} \cup \{(a,a),(d,c),(d,d)\} \\
= & \\
 & \{(a,a),(d,a),(d,c),(d,d)\} \\
= & \\
 & R_1 \enspace .
\end{array}
$$

When the necessary joins are defined, the following identities hold, where $\mathcal{A}$ is a non-empty set of relations. The proof of (a) and (b) is found in [23].

32	(a)	$P \sqcup \bigsqcap_{Q \in \mathcal{A}} Q = \bigsqcap_{Q \in \mathcal{A}} P \sqcup Q$	(b)	$P \sqcap \bigsqcup_{Q \in \mathcal{A}} Q = \bigsqcup_{Q \in \mathcal{A}} P \sqcap Q$
	(c)	$(P \sqcup Q)L = PL \cup QL$	(d)	$(P \sqcap Q)L = PL \cap QL$
	(e)	$t \cap (Q \sqcap R) = (t \cap Q) \sqcap (t \cap R)$	(f)	$t \cap (Q \sqcup R) = (t \cap Q) \sqcup (t \cap R)$
	(g)	$(t \sqcap u)^{\frown} = \widehat{t} \sqcup \widehat{u}$	(h)	$(t \sqcup u)^{\frown} = \widehat{t} \sqcap \widehat{u}$
	(i)	$P \cap Q \sqsubseteq P \sqcup Q$	(j)	$P \cup Q \sqsubseteq P \sqcup Q$
	(k)	$P \sqcap Q \sqsubseteq P \cap Q$	(l)	$P \sqcap Q \sqsubseteq P \cup Q$
	(m)	$t \sqcap u = t \cap u$	(n)	$t \sqcup u = t \cup u$
	(o)	$\widehat{t} \sqcap \widehat{u} = \widehat{t} \cup \widehat{u}$	(p)	$\widehat{t} \sqcup \widehat{u} = \widehat{t} \cap \widehat{u}$
	(q)	$R = t \cap R \sqcup \overline{t} \cap R$		.

3.4.3 Demonic Operators

An embedding of this semilattice in a relation algebra has been defined [23] in such a way that the lattice operators are preserved. One finds that the semilattice operation that corresponds to composition in the embedding algebra is the so-called *demonic composition* [9, 11, 23].

33 Definition. The *demonic composition* of relations Q and R is denoted by $Q \mathbin{\Box} R$ and given by $QR \cap \overline{Q\overline{RL}}$. $\Box$

The equivalent set-theoretic definition of demonic composition is more intuitive:

$$P \mathbin{\Box} Q = \{(s,s') | (s,s') \in PQ \land P.s \subseteq dom(Q)\} \ .$$

It is noteworthy that when P is deterministic, the demonic product of P by Q is the same as the traditional relational product.

As an example of demonic composition, we have the following equalities between relations defined over the set **Real**:

$$\begin{aligned} \{(s,s')|s'^2 = s\} \mathbin{\Box} \{(s,s')|s'^2 = s\} &= \{(s,s')|s = s' = 0\} \ , \\ \{(s,s')|s'^2 = s \land s' \geq 0\} \mathbin{\Box} \{(s,s')|s'^2 = s\} &= \{(s,s')|s'^4 = s\} \ . \end{aligned}$$

We assign to $\Box$ the same precedence as $\circ$. The following laws hold when the necessary joins exist. The proof of (a) to (h) are given in [23]. The others are proved using (a) to (h) and other aforementioned relational identities. In the following, we use symbol $\mathcal{A}$ for an arbitrary non-empty set of relations.

34	(a)	$P \mathbin{\Box} (Q \mathbin{\Box} R) = (P \mathbin{\Box} Q) \mathbin{\Box} R$	(b)	$P \mathbin{\Box} I = I \mathbin{\Box} P = P$
	(c)	$Q \sqsubseteq R \Rightarrow P \mathbin{\Box} Q \sqsubseteq P \mathbin{\Box} R$	(d)	$P \sqsubseteq Q \Rightarrow P \mathbin{\Box} R \sqsubseteq Q \mathbin{\Box} R$
	(e)	$P \mathbin{\Box} \bigsqcap_{Q \in \mathcal{A}} Q = \bigsqcap_{Q \in \mathcal{A}} P \mathbin{\Box} Q$	(f)	$(\bigsqcap_{P \in \mathcal{A}} P) \mathbin{\Box} Q = \bigsqcap_{P \in \mathcal{A}} P \mathbin{\Box} Q$
	(g)	$P \mathbin{\Box} \bigsqcup_{Q \in \mathcal{A}} Q \sqsupseteq \bigsqcup_{Q \in \mathcal{A}} P \mathbin{\Box} Q$	(h)	$(\bigsqcup_{P \in \mathcal{A}} P) \mathbin{\Box} Q \sqsupseteq \bigsqcup_{P \in \mathcal{A}} P \mathbin{\Box} Q$
	(i)	$\widehat{P}P \subseteq I \Rightarrow P \mathbin{\Box} Q = PQ$	(j)	$\widehat{P}L \subseteq QL \Rightarrow P \mathbin{\Box} Q = PQ$
	(k)	$Q \subseteq R \Rightarrow P \mathbin{\Box} Q \subseteq P \mathbin{\Box} R$	(l)	$t \cap (P \mathbin{\Box} Q) = (t \cap P) \mathbin{\Box} Q$
	(m)	$P \mathbin{\Box} (t \cap Q) = P \mathbin{\Box} t \cap P \mathbin{\Box} Q$	(n)	$\widehat{t} = L \mathbin{\Box} \widehat{t}$
	(o)	$(t \cap P \sqcup \overline{t} \cap Q) \mathbin{\Box} R =$ $t \cap P \mathbin{\Box} R \sqcup \overline{t} \cap Q \mathbin{\Box} R$.		

Henceforth, we may refer to some of the native operations of a relation algebra $(\cup, \cap, \circ)$ as *angelic* operations. Using the embedding, one naturally derives other demonic operations and demonic properties of relations corresponding to the angelic operations and angelic properties of relations.

35 Definition. A relation R is said to be *demonically reflexive* if and only if $R \sqsubseteq I$; a relation R is said to be *demonically transitive* if and only if $R \sqsubseteq R \Box R$; the i^{th} demonic relative power of relation R is denoted by $R^{\boxed{i}}$ and is given by

$$R^{\boxed{0}} \triangleq I \quad \text{and} \quad R^{\boxed{i+1}} \triangleq R \Box R^{\boxed{i}} \ ;$$

the *demonic transitive closure* of relation R is denoted by $R^{\boxed{+}}$ and given by $\bigsqcap_{i>0} R^{\boxed{i}}$; the *demonic reflexive transitive closure* (or simply *demonic closure* for short) of relation R is denoted by $R^{\boxed{*}}$ and is given by $R^{\boxed{+}} \sqcap I$ or, equivalently, $\bigsqcap_{i\geq 0} R^{\boxed{i}}$. □

Because it is based on demonic product, demonic closure may differ significantly from the angelic closure (*), as the following example illustrates it:

$$\{(s,s')|s'^2 = s\}^* = \{(s,s')|\exists n \geq 0 : s'^{2^n} = s\} \ ,$$

$$\{(s,s')|s'^2 = s\}^{\boxed{*}} = \{(s,s')|s = s' = 0\} \ .$$

We assign to $\boxed{*}$ the same precedence as $\widehat{\ }$ and $\overline{\ }$.

The next proposition provides an alternative definition of *demonic closure*, based on the notion of greatest fixpoint.

36 Proposition. *Let R be a relation; let f be the function given by $f.X \triangleq I \sqcap R \sqcap X \Box X$. The greatest fixpoint of f wrt $\sqsubseteq$ exists, and it is given by $R^{\boxed{*}}$.*

Proof. Let $f^0.X \triangleq X$, and $f^{n+1}.X \triangleq f.f^n.X$. By induction, we can show that $f^n.I \sqsubseteq I$ for all n. It is easy to show that f is $\sqcap$-continuous, using laws 34(e),(f). Therefore, we have

$$\begin{array}{ll} & f.\bigsqcap_{n\geq 0} f^n.I \\ = & \qquad \{ f \text{ is } \sqcap\text{-continuous} \} \\ & \bigsqcap_{n\geq 1} f^n.I \\ = & \qquad \{ f^n.I \sqsubseteq I \text{ for all } n \} \\ & \bigsqcap_{n\geq 0} f^n.I \ . \end{array}$$

Hence, $\bigsqcap_{n\geq 0} f^n.I$ is a fixpoint of f. Let Z be a fixpoint of f. We have $Z = I \sqcap R \sqcap Z \Box Z \sqsubseteq I$. Since f is monotonic, it is easy to show by induction that $Z \sqsubseteq f^n.I$ for all n. Hence, $Z \sqsubseteq \bigsqcap_{n\geq 0} f^n.I$, and $\bigsqcap_{n\geq 0} f^n.I$ is the greatest fixpoint of f. Now, a simple induction shows that $\bigsqcap_{n\geq 0} f^n.I = R^{\boxed{*}}$. □

The demonic closure satisfies properties similar to its angelic counterpart (R^*). We only mention four of them.

37 (a) $P \sqsubseteq Q \Rightarrow P^{\boxed{*}} \sqsubseteq Q^{\boxed{*}}$ (b) $P \sqsubseteq I \wedge P \sqsubseteq P \Box P \Leftrightarrow P = P^{\boxed{*}}$
(c) $P^{\boxed{*}} = P^{\boxed{*}} \Box P^{\boxed{*}}$ (d) $P^{\boxed{*}} = P^{\boxed{*}\,\boxed{*}}$
(e) $P^{\boxed{*}} \sqsubseteq (t \cap P \sqcup \bar{t} \cap I)^{\boxed{*}}$.

Proof. Identity (a) follows easily from the definition of $\boxed{*}$, since meet and demonic composition are monotonic. For (b), we have

$$\begin{array}{ll} & P \sqsubseteq I \wedge P \sqsubseteq P \Box P \\ \Leftrightarrow & \qquad \{ \text{reflexivity of } \sqsubseteq \} \\ & P \sqsubseteq I \wedge P \sqsubseteq P \Box P \wedge P \sqsubseteq P \\ \Leftrightarrow & \qquad \{ \text{property of meet} \} \\ & P \sqsubseteq I \sqcap P \sqcap P \Box P \\ \Leftrightarrow & \qquad \{ I \sqcap P \sqcap P \Box P \sqsubseteq P, \text{ by def. of meet} \} \\ & P = I \sqcap P \sqcap P \Box P \ . \end{array}$$

Therefore, P is a fixpoint of $f.X \triangleq I \sqcap P \sqcap X \Box X$. It is also the greatest fixpoint: let Y be a fixpoint of f; then $Y = I \sqcap P \sqcap Y \Box Y \sqsubseteq P$. By Proposition 36, we have $P = P^{\boxed{*}}$. For the reverse implication, we have:

$$
\begin{array}{ll}
 & P = P^{\boxed{*}} \\
\Rightarrow & \qquad \{ \text{ Proposition 36 } \} \\
 & P = I \sqcap P \sqcap P \Box P \\
\Leftrightarrow & \qquad \{ \text{ property of meet } \} \\
 & P \sqsubseteq I \wedge P \sqsubseteq P \wedge P \sqsubseteq P \Box P \\
\Rightarrow & \qquad \{ \text{ property of } \wedge \text{ } \} \\
 & P \sqsubseteq I \wedge P \sqsubseteq P \Box P \ \ .
\end{array}
$$

For (c), we have

$$
\begin{array}{ll}
 & P^{\boxed{*}} \\
= & \qquad \{ \text{ Proposition 36 } \} \\
 & I \sqcap P \sqcap P^{\boxed{*}} \Box P^{\boxed{*}} \\
\sqsubseteq & \qquad \{ \text{ property of meet } \} \\
 & P^{\boxed{*}} \Box P^{\boxed{*}} \\
\sqsubseteq & \qquad \{ P^{\boxed{*}} \sqsubseteq I, \text{ monotonicity of } \Box \text{ } \} \\
 & P^{\boxed{*}} \Box I \\
= & \qquad \{ \text{ 32(b) } \} \\
 & P^{\boxed{*}} \ \ .
\end{array}
$$

For (d), since $P^{\boxed{*}} = I \sqcap P \sqcap P^{\boxed{*}} \Box P^{\boxed{*}}$, we also have $P^{\boxed{*}} \sqsubseteq I$ and $P^{\boxed{*}} \sqsubseteq P^{\boxed{*}} \Box P^{\boxed{*}}$. From 37(b), it follows that $P^{\boxed{*}} = P^{\boxed{*}\,\boxed{*}}$.

For (e), we have

$$
\begin{array}{ll}
 & P^{\boxed{*}} \\
= & \qquad \{ \text{ Proposition 36 } \} \\
 & I \sqcap P \sqcap P^{\boxed{*}} \Box P^{\boxed{*}} \\
\sqsubseteq & \qquad \{ \text{ property of meet } \} \\
 & I \sqcap P \\
= & \qquad \{ \text{ law 32(i) } \} \\
 & t \cap (I \sqcap P) \sqcup \bar{t} \cap (I \sqcap P) \\
\sqsubseteq & \qquad \{ \text{ property of meet, monotonicity } \} \\
 & t \cap P \sqcup \bar{t} \cap I \ \ .
\end{array}
$$

Hence, by 37(a,d), we have

$$P^{\boxed{*}} = P^{\boxed{*}\,\boxed{*}} \sqsubseteq (t \cap P \sqcup \bar{t} \cap I)^{\boxed{*}} \ \ . \qquad \Box$$

The next operator is the basis for the definition of *demonic* division.

38 Definition.[22] The *conjugate kernel* of relations R and Q is denoted by $\kappa(R, Q)$ and defined by

$$\kappa(R, Q) \triangleq R/Q \cap L\widehat{Q} \ \ . \qquad \Box$$

The definition of conjugate kernel in a concrete relation algebra is more insightful:

$$\kappa(R, Q) \triangleq \{(s, s')|Q.s' \neq \emptyset \wedge Q.s' \subseteq R.s\} \ \ .$$

The following laws hold for the conjugate kernel [22].

39 (a) $\kappa(R,Q) = R/Q \cap R\widehat{Q}$
(b) $\kappa(R,Q) = R\widehat{Q} \Leftrightarrow R\widehat{Q}Q \subseteq R$
(c) $\kappa(R,Q) = R\widehat{Q} \wedge LR \subseteq LQ \Leftrightarrow R\widehat{Q}Q = R$.

As there exists a concept of division for the relational product, there also exists a concept of division for the demonic product.

40 Definition. A relation Q is said to be a *demonic right factor* of relation R if and only if the inequation $R \sqsubseteq X \,\Box\, Q$ has a solution in X. A relation Q is said to be a *demonic left factor* of relation R if and only if the inequation $R \sqsubseteq Q \,\Box\, X$ has a solution in X.

41 Definition.[23] The *demonic left residue* of relation R by relation Q is a partial operation denoted by $R /\!\!/ Q$ and defined by $\kappa(R,Q)$ whenever the following condition is satisfied:

$$RL \subseteq \kappa(R,Q)L \ .$$

The *demonic right residue* of relation R by relation Q is a partial operation denoted by $Q \backslash\!\!\backslash R$ and defined by $\kappa(\widehat{R},(RL \cap Q)^{\wedge})^{\wedge}$ whenever the following condition is satisfied:

$$RL \subseteq QL \wedge L \subseteq ((RL \cap Q)\backslash R)L \ .$$

We let $\mathbf{def}(R /\!\!/ Q)$ and $\mathbf{def}(Q \backslash\!\!\backslash R)$ represent the conditions of definition of $R /\!\!/ Q$ and $Q \backslash\!\!\backslash R$ respectively. $\Box$

We assign to $/\!\!/$ and $\backslash\!\!\backslash$ the same precedence as $/$, $\backslash$, $\circ$ and $\Box$. Operations $/\!\!/$ and $\backslash\!\!\backslash$ correspond to operations $/$ and $\backslash$ in the embedding of the demonic semilattice [23]. As such, we would expect them to be the least solutions to linear inequations. First, we investigate the existence of least solutions.

42 Proposition. *If the inequation $R \sqsubseteq X \,\Box\, Q$ has a solution in X, then there exists a least solution wrt $\sqsubseteq$.*

Proof. Let $\mathcal{A}$ be the non-empty set of solutions.

$$\begin{array}{ll} & \forall X : X \in \mathcal{A} \Rightarrow R \sqsubseteq X \,\Box\, Q \\ \Leftrightarrow & \qquad \{ \text{ definition of meet } \} \\ & R \sqsubseteq \bigsqcap_{X \in \mathcal{A}} X \,\Box\, Q \\ \Leftrightarrow & \qquad \{ \text{ distributivity of } \Box \text{ over } \sqcap \} \\ & R \sqsubseteq (\bigsqcap_{X \in \mathcal{A}} X) \,\Box\, Q \ . \qquad \Box \end{array}$$

43 Proposition. *If the inequation $R \sqsubseteq Q \,\Box\, X$ has a solution in X, then there exists a least solution wrt $\sqsubseteq$.*

Proof. Let $\mathcal{A}$ be the non-empty set of solutions.

$$\begin{array}{ll} & \forall X : X \in \mathcal{A} \Rightarrow R \sqsubseteq Q \,\Box\, X \\ \Leftrightarrow & \qquad \{ \text{ definition of meet } \} \\ & R \sqsubseteq \bigsqcap_{X \in \mathcal{A}} Q \,\Box\, X \\ \Leftrightarrow & \qquad \{ \text{ distributivity of } \Box \text{ over } \sqcap \} \\ & R \sqsubseteq Q \,\Box\, (\bigsqcap_{X \in \mathcal{A}} X) \ . \qquad \Box \end{array}$$

The following laws provide that $/\!\!/$ and $\backslash\!\!\backslash$ are the least solutions to linear inequations. Inequations (c) and (d) below hold provided that $/\!\!/$ and $\backslash\!\!\backslash$ are defined. Proofs can be done using results from [23].

44 (a) $R \sqsubseteq P \mathbin{\square} Q \Leftrightarrow \mathbf{def}(R/\!\!/Q) \wedge R/\!\!/Q \sqsubseteq P$
(b) $R \sqsubseteq P \mathbin{\square} Q \Leftrightarrow \mathbf{def}(P\backslash\!\!\backslash R) \wedge P\backslash\!\!\backslash R \sqsubseteq Q$
(c) $R \sqsubseteq (R/\!\!/Q) \mathbin{\square} Q$
(d) $R \sqsubseteq Q \mathbin{\square} (Q\backslash\!\!\backslash R)$.

Demonic residues are monotonic and antitonic wrt $\sqsubseteq$, like division in arithmetic.

45 Proposition. *If $P \sqsubseteq Q$ then the following laws hold, provided that $/\!\!/$ and $\backslash\!\!\backslash$ are defined:*

(a) $R/\!\!/Q \sqsubseteq R/\!\!/P$
(b) $Q\backslash\!\!\backslash R \sqsubseteq P\backslash\!\!\backslash R$
(c) $P/\!\!/R \sqsubseteq Q/\!\!/R$
(d) $R\backslash\!\!\backslash P \sqsubseteq R\backslash\!\!\backslash Q$.

Proof. For (a), we have

$$R \sqsubseteq (R/\!\!/P) \mathbin{\square} P \Rightarrow R \sqsubseteq (R/\!\!/P) \mathbin{\square} Q \Leftrightarrow R/\!\!/Q \sqsubseteq R/\!\!/P \ .$$

Similarly for (b). For (c), we have

$$P \sqsubseteq Q \Rightarrow P \sqsubseteq (Q/\!\!/R) \mathbin{\square} R \Leftrightarrow P/\!\!/R \sqsubseteq Q/\!\!/R \ .$$

Similarly for (d). $\square$

Demonic residues satisfy the following properties, provided they exist.

46

(a)	$(R/\!\!/Q)\backslash\!\!\backslash R \sqsubseteq Q$	(b)	$R/\!\!/(Q\backslash\!\!\backslash R) \sqsubseteq Q$
(c)	$R/\!\!/((R/\!\!/Q)\backslash\!\!\backslash R) = R/\!\!/Q$	(d)	$(R/\!\!/(Q\backslash\!\!\backslash R))\backslash\!\!\backslash R = Q\backslash\!\!\backslash R$
(e)	$R/\!\!/R \sqsubseteq I$	(f)	$R\backslash\!\!\backslash R \sqsubseteq I$
(g)	$(R/\!\!/R) \mathbin{\square} R = R$	(h)	$R \mathbin{\square} (R\backslash\!\!\backslash R) = R$
(i)	$R/\!\!/R \sqsubseteq (R/\!\!/R) \mathbin{\square} (R/\!\!/R)$	(j)	$R\backslash\!\!\backslash R \sqsubseteq (R\backslash\!\!\backslash R) \mathbin{\square} (R\backslash\!\!\backslash R)$
(k)	$R/\!\!/R = (R/\!\!/R)^{\boxast}$	(l)	$R\backslash\!\!\backslash R = (R\backslash\!\!\backslash R)^{\boxast}$
(m)	$R/\!\!/I = R$	(n)	$I\backslash\!\!\backslash R = R$
(o)	$(R/\!\!/R)\backslash\!\!\backslash R = R$	(p)	$R/\!\!/(R\backslash\!\!\backslash R) = R$.

Proof. For (a), we have

$$\begin{array}{ll} & (R/\!\!/Q)\backslash\!\!\backslash R \sqsubseteq Q \\ \Leftrightarrow & \qquad \{\ 44(a)\ \} \\ & R \sqsubseteq R/\!\!/Q \mathbin{\square} Q \\ \Leftrightarrow & \qquad \{\ 44(c)\ \} \\ & \mathbf{true}\ . \end{array}$$

Similarly for (b). For (c), by law 46(b), we have $R/\!\!/((R/\!\!/Q)\backslash\!\!\backslash R) \sqsubseteq R/\!\!/Q$. In addition, by laws 46(a) and 45(a), we have $R/\!\!/Q \sqsubseteq R/\!\!/((R/\!\!/Q)\backslash\!\!\backslash R)$. Similarly for (d). For (e), we have

$$R \sqsubseteq I \mathbin{\square} R \Leftrightarrow R/\!\!/R \sqsubseteq I \ .$$

Similarly for (f). For (g), using 44(c) and 46(e), we have

$$R \sqsubseteq R /\!\!/ R \mathbin{\Box} R \sqsubseteq I \mathbin{\Box} R = R \ .$$

Similarly for (h). For (i), we have

$$\begin{array}{ll}
 & R /\!\!/ R \sqsubseteq (R /\!\!/ R) \mathbin{\Box} (R /\!\!/ R) \\
\Leftrightarrow & \qquad \{ \ 44(a) \ \} \\
 & R \sqsubseteq (R /\!\!/ R) \mathbin{\Box} (R /\!\!/ R) \mathbin{\Box} R \\
\Leftarrow & \qquad \{ \ 46(g) \ \} \\
 & R \sqsubseteq R /\!\!/ R \mathbin{\Box} R \\
\Leftrightarrow & \qquad \{ \ 46(g) \ \} \\
 & \mathbf{true} \ .
\end{array}$$

Similarly for (j). Law (k) follows from (e), (i) and law 37(b). Similarly for (l). For (m), we have

$$R \sqsubseteq R \mathbin{\Box} I \Leftrightarrow R /\!\!/ I \sqsubseteq R \wedge \mathbf{def}(R /\!\!/ I)$$

by law 44(a). Also, we have

$$R \sqsubseteq (R /\!\!/ I) \mathbin{\Box} I = R /\!\!/ I$$

by law 44(c). Hence, $R = R /\!\!/ I$. Similarly for (n). For (o), we have

$$\begin{array}{ll}
 & R \sqsubseteq (R /\!\!/ R) \backslash\!\!\backslash R \\
\Leftrightarrow & \qquad \{ \ 46(n) \ \} \\
 & I \backslash\!\!\backslash R \sqsubseteq (R /\!\!/ R) \backslash\!\!\backslash R \\
\Leftarrow & \qquad \{ \ 45(b) \ \} \\
 & R /\!\!/ R \sqsubseteq I \\
\Leftrightarrow & \qquad \{ \ 46(e) \ \} \\
 & \mathbf{true} \ .
\end{array}$$

We also have $(R /\!\!/ R) \backslash\!\!\backslash R \sqsubseteq R$ by (a). The proof is similar for (p). □

The computation of demonic residues is rather unwieldy, due to the complex definition of the conjugate kernel. However, for the following class of relations, it is very easy to compute.

47 Definition. A relation Q is said to be *regular relative* to R if and only if

$$R \sqsubseteq R\widehat{Q}Q \ . \qquad \Box$$

The following law provides an easier formula for computing kernels, provided the condition of relative regularity is met.

48 $\quad R \sqsubseteq R\widehat{Q}Q \Leftrightarrow \mathbf{def}(R /\!\!/ Q) \wedge R /\!\!/ Q = R\widehat{Q}$

Proof.

$$\begin{array}{ll}
 & R \sqsubseteq R\widehat{Q}Q \\
\Leftrightarrow & \qquad \{ \text{ definition of } \sqsubseteq \ \} \\
 & RL \cap R\widehat{Q}Q \subseteq R \wedge RL \subseteq R\widehat{Q}QL \\
\Leftrightarrow & \qquad \{ \text{ Boolean laws } \} \\
 & R\widehat{Q}Q \subseteq R \wedge RL \subseteq R\widehat{Q}QL \\
\Leftrightarrow & \qquad \{ \ 39(b) \ \} \\
 & \kappa(R,Q) = R\widehat{Q} \wedge RL \subseteq \kappa(R,Q)L \\
\Leftrightarrow & \qquad \{ \text{ definition of } /\!\!/ \ \} \\
 & \mathbf{def}(R /\!\!/ Q) \wedge R /\!\!/ Q = R\widehat{Q} \ . \qquad \Box
\end{array}$$

The notion of relative regularity is an extension of the notion of *regularity.*

49 Definition.[46] A relation R is said to be *regular* (or *difunctional* [82], from the French *difonctionnelle* [80]) if and only if

$$R\widehat{R}R \subseteq R \ . \qquad \Box$$

When R is regular relative to itself, we have

$$\begin{array}{ll} & R \sqsubseteq R\widehat{R}R \\ \Leftrightarrow & \qquad \{ \text{ definition of } \sqsubseteq \ \} \\ & R\widehat{R}R \cap RL \subseteq R \wedge RL \subseteq R\widehat{R}RL \\ \Leftrightarrow & \qquad \{ \ R\widehat{R}R \subseteq RL, \text{ by } \widehat{R}R \subseteq L \text{ and monotonicty } \} \\ & R\widehat{R}R \subseteq R \wedge RL \subseteq R\widehat{R}RL \\ \Leftrightarrow & \qquad \{ \ RL \subseteq R\widehat{R}RL, \text{ by Dedekind (18) and monotonicity } \} \\ & R\widehat{R}R \subseteq R \quad . \end{array}$$

A necessary and sufficient condition for a concrete relation R to be regular is [46, 80]:

$$\forall s, s' : R.s \cap R.s' \neq \emptyset \Rightarrow R.s = R.s' \ .$$

In other words, when two elements share an image in a regular relation, they have the same image sets. The following laws follow easily from the definition of regularity and from the Schröder rule and the Dedekind rule [46, 80].

50 (a) R regular $\Leftrightarrow R\widehat{R}R = R$
(b) R regular $\Leftrightarrow R\widehat{R} \subseteq R/R$
(c) R regular $\Leftrightarrow \widehat{R}$ regular
(d) RL regular
(e) R deterministic $\Rightarrow R$ regular
(f) R regular $\wedge$ Q regular $\Rightarrow R \cap Q$ regular

We need the following lemma to prove a connection between regular relations and conjugate kernels.

51 Lemma.

$$\kappa(R,Q)\kappa(R,Q)\widehat{} \subseteq R\widehat{R}$$

PROOF.

$$\begin{array}{ll} & \kappa(R,Q)\kappa(R,Q)\widehat{} \\ = & \qquad \{ \text{ law 39(a) } \} \\ & (R/Q \cap R\widehat{Q})(R/Q \cap R\widehat{Q})\widehat{} \\ \subseteq & \qquad \{ \text{ 12(g) } \} \\ & R\widehat{Q}(\widehat{Q}\backslash\widehat{R}) \\ \subseteq & \qquad \{ \text{ law 20(d) } \} \\ & R\widehat{R} \ . \qquad \Box \end{array}$$

The conjugate kernel preserves regularity.

52 Proposition. *If relation R is regular then $\kappa(R,Q)$ is regular.*

Proof.

$$\begin{array}{ll}
 & \kappa(R,Q)\kappa(R,Q)^\frown\kappa(R,Q) \\
\subseteq & \qquad \{ \text{ Lemma 51 } \} \\
 & R\widehat{R}\kappa(R,Q) \\
\subseteq & \qquad \{ \ R \text{ regular and law 50(b) } \} \\
 & (R/R)\kappa(R,Q) \\
= & \qquad \{ \text{ definition of conjugate kernel } \} \\
 & (R/R)(R/Q \cap L\widehat{Q}) \\
\subseteq & \qquad \{ \text{ 15(q) and 20(e) } \} \\
 & R/Q \cap L\widehat{Q} \\
= & \qquad \{ \text{ definition of } R/Q \cap L\widehat{Q} \text{ (38) } \} \\
 & \kappa(R,Q) \ . \qquad \Box
\end{array}$$

A special class of regular relations is the set of *partial equivalence relations*. They are especially useful in the construction of loops.

53 Definition. A relation R is said to be *a partial equivalence relation (PER)* if and only if it is symmetric ($R = \widehat{R}$) and transitive ($RR \subseteq R$). $\Box$

54 Proposition. *Relation R is a PER if and only if $R = R\widehat{R}$.*

Proof. For the "⇒" implication, we observe first that

$$R\widehat{R} = RR \subseteq R \ .$$

Then, we have

$$\begin{array}{ll}
 & \textbf{true} \\
\Leftrightarrow & \qquad \{ \text{ Dedekind } \} \\
 & RL \cap I \subseteq R\widehat{R} \\
\Rightarrow & \qquad \{ \text{ observation above } \} \\
 & RL \cap I \subseteq R \\
\Rightarrow & \qquad \{ \text{ monotonicity of } \circ \ \} \\
 & (RL \cap I)R \subseteq RR \\
\Leftrightarrow & \qquad \{ \text{ vector laws } \} \\
 & R \subseteq RR \\
\Leftrightarrow & \qquad \{ \text{ definition of PER: } R = \widehat{R} \ \} \\
 & R \subseteq R\widehat{R} \ .
\end{array}$$

For the "⇐" implication, we have

$$\begin{array}{ll}
 & R = R\widehat{R} \\
\Leftrightarrow & \qquad \{ \ {}^\frown \text{ laws } \} \\
 & \widehat{R} = \widehat{R\widehat{R}} \\
\Leftrightarrow & \qquad \{ \ {}^\frown \text{ laws } \} \\
 & \widehat{R} = R\widehat{R} \ .
\end{array}$$

Hence, we have $R = \widehat{R}$. Also,

$$\begin{array}{ll}
 & R\widehat{R} = R \\
\Rightarrow & \qquad \{ \text{ definition of } \subseteq \ \} \\
 & R\widehat{R} \subseteq R \\
\Leftrightarrow & \qquad \{ \ R = \widehat{R} \ \} \\
 & RR \subseteq R \ . \qquad \Box
\end{array}$$

55 Proposition. *If relation R is a PER, then R is regular, $R /\!\!/ R = R$, $R \backslash\!\!\backslash R = R$ and $R = R^{\boxast}$.*

Proof.

$$R\widehat{R}R = R\widehat{R}\widehat{R} = R\widehat{R} = R \ ,$$

$$R /\!\!/ R = \kappa(R, R) = R\widehat{R} = R \ ,$$

$$R \backslash\!\!\backslash R = \kappa(\widehat{R}, \widehat{R})^{\frown} = \kappa(R, R)^{\frown} = \widehat{R} = R \ ,$$

$$R = R /\!\!/ R = (R /\!\!/ R)^{\boxast} = R^{\boxast} \ . \qquad \Box$$

As a last useful operation on the semilattice, we mention intersection by a left vector. It features many simple properties, like

56 (a) $Q \sqsubseteq R \ \Rightarrow \ t \cap Q \sqsubseteq t \cap R$ (b) $t \cap R \sqsubseteq R$
(c) $t \sqsubseteq u \Rightarrow t \cap P \sqsubseteq u \cap P$ (d) $P \sqsubseteq Q \Rightarrow t \cap P \sqsubseteq t \cap Q$.

To summarize the precedence of relational operators, from highest to lowest we have

$$(^{-}, ^{\frown}, *, ^{\boxast}, +, ^{\boxplus}), (\circ, \Box, /, /\!\!/, \backslash, \backslash\!\!\backslash), (\cap, \sqcap), (\cup, \sqcup) \ .$$

3.4.3.1 Demonic Operations versus Angelic Operations

Relations and the relational operators can be used to define the semantics of programming languages. In this context, the angelic operators $\cup$ and $\circ$ serve to give an *angelic semantics*, by providing that an execution will terminate successfully whenever there is a possibility for successful termination. By contrast, the demonic operators provide that an execution will fail to terminate successfully whenever failure is possible. For all their metaphysical drawbacks, demonic operators offer us a crucial advantage: they are more adequate for the purpose of correctness preservation, because they are monotonic with respect to refinement; we will come back to this matter in more detail in Chapter 4.

The definitions of demonic composition, demonic join and demonic meet seem complex at first hand. This complexity may raise the question of their legitimacy for the purpose of program construction. Of course, their monotonicity with respect to refinement makes them acceptable candidates, but other monotonic operators wrt $\sqsubseteq$ exist, whence monotonicity is not sufficient. Another fundamental characteristic has guided our choice: these operators are faithful to the notions of sequential execution, sum and choice of requirements.

In the next paragraphs, we take an operational point of view to justify the definitions of demonic composition, demonic join and demonic meet. We use relations P and Q given below for this purpose.

$$P \triangleq \{(1,1), (1,2), (3,2), (3,3)\} \ ,$$

$$Q \triangleq \{(2,1), (2,2), (3,1), (3,3)\} \ .$$

We consider three different combinations of P and Q: sequence, sum and choice. For each combination, we ask a *demon* to execute a computation. Recall the interpretation of a specification R: if a computation starts in a state s in the domain of relation R, then it must terminate in a state s' such that $(s,s') \in R$; if a computation starts in a state not in the domain of R, then any outcome is acceptable, including nontermination. The demon will consider all possible paths and always select those that lead to nontermination whenever it is possible. Hereby, the demon will satisfy a necessary condition for monotonicity wrt to $\sqsubseteq$. Let us explain this matter in more details.

When refining a relation R, one may only reduce its nondeterminacy or increase its domain. By reducing nondeterminacy, one can only remove paths leading to a failure. By increasing the domain, one can only add new successful paths. Hence, in the case where one refines P into P' and Q into Q', if the combination of P and Q terminates for an initial state, then the combination of P' and Q' also terminates; moreover, if the combination of P' and Q' can terminate in state s', then the combination of P and Q can also terminate in s'. Whence, the combination of P' and Q' refines the combination of P and Q. In the next paragraphs, we justify the definitions of the demonic operators corresponding to the different combinations mentioned above.

Sequential Execution If we ask the demon to execute in sequence a computation satisfying P followed by a computation satisfying Q, what happens, knowing the malevolent nature of the demon? If the computation is started on input 1, the demon must select either 1 or 2 as output, in order to satisfy the specification. Of course the demon selects 1, because Q is not defined for input 1; hence the computation of Q aborts. For input 3, the demon is powerless: Q is defined for any possible output of P; hence it is unable to abort the computation. For any other input, P is not defined, so the demon aborts the computation.

The lesson learned from this example is the following: if the set of images of an input s by relation P is not included in the domain of Q, then the demon may arrange for the computation to abort; otherwise, it is forced to produce an output s' satisfying the following predicate:

$$\exists t : (s,t) \in P \wedge (t,s') \in Q \ .$$

This is precisely the definition of the usual (angelic) composition PQ. Using this lesson, we derive the following definition of demonic composition:

$$P \mathbin{\scriptstyle\Box} Q \triangleq \{(s,s') | (s,s') \in PQ \wedge P.s \subseteq dom(Q)\} \ .$$

The following calculation shows that this definition corresponds to the set-theoretic interpretation of the algebraic definition of demonic composition.

$$\begin{array}{ll}
 & (s,s') \in PQ \cap \overline{P\overline{QL}} \\
\Leftrightarrow & \qquad \{ \text{ interpretation of } \cap \ \} \\
 & (s,s') \in PQ \wedge (s,s') \in \overline{P\overline{QL}} \\
\Leftrightarrow & \qquad \{ \text{ interpretation of } ^{-} \ \} \\
 & (s,s') \in PQ \wedge \neg((s,s') \in P\overline{QL}) \\
\Leftrightarrow & \qquad \{ \text{ interpretation of } \circ \ \} \\
 & (s,s') \in PQ \wedge \neg(\exists t : (s,t) \in P \wedge (t,s') \in \overline{QL}) \\
\Leftrightarrow & \qquad \{ \text{ interpretation of } ^{-} \ \} \\
 & (s,s') \in PQ \wedge \neg(\exists t : (s,t) \in P \wedge \neg((t,s') \in QL)) \\
\Leftrightarrow & \qquad \{ \text{ interpretation of } \circ \ \}
\end{array}$$

$$\begin{array}{ll}
& (s,s') \in PQ \wedge \neg(\exists t : (s,t) \in P \wedge \neg(\exists u : (t,u) \in Q \wedge (u,s') \in L)) \\
\Leftrightarrow & \qquad \{ \text{ interpretation of } L \text{ } \} \\
& (s,s') \in PQ \wedge \neg(\exists t : (s,t) \in P \wedge \neg(\exists u : (t,u) \in Q)) \\
\Leftrightarrow & \qquad \{ \text{ definition of } \mathit{dom}(Q) \text{ } \} \\
& (s,s') \in PQ \wedge \neg(\exists t : (s,t) \in P \wedge \neg(t \in \mathit{dom}(Q))) \\
\Leftrightarrow & \qquad \{ \text{ remove } \neg \text{ } \} \\
& (s,s') \in PQ \wedge \forall t : (s,t) \notin P \vee t \in \mathit{dom}(Q) \\
\Leftrightarrow & \qquad \{ \text{ property of } \Rightarrow \text{ } \} \\
& (s,s') \in PQ \wedge \forall t : (s,t) \in P \Rightarrow t \in \mathit{dom}(Q) \\
\Leftrightarrow & \qquad \{ \text{ definition of } P.s \text{ } \} \\
& (s,s') \in PQ \wedge \forall t : t \in P.s \Rightarrow t \in \mathit{dom}(Q) \\
\Leftrightarrow & \qquad \{ \text{ definition of } \subseteq \text{ } \} \\
& (s,s') \in PQ \wedge P.s \subseteq \mathit{dom}(Q) \ .
\end{array}$$

Sum We now ask our demon to execute a computation that satisfies P and that satisfies Q. For input 1, the demon must produce either 1 or 2 to satisfy P. Relation Q is not defined for input 1; hence, the demon is allowed to produce any output, including none. In this case, to satisfy P and Q simultaneously, the demon must select an output from P. The analysis is similar for input 2: P is not defined but Q is; hence the demon must select an output from Q. For input 3, both P and Q are defined. To satisfy them simultaneously, the demon must produce an output that is allowed by both P and Q. His sole option is output 3. One might ask what happens if there is no output common to P and Q? Well, in that case the demon is unable to produce a computation that satisfies P and Q simultaneously. For any other input, P and Q are not defined, so the demon aborts.

The reader may recognize the description of the demonic join operator in the previous example. Indeed, the specification that the demon must obey when satisfying P and Q simultaneously is exactly the join $P \sqcup Q$:

$$P \sqcup Q \triangleq (P \cap \overline{QL}) \ \cup \ (P \cap Q) \ \cup \ (\overline{PL} \cap Q) \ .$$

The first term describes the behavior when P is defined but not Q. The second term represents the behavior when both are defined, and the third term describes the behavior when Q is defined but not P. An undefined join corresponds to the case where there is an input for which there exists non-common output.

Choice We now ask our demon to execute a computation that satisfies P or that satisfies Q. The demon selects which relation to satisfy for each input. We have no control over its selection. For input 1, the demon must produce either 1 or 2 to satisfy P, but it is not required to produce any output to satisfy Q. Therefore, the demon selects Q, and he aborts the computation. The analysis for input 2 is similar: P is not defined but Q is; so the demon selects P and aborts the computation. For input 3, both P and Q are defined. The demon is then allowed to produce any output from P or Q. On any other input, the demon aborts the computation.

The specification that the demon follows is given exactly by the demonic meet operator:

$$P \sqcap Q \triangleq PL \cap QL \cap (P \cup Q) \ .$$

This represents the union of P and Q prerestricted by the intersection of the domain of P and the domain of Q.

3.5 The Prerestriction Ordering

In defining the semantics of iterative constructs in an extension to Dijkstra's language of guarded commands, Nelson [74] introduced the *prerestriction ordering* on predicate transformers (Nelson calls it the *approximate ordering*, in reference to domain theory). This ordering enjoys strong fixpoint properties. Nguyen [75] derived a relational set-theoretic model of this ordering, which we now translate into an equivalent abstract relational definition. We will use this ordering to prove properties of iterative constructs.

57 Definition. A relation Q is said to be a *prerestriction* of relation R (noted $Q \preceq R$) if and only if

$$Q = QL \cap R \ . \qquad \Box$$

The following proposition provides insights in the kind of relationship that exists when $Q \preceq R$.

58 Proposition. *$Q \preceq R \Leftrightarrow Q \subseteq R \wedge Q \sqsubseteq R$.*

Proof.

$$\begin{array}{ll}
& Q \subseteq R \wedge Q \sqsubseteq R \\
\Leftrightarrow & \qquad \{ \text{ definition of } \sqsubseteq (24) \ \} \\
& Q \subseteq R \wedge QL \subseteq RL \wedge QL \cap R \subseteq Q \\
\Leftrightarrow & \qquad \{ \ Q \subseteq QL \text{ and } 12(\text{g}),\ Q \subseteq R \Rightarrow QL \subseteq RL \ \} \\
& Q \subseteq QL \cap R \wedge QL \cap R \subseteq Q \\
\Leftrightarrow & \qquad \{ \text{ antisymmetry of } \subseteq \} \\
& Q = QL \cap R \\
\Leftrightarrow & \qquad \{ \text{ definition of } \preceq (57) \ \} \\
& Q \preceq R \ . \qquad \Box
\end{array}$$

It is easy to see that $\preceq$ is an ordering, since the previous proposition provides that $\preceq$ is the intersection of two orderings. As usual, we write $Q \prec R$ if and only if $Q \preceq R \wedge Q \neq R$. The next proposition states an important property of relation $\preceq$ that will be used later to prove the existence of fixpoints.

59 Proposition. *Relations under the prerestriction ordering form a CPO.*

Proof. Let j range over an index set J. We show that the least upper bound of a chain $\{P_j\}$ is $\bigcup_j P_j$.

1. $\bigcup_j P_j$ is an upper bound of $\{P_j\}$. Let $k \in J$.

$$\begin{array}{ll}
& P_k \preceq \bigcup_j P_j \\
\Leftrightarrow & \qquad \{ \text{ definition of } \preceq (57) \ \} \\
& (\bigcup_j P_j) \cap P_k L \ = \ P_k \\
\Leftrightarrow & \qquad \{ \text{ distributivity } \} \\
& (\bigcup_{j:P_j \preceq P_k} P_j \cap P_k L) \cup (\bigcup_{j:P_k \prec P_j} P_j \cap P_k L) \ = \ P_k \\
\Leftrightarrow & \qquad \{ \ P_j \preceq P_k \Rightarrow P_j \subseteq P_k, P_k \subseteq P_k L, \text{ definition of } \subseteq (5) \ \} \\
& P_k \cup \bigcup_{j:P_k \prec P_j} P_j \cap P_k L \ = \ P_k \\
\Leftrightarrow & \qquad \{ \text{ Definition 57, 12(a) } \}
\end{array}$$

$P_k \cup P_k \;=\; P_k$

$\Leftrightarrow$ $\{$ idempotence (12(a)) $\}$

true .

2. $\bigcup_j P_j$ is a prerestriction of any upper bound Q of $\{P_j\}$

$\bigcup_j P_j \preceq Q$

$\Leftrightarrow$ $\{$ definition of $\preceq$ (57) $\}$

$(\bigcup_j P_j)L \cap Q = \bigcup_j P_j$

$\Leftrightarrow$ $\{$ distributivity $\}$

$\bigcup_j P_j L \cap Q = \bigcup_j P_j$

$\Leftrightarrow$ $\{$ hypothesis $P_j \preceq Q$, Proposition 57 $\}$

$\bigcup_j P_j = \bigcup_j P_j$

$\Leftrightarrow$

true . $\square$

The following properties hold for the prerestriction ordering.

60 (a) $P \preceq Q \Rightarrow P \mathbin{\scriptstyle\square} R \preceq Q \mathbin{\scriptstyle\square} R$
(b) $Q \preceq R \Rightarrow P \mathbin{\scriptstyle\square} Q \preceq P \mathbin{\scriptstyle\square} R$
(c) $Q \preceq R \Rightarrow v \cap P \cup \overline{v} \cap Q \preceq v \cap P \cup \overline{v} \cap R$
(d) $P \preceq Q \Rightarrow PR \preceq QR$
(e) $P \preceq Q \Rightarrow P \sqcap R \preceq Q \sqcap R$

Proof. Law 60(b) and (c) hold since they are valid when $\preceq$ is replaced by $\subseteq$ or $\sqsubseteq$. For (a), assume $P \preceq Q$. Then

$P \mathbin{\scriptstyle\square} R$

$=$ $\{$ hypothesis and definition of $\preceq$ (57) $\}$

$(PL \cap Q) \mathbin{\scriptstyle\square} R$

$=$ $\{$ law 34(l) $\}$

$Q \mathbin{\scriptstyle\square} R \cap PL$

$\subseteq$ $\{$ 12(e) $\}$

$Q \mathbin{\scriptstyle\square} R$.

We also have

$P \preceq Q$

$\Rightarrow$ $\{$ Proposition 58 $\}$

$P \sqsubseteq Q$

$\Rightarrow$ $\{$ monotonicity of $\mathbin{\scriptstyle\square}$ (34(d)) $\}$

$P \mathbin{\scriptstyle\square} R \sqsubseteq Q \mathbin{\scriptstyle\square} R$.

The result follows from Proposition 58. For (d), assume $P \preceq Q$. We have

$P \preceq Q$

$\Rightarrow$ $\{$ Proposition 58 $\}$

$P \subseteq Q$

$\Rightarrow$ $\{$ monotonicity of $\circ$ (15(f)) $\}$

$PRL \subseteq QRL$.

We also have

$$
\begin{array}{ll}
 & QR \cap PRL \\
= & \qquad \{ \text{ hypothesis and definition of } \preceq \text{ (57) } \} \\
 & QR \cap (Q \cap PL)RL \\
= & \qquad \{ \text{ 15(q) } \} \\
 & QR \cap QRL \cap PL \\
= & \qquad \{ \ QR \subseteq QRL, \text{ 15(q) } \} \\
 & (Q \cap PL)R \\
= & \qquad \{ \text{ hypothesis } P \preceq Q \text{ and definition of } \preceq \text{ (57) } \} \\
 & PR \ .
\end{array}
$$

For (e), we have

$$P \preceq Q \Rightarrow P \sqsubseteq Q \Rightarrow P \sqcap R \sqsubseteq Q \sqcap R \ .$$

Also,

$$
\begin{array}{ll}
 & P \sqcap R \\
= & \qquad \{ \text{ definition of } \sqcap \ \} \\
 & PL \cap RL \cap (P \cup R) \\
= & \qquad \{ \text{ hypothesis } P \preceq Q \text{ and definition of } \preceq \text{ (57) } \} \\
 & PL \cap RL \cap ((PL \cap Q) \cup R) \\
\subseteq & \qquad \{ \text{ hypothesis } P \preceq Q \text{ implies } PL \subseteq QL \ \} \\
 & QL \cap RL \cap ((QL \cap Q) \cup R) \\
= & \qquad \{ \ Q \subseteq QL \ \} \\
 & QL \cap RL \cap (Q \cup R) \\
= & \qquad \{ \text{ definition of } \sqcap \ \} \\
 & Q \sqcap R \ . \qquad \Box
\end{array}
$$

This completes the presentation of our mathematical foundations.

Part II

A Logic of Program Construction by Parts

Chapter 4

A Language for Program Construction by Parts

As with any complex artefact, a specification may be greatly improved, and its analysis made easier, by the proper application of structuring devices. The discussions of the previous chapter lead us to believe that the join in the refinement ordering is a natural device for structuring specifications. Furthermore, the demonic join operator is orthogonal to the traditional structuring devices of programming languages such as alternation, conditional, iteration, and recursion —in the sense that a demonic join does not dictate a specific program structure, nor does it favor one particular program structure over another; as such, the demonic join operator fosters the practice of good specification, as advocated by several authors in the past [51, 60, 76].

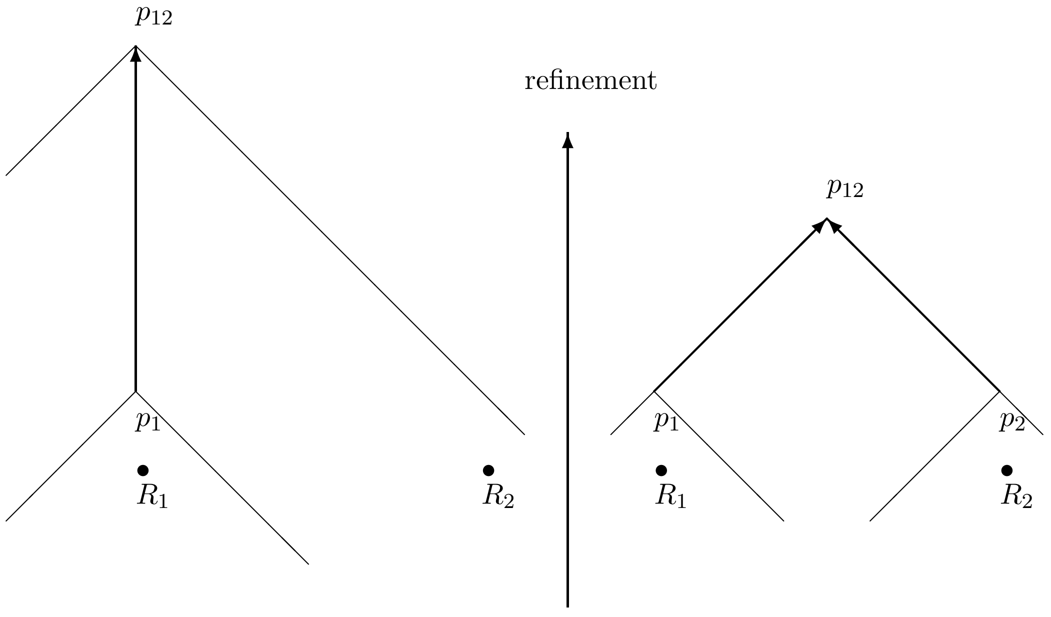
Our purpose, in this chapter, is to define a language called *Utica* which supports the stepwise transformation of a specification into a program; to do so, our language must support both the structuring of specifications (using demonic joins) and the structuring of programs (using constructs for sequence, alternation, conditional, and iteration). Hence the derivation of a program from a specification will proceed from a description that is structured exclusively as a specification (for the sake of abstraction [51, 60, 76]) to a description that is structured exclusively as a program (for the sake of executability). First, we discuss the programming paradigm that *Utica* is designed to support.

4.1 A Program Construction Paradigm

The interpretation of the demonic join of two specifications as the sum of their input/output information leads us to consider the demonic join as a natural structuring device of specifications: complex specification are defined as the demonic join of simpler specifications. Given a specification structured as $R_1 \sqcup R_2$, we address the question of how to construct a program correct with respect to R. We see two separation-of-concerns approaches:

Hiking Up the Lattice. We consider specification R_1, find a program p_1 correct with respect to it; then we attempt to transform p_1 so as to make it correct with respect to R_2 while preserving its correctness with respect to R_1. In lattice-theoretic terms, this amounts to pushing p_1 up the lattice until it is greater than R_2. See Figure 4.1(a).

Computing the lub of Programs. We consider each subspecification in turn, and construct programs for each, say p_1, and p_2. Then we compose these programs by the demonic join operator to find a solution to R. The main difficulty with this option is, of course, how do



(a) Hiking Up the Lattice (b) Computing *lub* of Programs

Figure 4.1: Two Paradigms of Constraint Programming

we implement the *demonic join* as a programming construct? Failing that, the question is then how do we get rid of demonic join constructs that appear in the specification? See Figure 4.1(b).

In Figure 4.1, the cones represented on a program p describe all the specifications with respect to which the program is correct; hence, e.g., if R_1 appears in the cone based on p_1, we know that p_1 is correct with respect to R_1.

Both of these paradigms require that we use a notation which allows us to express nondeterminacy: the programs we represent in this notation must be sufficiently undefined, so that it is possible to refine them further to accommodate additional requirements (than those for which they are originally written). In the sequel, we concentrate our discussions on the second paradigm.

4.2 Design Principles

It appears from the discussion above that the first order of business is to design a language to support the program construction paradigm that we propose. Before we present the details of the language we pause here to outline the principles that have driven the design of this language.

Nondeterminacy. Clearly, the language must be nondeterministic, i.e., it must allow us to describe situations where the output is only partly determined as a function of the input; this is essential if we are to combine two *component* programs into an aggregate program that has the functional properties of both components. We go one step further: the language must allow us to represent programs that are arbitrarily *undefined* (in the sense of the *refinement* ordering), i.e., arbitrarily low in the lattice of specifications; this means not only that they are nondeterministic, but also that they are partial (i.e., may be undefined for some inputs).

Simplicity. The overriding consideration in designing this language is the simplicity of its syntax (simple elementary statements, uniform compound statements) and its semantics (each construct is captured by a simple relational formula). We consent to sacrifice expressive power for the sake of simplicity.

Economy of Concept. The denotations of elementary statements in our language are relations; the denotations of constructs are relational operators.

Monotonicity. All the constructs of the language are monotonic with respect to the refinement ordering, in the following sense: for each construct, say $\gamma(R, R')$, where R and R' are statements of the language, whenever R (or R') is refined, so is $\gamma(R, R')$. This property is essential to our design paradigm, in the following sense: as soon as we have decided to structure our programming solution as $\gamma(R, R')$, we can refine (i.e., raise, in the lattice) R and R' independently, secure in the knowledge that the γ combination of their refinements is a refinement of $\gamma(R, R')$.

By adhering closely to a relational interpretation we gain the following advantage: we do not have to develop a specific calculus of programming for our language; rather, all the body of laws and principles of relational calculus [82, 86] become our (inherited) programming calculus.

4.3 The Utica Language

4.3.1 Syntax

It is customary when defining a programming language to present the *concrete* syntax and the *abstract* syntax. The concrete syntax describes the legal sentences that the programmer may write. It may include special constructs (syntactic sugars) that make the life of the programmer easier. The designer of the language uses the abstract syntax to make *his* life easier in defining the semantics. The concrete syntax for Utica is described by the following BNF grammar:

$\langle \textit{Program-Utica} \rangle$ ::= **Specification** $\langle \textit{pgm-name} \rangle$ $\langle \textit{block} \rangle$

$\langle \textit{pgm-name} \rangle$ ::= $\langle \textit{identifier} \rangle$

$\langle \textit{block} \rangle$ ::= **var** $\langle \textit{variable-declaration} \rangle$ $\langle \textit{statement} \rangle$

$\langle \textit{variable-declaration} \rangle$ ::= $\langle \textit{declaration} \rangle$ | $\langle \textit{declaration} \rangle$, $\langle \textit{variable-declaration} \rangle$

$\langle \textit{declaration} \rangle$ ::= $\langle \textit{variable-list} \rangle$: $\langle \textit{type-expression} \rangle$

$\langle \textit{variable-list} \rangle$ ::= $\langle \textit{variable} \rangle$ | $\langle \textit{variable} \rangle$, $\langle \textit{variable-list} \rangle$

$\langle \textit{variable} \rangle$ ::= $\langle \textit{identifier} \rangle$

$\langle \textit{statement} \rangle$ ::= $\langle \textit{pure-Utica-stmt} \rangle$ | $\langle \textit{extended-Utica-stmt} \rangle$

$\langle$*pure-Utica-stmt*$\rangle$ ::= $\langle$*elementary-stmt*$\rangle$ | $\langle$*compound-stmt*$\rangle$ | $\langle$*local-block*$\rangle$

$\langle$*elementary-stmt*$\rangle$ ::=
  skip($\langle$*condition*$\rangle$) |
  establish($\langle$*condition*$\rangle$) |
  preserve($\langle$*condition*$\rangle$) |
  rel $\langle$*definition*$\rangle$ **ler**

$\langle$*compound-stmt*$\rangle$ ::=
  $\langle$*condition*$\rangle$ $\rightarrow$ $\langle$*statement*$\rangle$ |
  seq $\langle$*statement-list*$\rangle$ **qes** |
  glb $\langle$*statement-list*$\rangle$ **blg** |
  clo $\langle$*statement*$\rangle$ **olc** |
  lub $\langle$*statement-list*$\rangle$ **bul**|

$\langle$*local-block*$\rangle$ ::= $\langle$*block*$\rangle$

$\langle$*extended-Utica-stmt*$\rangle$ ::=
  if $\langle$*condition*$\rangle$ **then** $\langle$*statement*$\rangle$ |
  if $\langle$*condition*$\rangle$ **then** $\langle$*statement*$\rangle$ **else** $\langle$*statement*$\rangle$ |
  while $\langle$*condition*$\rangle$ **do** $\langle$*statement*$\rangle$ |
  par $\langle$*statement-list*$\rangle$ **rap**

$\langle$*statement-list*$\rangle$ ::= $\langle$*statement*$\rangle$ | $\langle$*statement*$\rangle$, $\langle$*statement-list*$\rangle$

This grammar leaves unspecified the following nonterminal symbols:

$\langle$*identifier*$\rangle$, $\langle$*type-expression*$\rangle$, $\langle$*condition*$\rangle$ and $\langle$*definition*$\rangle$.

The generation of identifiers follow the same rules as in other programming languages (e.g., Pascal). Utica does not propose any atomic type or any type constructor for generating type expressions. Because the subject of our study is algorithmic refinement, our calculus is independent of which types are used. The specifier has the freedom to select any atomic type and any type constructor to specify his problem. In return, he has the responsibility of ensuring that his target programming language supports them*. The nonterminal symbol $\langle$*condition*$\rangle$ represents a logical expression constructed using variables, operations and constants from the types selected by the specifier. The nonterminal symbol $\langle$*definition*$\rangle$ represents a logical expression that may contain "primed" variables (e.g., x'). We may refer to such expressions as "primed logical expressions". Unprimed variables refer to the input of a computation whereas primed variables refer to the output of a computation.

The two production rules generating **if-then** statements and **if-then-else** statements make this grammar *ambiguous*. We adopt the traditional disambiguating rule that an **else** is matched to the closest **if**. In addition, binary constructs like $\rightarrow$ have precedence over **while** and **if**.

*If the programming language does not support the selected types, then the programmer may apply *data refinement* [21, 43, 70] to derive a (concrete) implementation of a solution.

4.3.2 Static Semantics

The *static semantics* of a language defines the legality of programs generated from a grammar. It is concerned with aspects like type-checking of expressions, declaration of variables before they are referred to and the scope of variable declarations. These rules can not be expressed in a BNF grammar. The rules in Utica are similar to the rules found in a programming language like Pascal.

A variable must be declared before it is used in a Utica statement. The scope of a variable declaration extends only on the statement following the declaration. For instance, in the following program fragment, variable x may be only used in statement p:

$$\textbf{seq var } x : T\ p,\ q\ \textbf{qes}\ .$$

Of course, p can be a compound statement generated from ⟨*compound-stmt*⟩, and all its components may refer to variable x. A variable must be unique in a variable declaration. The same variable may appear in different variable declarations. For instance, the statement p of the example above may contain another declaration of variable x, possibly with a different type (although in general this is not recommended as a good programming practice). The semantics and the scoping rules of a block shall be made clear in Sections 4.3.5 and 4.3.9.

4.3.3 Context, Space and State

Each statement in a Utica program is preceded by at least one declaration of variables. The *context* of a statement is the list of variables and types declared before the statement.

61 Definition. A *context* C is a triple $(V, \mathcal{T}, b)$ where:

V is a finite set of variables,
$\mathcal{T}$ is a finite set of types (sets),
b is a total function of $V \to \mathcal{T}$.

We call function b the *type binding* of the context. Contexts $C_1 = (V_1, \mathcal{T}_1, b_1)$ and $C_2 = (V_2, \mathcal{T}_2, b_2)$ are said to be disjoint if and only if $V_1 \cap V_2 = \emptyset$. □

We define three operations on contexts that will be used in the semantics of variable declarations and blocks.

62 Definition. The *extension* of context $C_1 = (V_1, \mathcal{T}_1, b_1)$ by context $C_2 = (V_2, \mathcal{T}_2, b_2)$ is defined as

$$C_1 + C_2 \triangleq (V_1 \cup V_2, \mathcal{T}_1 \cup \mathcal{T}_2, b) \text{ where}$$

$$b.v \triangleq \text{ if } v \in V_2 \text{ then } b_2.v \text{ else } b_1.v\ . \qquad \Box$$

An extension contains the variables of the two contexts. Variables common to the two contexts get their types from the second context.

63 Definition. The *difference* of context $C_1 = (V_1, \mathcal{T}_1, b_1)$ and context $C_2 = (V_2, \mathcal{T}_2, b_2)$ is defined as

$$C_1 - C_2 \triangleq (V_1 \setminus V_2, \mathcal{T}_1, b_1.i_{V_1 \setminus V_2})\ . \qquad \Box$$

A difference contains the variables appearing in the first context only with the type binding of the first context.

64 Definition. The *overlap* of contexts $C_1 = (V_1, \mathcal{T}_1, b_1)$ and $C_2 = (V_2, \mathcal{T}_2, b_2)$ is defined as

$$C_1 * C_2 \triangleq (V_1 \cap V_2, \mathcal{T}_1, b_1.i_{V_1 \cap V_2}) \ . \qquad \Box$$

An overlap of contexts contains the variables common to the two contexts with the type bindings of the first context. The next definition introduces the notions of space and state which will be used in the sequel to provide a relational semantics for Utica.

65 Definition. The *space* S_C generated from a context $C = (V, \mathcal{T}, b)$ is the set of all total functions $s : V \to \bigcup_{T \in \mathcal{T}} T$ such that

$$\forall v \in V : s.v \in b.v \ .$$

An element s of S_C is called a *state*. $\Box$

4.3.4 Denotational Semantics

We have designed our language so that each Utica construct is easily interpreted by a simple relational expression. The denotational semantics of our language is given by a function mapping each Utica program to a mathematical object called a *denotation*. The denotation of a Utica program is a relation on the space generated from the context of the program. Relations are not necessarily total. The domain of a relation represents the set of initial states for which a program terminates. If a program may not or does not terminate for a given initial state, then the associated relation is undefined for that state. We do not distinguish between different causes of nontermination like divergence (infinite loops) or failure (e.g., division by zero). Our spaces do not contain a special element to denote nontermination. The denotational semantics that we are proposing differs in some aspects from the "usual" denotational semantics [71], where denotations are elements of *semantic domains* [35] and nontermination is denoted by the least element of a semantic domain.

Compositionality is a key aspect of a denotational semantics. Our semantic function is defined inductively: denotations of elementary statements are relations, and the denotation of a compound statement is given by a formula based on the denotations of the components. There is a close correspondence between the compound statements and the demonic operations of a relation algebra. The semantics of a Utica program is given by the following function:

$$[\![-]\!] : (\textit{Program-Utica}) \to \textit{Relations} \ .$$

We refer to the function above as the semantic function. Its domain is a *phrase sort.* A phrase sort is the set of derivation trees generated from a nonterminal symbol. We distinguish between a phrase sort and a nonterminal symbol by enclosing the former between "()" and enclosing the latter between "<>". The range of the semantic function is the set of all relations definable from $\mathcal{C}$, the set of all contexts generated from the elementary types and the type constructors. In the definition of the semantic function $[\![\text{-}]\!]$, we use an auxiliary function $\Gamma[\![\text{-}]\!]$ to compute the space of the denotation of a Utica program. It maps a declaration of variables to a context.

$$\Gamma[\![-]\!] : (\textit{variable-declaration}) \to \mathcal{C} \ .$$

The semantics of a Utica program is defined by the following axiom:

$$\llbracket \textbf{Specification}\ \langle \textit{pgm-name} \rangle\ \textbf{var}\ \langle \textit{variable-declaration} \rangle \langle \textit{statement} \rangle \rrbracket \triangleq$$
$$\llbracket \langle \textit{statement} \rangle \rrbracket$$

The axiom above provides that the denotation of a Utica program is the denotation of its main statement. The denotation of a Utica program is defined on the space generated from the context of the main statement of the program, that is, from the context $\Gamma\llbracket\langle \textit{variable-declaration}\rangle\rrbracket$.

In the sequel, we will use some abbreviations to reduce the size of semantic expressions and calculations. We use t (possibly subscripted) to represent the derivation tree of $\langle \textit{condition} \rangle$; we use *plex* to represent the derivation tree of $\langle \textit{definition} \rangle$; we use p, q (possibly subscripted) to represent the derivation tree of $\langle \textit{statement} \rangle$; we use w to represent the derivation tree of $\langle \textit{variable-declaration} \rangle$. In axioms defining the semantic function $\llbracket\text{-}\rrbracket$, we usually do not mention explicitly the context of a denotation. We use $C = (V, \mathcal{T}, b)$ to denote the context of the statements for which the semantics is defined. We extend the refinement relation $\sqsubseteq$ to Utica programs in the following manner: if p, q are two Utica programs then $p \sqsubseteq q$ if and only if $\llbracket p \rrbracket \sqsubseteq \llbracket q \rrbracket$. We say that two Utica statements p and q are equivalent if and only if $\llbracket p \rrbracket = \llbracket q \rrbracket$. In proofs, we use upper case letters for the denotation of lower case variables representing Utica statements (e.g., P represents $\llbracket p \rrbracket$). If t is the drivation tree of a condition, we still use t to represent its denotation in proofs (because the denotation of a condition is a left vector). We may indicate the space of a relation by subscripting the relation with its associated context (e.g., P_C instead of P_{S_C}). In the next sections, we will define functions $\llbracket\text{-}\rrbracket$ and $\Gamma\llbracket\text{-}\rrbracket$.

4.3.5 Declaration of Variables

A declaration of variables may consist of a single declaration or of a list of declarations. We define the semantics of a single declaration first. Let $x_1, \ldots, x_n$ be variable identifiers and T be a type expression. The semantics of a derivation tree generated from the nonterminal symbol $\langle \textit{declaration} \rangle$ is:

$$\Gamma\llbracket x_1, \ldots, x_n : T \rrbracket \triangleq (V, \mathcal{T}, b) \text{ where}$$

$$\begin{aligned} & V = \{x_1, \ldots, x_n\} \quad , \\ & \mathcal{T} = \{T\} \quad , \\ & \forall j \in 1 \ldots n : b.x_j = T \quad . \end{aligned}$$

The semantics of a list of declarations is defined with the extension operation on contexts. In the above axiom, we use the symbol T appearing in the declaration to represent the type. Of course, it is an abuse of notation, because T is a syntactic element, not a set. Since we do not define which types are available in our language, it is simpler to use the same symbol for a syntactic element and its corresponding set.

$$\Gamma\llbracket \langle \textit{declaration} \rangle,\ \langle \textit{variable-declaration} \rangle \rrbracket \ \triangleq$$
$$\Gamma\llbracket \langle \textit{declaration} \rangle \rrbracket + \Gamma\llbracket \langle \textit{variable-declaration} \rangle \rrbracket$$

This completes the definition of function $\Gamma\llbracket\text{-}\rrbracket$.

4.3.6 Logical Expressions

Let $C = (V, \mathcal{T}, b)$ be the context of a condition t (i.e., an expression containing only unprimed variables). Let $t[s]$ denote, for all variables $x \in V$, the replacement of every free occurrence of variable x in t by $s.x$. The denotation of a condition is a relation on space S_C given by:

$$[\![t]\!] \triangleq \{(s,s')|t[s]\} \quad .$$

This relation is a left vector since variable s' does not appear in condition $t(s)$. The relation denoting a *primed* logical expression is quite similar. Let *plex* be a primed logical expression. Let *plex*$[s,s']$ denote, for all variables $x \in V$, the replacement of every free occurrence of variables x and x' in *plex* by $s.x$ and $s'.x$ respectively. The denotation of a primed logical expression is a relation on space S_C given by:

$$[\![plex]\!] \triangleq \{(s,s')|plex[s,s']\} \quad .$$

For simplicity, we assume that all operators of the data types are total functions, i.e., that they are defined for any value of the type. This completes the definition of the semantics of logical expressions.

4.3.7 Elementary Statements

We give below the semantics of elementary statements of *Utica*. For the sake of the reader's intuition, and although our language is not necessarily executable, we introduce each statement by a brief description of its imaginary operational semantics.

4.3.7.1 Skip

Given a logical expression t, statement **skip**(t) is defined for all states satisfying t, and returns the same state. Its semantics is defined by the following equation:

$$[\![\mathbf{skip}(t)]\!] \triangleq [\![t]\!] \cap I_C \quad ,$$

For $t \Leftrightarrow true$, we find that

$$[\![\mathbf{skip}(t)]\!] = L_C \cap I_C = I_C \quad .$$

This gives the traditional interpretation of the **skip** statement. See Figure 4.2.

4.3.7.2 Establish

Given a logical expression t, statement **establish**(t) is defined for all states, and produces a state for which the logical expression t evaluates to **true**. Its semantics is defined by the following equation:

$$[\![\mathbf{establish}(t)]\!] \triangleq \widehat{[\![t]\!]} \quad .$$

The semantics of statement **establish**(t) is defined by a right vector. We find that

$$[\![\mathbf{establish}(\mathbf{true})]\!] = L_C \quad .$$

This yields the statement that is defined on all initial states and returns an arbitrary output state (usually referred to as: *chaos*, or *havoc*). See Figure 4.3. For $t \Leftrightarrow$ **false**, we find

$$[\![\mathbf{establish}(\mathbf{false})]\!] = \emptyset \quad .$$

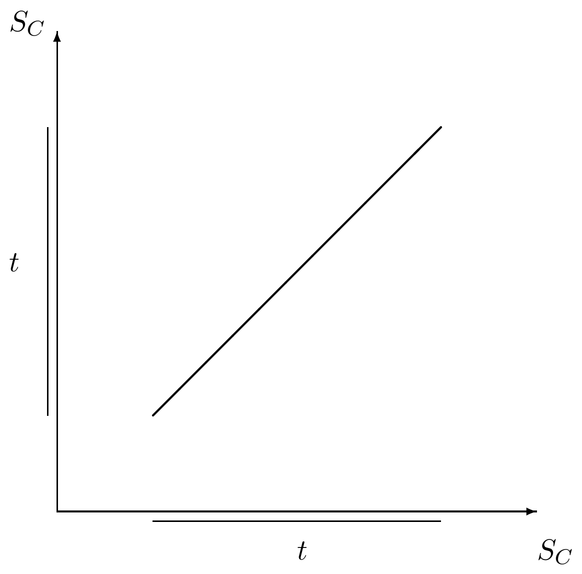


Figure 4.2: Relation defined by **skip**(t).

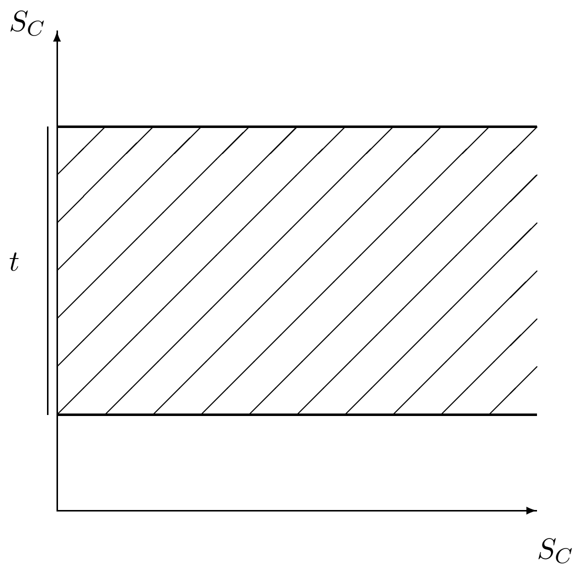


Figure 4.3: Relation defined by **establish**(t).

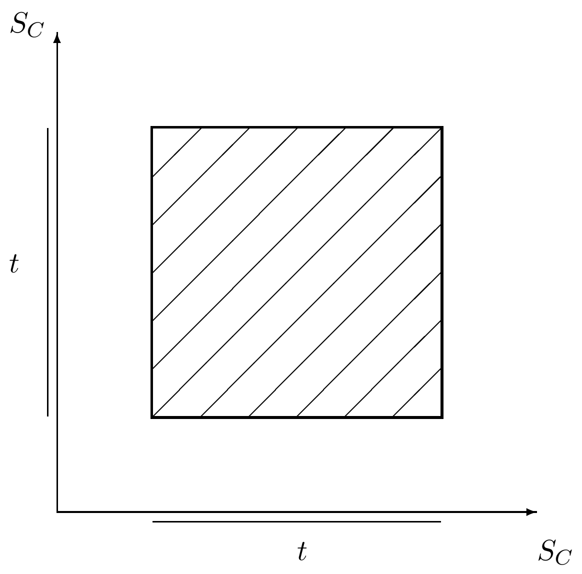


Figure 4.4: Relation defined by **preserve**(t).

4.3.7.3 Preserve

Given a logical expression t, statement **preserve**(t) is defined for all states satisfying t, and produces an arbitrary state satisfying t. Its semantics is defined by the following equation:

$$[\![\mathbf{preserve}(t)]\!] \triangleq [\![t]\!] \cap \widehat{[\![t]\!]} \quad .$$

This relation is a rectangle. In terms of the concrete theory of relations, this can be formulated by the following relation on S_C:

$$\{(s, s')|t[s] \wedge t[s']\}$$

which justifies the name of this statement. For $t \Leftrightarrow \mathbf{true}$, we find

$$[\![\mathbf{preserve}(\mathbf{true})]\!] = L_C \ .$$

See Figure 4.4.

4.3.7.4 Closed Form Relation

In *Utica*, a closed form relation is a statement of the form

```
rel
    plex
ler
```

where *plex* is a primed logical expression. Such statements are used in *Utica* to express a binary property between inputs and outputs. The semantic definition of closed form relations is given by the following equation:

$$[\![\mathbf{rel}\ \mathit{plex}\ \mathbf{ler}]\!] \triangleq [\![\mathit{plex}]\!] \quad .$$

4.3.8 Compound Statements: Pure Utica

All *Utica* compound statements have a uniform syntax: a list of *Utica* statements, separated by commas, and embedded within a pair made up of a three-letter keyword and its inverse.[†] We define the semantics of each compound statement by giving the relation associated with the statement as a function of the relations associated with the components.

4.3.8.1 Prerestriction

To restrict the relation of some *Utica* statement p to those input states that satisfy condition t, we write:

$$t \rightarrow p \quad .$$

We interpret this construct by means of the following equation:

$$[\![t \rightarrow p]\!] \triangleq [\![t]\!] \cap [\![p]\!] \ .$$

4.3.8.2 Sequential Combination

To combine two *Utica* statements p and q in sequence, we write them as follow:

seq
  $p,\ q$
qes .

We interpret this construct by means of the demonic composition operator, as given in the following equation:

$$[\![\mathbf{seq}\ p,\ q\ \mathbf{qes}]\!] \triangleq [\![p]\!] \,\Box\, [\![q]\!] \quad .$$

Unlike the angelic composition, the demonic composition is monotonic with respect to the refinement ordering, i.e., the product $R_1 \,\Box\, R_2$ is refined whenever R_1 or R_2 is refined (see laws 34 (c,d)). The semantics of an n-ary ($n > 2$) sequential combination is:

$$[\![\mathbf{seq}\ p_1, \ldots, p_n\ \mathbf{qes}]\!] \triangleq [\![\mathbf{seq}\ p_1, \ldots, p_{n-1}\ \mathbf{qes}]\!] \,\Box\, [\![p_n]\!] \quad .$$

4.3.8.3 Greatest Lower Bound

When we know of two *Utica* statements p and q which solve our problem, but have no basis for choosing between them, it is advantageous, for the sake of leaving future design options open, to combine them by the **glb** construct, as:

glb
  $p,\ q$
blg .

[†] The pattern of discriminating between lists by means of their enclosing symbols (rather than their separators) is not new: the set $\{a, b, c\}$ and the list (a, b, c) both use the comma for separation, and are distinguished by the enclosing brackets.

Imagine, e.g., that upon inspecting specification R we have determined that p and q both satisfy R; it is tempting to select one of them at random and proceed, but is ill-advised. Indeed, we may at a later stage have to satisfy another specification, say Q, which one of them satisfies but the other does not. The safest option is to take the **glb** of p and q and postpone the decision of which to choose to a later stage in the design process, when more information will be available. The semantics of this construct is defined by the following equation:

$$[\![\textbf{glb}\ p,\ q\ \textbf{blg}]\!] \triangleq [\![p]\!] \sqcap [\![q]\!] \quad .$$

Clearly, this construct is monotonic with respect to the refinement ordering, since its semantics is defined in terms of a lattice operator. The semantics of an n-ary $(n > 2)$ **glb** is:

$$[\![\textbf{glb}\ p_1, \ldots, p_n\ \textbf{blg}]\!] \triangleq [\![\textbf{glb}\ p_1, \ldots, p_{n-1}\ \textbf{blg}]\!] \sqcap [\![p_n]\!] \quad .$$

4.3.8.4 Closure

When we want to execute a *Utica* statement p an indefinite number of times, we write:

clo p **olc** .

This construct is quite useful in program construction by parts: complex specifications are often made up of a component that defines what must be preserved (between the input state and the output state), and a component that defines what must be achieved (by the output state); typically, the former is defined by an equivalence relation, while the latter is defined by a right vector. Most typically the closure construct allows us to refine the preservation component: what the closure statement provides is that as long as we are executing statement p, the desired property is preserved; because we do not know what property we are trying to achieve in the output, we do not know when to stop —hence the absence of a condition in the closure statement. See the example of Chapter 6.

The semantic definition of this construct is given by the following equation:

$$[\![\textbf{clo}\ p\ \textbf{olc}]\!] \triangleq [\![p]\!]^{\boxed{*}} \quad .$$

This formula is clearly monotonic in p, because it combines two monotonic operators: the demonic product, and the demonic meet. Intuitively, this equation can be justified as follows: because we do not know how many times p has to be executed, all we can infer about the execution of the closure statement is that it can be refined by any number of execution of p. The most refined relation satisfying this property is precisely the greatest lower bound of $\{[\![p]\!]^{\boxed{i}}\ | i \geq 0\}$, i.e., the demonic closure of p.[‡]

4.3.8.5 The Least Upper Bound

When we have two *Utica* statements and want to define a program that refines them both simultaneously, we write:

lub
    p, q
bul .

[‡]Should we take the least refined relation satisfying the aforementioned property, we would obtain the empty relation.

Of course simply writing p and q in a **lub** construct does not solve the problem of combining them to get a stronger program since we do not know how a compiler can generate code for a **lub** construct; transformation rules are needed to map this construct into a compilable program. This will be discussed in Chapter 5.

The semantics of the **lub** construct is defined by the following equations:

$$
\begin{array}{l}
cs([\![p]\!],[\![q]\!]) \Rightarrow \\
\quad [\![\textbf{lub}\ p,\ q\ \textbf{bul}]\!] \triangleq [\![p]\!] \sqcup [\![q]\!] \\
\\
cs([\![\textbf{lub}\ p_1,\ldots,p_{n-1}\ \textbf{bul}]\!],[\![p_n]\!]) \Rightarrow \\
\quad [\![\textbf{lub}\ p_1,\ldots,p_n\ \textbf{bul}]\!] \triangleq [\![\textbf{lub}\ p_1,\ldots,p_{n-1}\ \textbf{bul}]\!] \sqcup [\![p_n]\!] \quad .
\end{array}
$$

The **lub** statement is undefined if its components do not satisfy the consistency condition. Indeed, inconsistent specifications are specifications which contradict each other. For instance, one component may require the final value of variable x to be 1 whereas another may require the final value of x to be 2. It is impossible to satisfy these two requirements simultaneously.

Our definition of **lub** imposes a proof obligation to the specifier. It is an unfortunate situation, but it seems difficult to avoid. The usefulness of the **lub** statement outweighs, to our opinion, the burden of the proof obligation. Other refinement calculi are also facing similar problems, but for different reasons. For instance, in Morgan's calculus [68], one must prove that a specification statement is *feasible*, that is, for every initial state satisfying the precondition there exists a final state satisfying the postcondition. This proof obligation is due to the structure of a specification as a pair of predicates. In Hehner's practical theory of programming [41], one must prove that a specification is *implementable*, that is, the specification is defined for every initial value of the state variables. This proof obligation is due to the definition of refinement as logical implication. In our case, the proof obligation is not due to the specification model nor to the refinement model. It is due to the addition of a useful operator for structuring specifications.

Deciding on the appropriate moment to verify the consistency is an interesting issue. Two consistent specifications may be refined individually up to a point where they are no longer consistent. Should the specifier verify the consistency after every refinement of a **lub** component? The following proposition provides that if two specifications are inconsistent, their refinements will also be inconsistent. Thus, a specifier may freely apply any number of refinements to the components of a **lub** and check the consistency condition at the end only. If the final components are consistent, then all the intermediate components were also consistent. In fact, we will find in Section 5.1 that undefined **lub**'s will become obvious in the refinement process, because transformation rules are not applicable to an undefined **lub**.

66 Proposition. *Let p, p', q, q' be Utica programs such that $p \sqsubseteq p'$ and $q \sqsubseteq q'$. If relations $[\![p]\!]$ and $[\![q]\!]$ do not satisfy the consistency condition, then relations $[\![p']\!]$ and $[\![q']\!]$ do not satisfy the consistency condition.*

Proof. We prove the contrapositive, i.e., $cs(P',Q') \Rightarrow cs(P,Q)$. Assume $cs(P',Q')$.

$$
\begin{array}{ll}
 & PL \cap QL \\
= & \qquad \{\ PL \subseteq P'L \text{ and } QL \subseteq Q'L \text{ from hypotheses } p \sqsubseteq p' \text{ and } q \sqsubseteq q'\ \} \\
 & PL \cap QL \cap P'L \cap Q'L \\
= & \qquad \{\ P'L \cap Q'L = (P' \cap Q')L \text{ from hypothesis } cs(P',Q')\ \} \\
 & PL \cap QL \cap (P' \cap Q')L
\end{array}
$$

$=$ { 15(q) }
$(PL \cap QL \cap P' \cap Q')L$
$\subseteq$ { $P' \cap PL \subseteq P$ and $Q' \cap QL \subseteq Q$ from hypotheses $p \sqsubseteq p'$ and $q \sqsubseteq q'$ }
$(P \cap Q)L$. □

4.3.9 Local Block

Let w be a variable declaration and p be a statement. When it does not appear as the first statement of a Utica program, the block statement

var $w\ p$,

introduces local variables. The initial value of a local variable is arbitrary. The final value of a local variable is lost when the execution of the body of the local block is completed. Let C be the context of the local block above. If a variable in C is redeclared in w, its value is preserved by the execution of the local block. The denotation of statement p is a relation on the context C extended by the context generated from the variable declaration w. The denotation of the local block is the denotation of statement p *reduced* to the context of the local block. The next section introduces the mathematics supporting reduction.

4.3.9.1 Reduction

A reduction is defined in terms of two operations which we introduce in turn.

67 Definition. Let $C_1 = (V_1, \mathcal{T}_1, b_1)$ and $C_2 = (V_2, \mathcal{T}_2, b_2)$ be two disjoint contexts. The *projection* of $S_{C_1+C_2}$ to S_{C_1} is a relation Π_{C_1,C_2} from $S_{C_1+C_2}$ to S_{C_1} defined by:

$$\Pi_{C_1,C_2} \triangleq \{(s, s') \in S_{C_1+C_2} \times S_{C_1} | \exists x \in S_{C_2} : s = s' \cup x\} \ .$$

□

We may drop the subscripts of Π if there is no confusion on which contexts are under consideration. The following laws follow from the definition of Π.

68 (a) $\Pi L = L$ (b) $L\Pi = L$
(c) $\widehat{\Pi}\Pi = I$ (d) $\Pi\widehat{\Pi} \sqsubseteq I$

69 Definition. Let $C_1 = (V_1, \mathcal{T}_1, b_1)$ and $C_2 = (V_2, \mathcal{T}_2, b_2)$ be two disjoint contexts. Let Q, R be relations on spaces S_{C_1}, S_{C_2} respectively. The *coordinate-wise Cartesian product* of Q, R is given by:

$$Q \otimes R \triangleq \{(s_1 \cup s_2, s'_1 \cup s'_2) \in S_{C_1+C_2} \times S_{C_1+C_2} | (s_1, s'_1) \in Q \wedge (s_2, s'_2) \in R\} \ . \qquad \square$$

A *trivial space* is a space generated from a context with an empty set of variables. The only element of a trivial space is ϵ (see definition on page 8). Let L_ϵ be the universal relation on a trivial space. Relation L_ϵ contains only the pair (ϵ, ϵ). It stems from this that L_ϵ is the identity element of operation $\otimes$. The following laws hold when $\otimes$ is defined.

70 (a) $Q \otimes R = R \otimes Q$ (b) $P \otimes (Q \otimes R) = (P \otimes Q) \otimes R$
(c) $R \otimes L_\epsilon = L_\epsilon \otimes R = R$ (d) $P \sqsubseteq Q \Rightarrow P \otimes R \sqsubseteq Q \otimes R$
(e) $Q \sqsubseteq R \Rightarrow P \otimes Q \sqsubseteq P \otimes R$ (f) C_1, C_2 disjoint $\Rightarrow$
$P_{C_1} \otimes L_{C_2} = \Pi_{C_1,C_2} P_{C_1} \widehat{\Pi}_{C_1,C_2}$

71 Definition. Let C_1, C_2 be two contexts; let R be a relation on context $C_1 + C_2$; the *reduction* of relation R to context C_1 is given by:

$$R_{C_1+C_2} \downarrow \triangleq (\widehat{\Pi}_{C_1-C_2,C_2} \sqcap R \sqcap \Pi_{C_1-C_2,C_2}) \otimes I_{C_1*C_2} \quad . \qquad \Box$$

A reduction preserves variables appearing simultaneously in the context of the block and in the variable declaration of the block. A reduction $R \downarrow$ is defined for an initial state s if relation R is defined for every initial value of local variables appended to state s.

4.3.9.2 Semantics of Local Blocks

We define the semantics of a local block as follows:

$$[\![\mathbf{var}\ w\ p]\!]_C \triangleq ([\![p]\!]_{C+\Gamma[\![w]\!]}) \downarrow \quad .$$

This axiom provides that the denotation of a local block is the reduction of the denotation of the statement in the block. The following examples illustrate the definition of local blocks. Assume that the context of the local blocks in the following examples is variable x of type **Integer**:

$$[\![\mathbf{var}\ y : \mathbf{Integer}\ \mathbf{rel}\ y > 0 \wedge x' = y\ \mathbf{ler}]\!] = \emptyset \quad .$$

To see that the denotation of the above local block is empty, let Q be $[\![\mathbf{rel}\ y > 0 \wedge x' = y\ \mathbf{ler}]\!]$. The semantics of the local block is given by

$$\widehat{\Pi} \sqcap Q \sqcap \Pi \quad .$$

Recall the set-theoretic interpretation of demonic product:

$$\widehat{\Pi} \sqcap Q = \{(s, s') | (s, s') \in \widehat{\Pi}Q \wedge \widehat{\Pi}.s \subseteq dom(Q)\}$$

For each s in the domain of $\widehat{\Pi}$, the set $\widehat{\Pi}.s$ is given by

$$\widehat{\Pi}.s = \{s' | s'.x = s.x \wedge s.y \in \mathbf{Integer}\} \quad .$$

The domain of Q is

$$dom(Q) = \{s' | s'.y \in \mathbf{Integer} \wedge s.y > 0\} \quad .$$

Thus, for all s in the domain of $\widehat{\Pi}$, the following inclusion *does not* hold

$$\widehat{\Pi}.s \subseteq dom(Q) \quad .$$

Hence, the product $\widehat{\Pi} \sqcap Q$ is empty, and so is the denotation of the local block

The next example illustrates that the initial value of a local variable is arbitrary:

$$[\![\mathbf{var}\ y : \mathbf{Integer}\ \mathbf{rel}\ x' = y\ \mathbf{ler}]\!] = L_x \quad .$$

In the example above, the statement in the local block is defined for any initial value of y. Since the initial value of y is arbitrary, the final value of x is also arbitrary. The next example illustrates that the redeclared variables are preserved:

$$[\![\mathbf{var}\ x, y : \mathbf{Integer}\ \mathbf{rel}\ x' = y\ \mathbf{ler}]\!] = I_x \quad .$$

In the example above, the denotation of the statement in the local block is reduced to the trivial space ϵ, because all the variables of the context are declared locally. The coordinate-wise Cartesian product with the identity on the space of the variable x yields the identity relation, because the trivial space is the identity element of $\otimes$.

Since we require monotonicity for each construct of the language, the following proposition establishes monotonicity for the local block construct.

72 Proposition. *Let p, q be two Utica statements; let w be a variable declaration; let C be the context of statements* **var** *w p and* **var** *w q; then*

$$[\![p]\!]_{C+\Gamma[\![w]\!]} \sqsubseteq [\![q]\!]_{C+\Gamma[\![w]\!]} \Rightarrow [\![\mathbf{var}\ w\ p]\!]_C \sqsubseteq [\![\mathbf{var}\ w\ q]\!]_C \ .$$

Proof. Assume $[\![p]\!]_{C+\Gamma[\![w]\!]} \sqsubseteq [\![q]\!]_{C+\Gamma[\![w]\!]}$.

$$\begin{array}{ll}
 & [\![\mathbf{var}\ w\ p]\!]_C \\
= & \qquad \{ \text{ semantics of } \mathbf{var} \ \} \\
 & ([\![p]\!]_{C+\Gamma[\![w]\!]})\downarrow \\
= & \qquad \{ \text{ definition of } \downarrow \ \} \\
 & (\widehat{\Pi} \circ [\![p]\!]_{C+\Gamma[\![w]\!]} \circ \Pi) \otimes I_{S_{C_1 * C_2}} \\
\sqsubseteq & \qquad \{ \text{ hypothesis, monotonicity of } \circ \text{ and } \otimes \ \} \\
 & (\widehat{\Pi} \circ [\![q]\!]_{C+\Gamma[\![w]\!]} \circ \Pi) \otimes I_{S_{C_1 * C_2}} \\
= & \qquad \{ \text{ definition of } \downarrow \ \} \\
 & ([\![q]\!]_{C+\Gamma[\![w]\!]})\downarrow \\
= & \qquad \{ \text{ semantics of } \mathbf{var} \ \} \\
 & [\![\mathbf{var}\ w\ q]\!]_C \ . \qquad \Box
\end{array}$$

4.3.10 Extended Utica

The statements we have introduced so far define what we refer to as *Pure Utica*. In order to define a complete language capable of accommodating the structuring of specifications as well as the structuring of programs, we need a few additional statements —which we will present in this section. They are not new statements as such; rather they are shorthands for specific patterns of *Pure Utica* statements. Hence, we do not define their semantics by means of mathematical equations, as we have done for Pure Utica statements; rather, we show what Pure Utica patterns they represent.

4.3.10.1 The Assignment

Let $x_1, \ldots, x_n$ be the variables of the context, and let $e(x_1, \ldots, x_n)$ be an expression: A *Utica* statement of the form

$$\begin{array}{l}
\mathbf{rel} \\
\quad x'_i = e(x_1, \ldots, x_n) \wedge \forall j \in 1..n : i \neq j \Rightarrow x'_j = x_j \\
\mathbf{ler}
\end{array}$$

can be written as

$$x_i := e(x_1, \ldots, x_n) \quad .$$

4.3.10.2 The Alternation

A *Utica* statement of the form

```
lub
    $t \to p, \neg t \to q$
bul
```

can be written as

if t **then** p **else** q .

In Section 4.3.11 we will explain why this definition is consistent with the traditional semantics associated with the **if-then-else** statement.

4.3.10.3 The Conditional

Whenever a *Utica* statement of the form

if t **then** p **else** q .

satisfies the condition

$$\overline{[\![t]\!]} \cap [\![q]\!] = \overline{[\![t]\!]} \cap I \quad ,$$

it can be written as follows:

if t **then** p .

In Section 4.3.11 we will explain why this definition is consistent with the traditional semantics associated with the **if-then** statement.

4.3.10.4 The While Statement

Whenever a *Utica* statement of the form

```
lub
    clo if $t$ then $p$ olc,
    establish($\neg t$)
bul
```

satisfies the conditions

1. $[\![t]\!] \subseteq [\![p]\!]L$,
2. $[\![t]\!] \cap [\![p]\!]$ is progressively finite,

it can be written as follows:

while t **do** p .

In Section 4.3.11 we will explain why this definition is consistent with the traditional semantics associated with the **while-do** statement.

We often use the following proposition to prove that the conditions above hold.

73 Proposition. *Let p and q be two consistent Utica statements. If condition t and Utica statement p satisfy*

1. $[\![t]\!] \subseteq [\![p]\!]L$,
2. $[\![t]\!] \cap [\![p]\!]$ *is progressively finite,*

then the following conditions hold:

1. $[\![t]\!] \subseteq [\![\mathbf{lub}\ p,\ q\ \mathbf{bul}]\!]L$,
2. $[\![t]\!] \cap [\![\mathbf{lub}\ p,\ q\ \mathbf{bul}]\!]$ *is progressively finite.*

Proof. For the first condition, we have

$$\begin{array}{ll} & [\![t]\!] \\ \subseteq & \quad \{ \text{ hypothesis } \} \\ & [\![p]\!]L \\ \subseteq & \quad \{ \text{ by hypothesis the join is defined, 32(c) } \} \\ & ([\![p]\!] \sqcup [\![q]\!])L \\ = & \quad \{ \text{ semantics of } \mathbf{lub} \ \} \\ & [\![\mathbf{lub}\ p,\ q\ \mathbf{bul}]\!]L \ . \end{array}$$

For the second condition, we have

$$\begin{array}{ll} & [\![t]\!] \cap ([\![p]\!] \sqcup [\![q]\!]) \\ = & \quad \{ \text{ definition of } \sqcup \ \} \\ & [\![t]\!] \cap ([\![p]\!] \cap \overline{[\![q]\!]L} \cup [\![p]\!] \cap [\![q]\!] \cup \overline{[\![p]\!]L} \cap [\![q]\!]) \\ = & \quad \{ \text{ distributivity } \} \\ & [\![t]\!] \cap ([\![p]\!] \cap \overline{[\![q]\!]L} \cup [\![p]\!] \cap [\![q]\!]) \cup [\![t]\!] \cap \overline{[\![p]\!]L} \cap [\![q]\!]) \\ = & \quad \{ \text{ by hypothesis } [\![t]\!] \subseteq [\![p]\!]L \text{ and } \ 12(\text{i}){:}\ [\![t]\!] \cap \overline{[\![p]\!]L} = \emptyset \ \} \\ & [\![t]\!] \cap ([\![p]\!] \cap \overline{[\![q]\!]L} \cup [\![p]\!] \cap [\![q]\!]) \\ = & \quad \{ \text{ distributivity } \} \\ & [\![t]\!] \cap [\![p]\!] \cap (\overline{[\![q]\!]L} \cup [\![q]\!]) \ . \end{array}$$

By hypothesis, $[\![t]\!] \cap [\![p]\!]$ is progressively finite. By law 22(b) and the result above, we can conclude that

$$[\![t]\!] \cap [\![\mathbf{lub}\ p,\ q\ \mathbf{bul}]\!]$$

is progressively finite. □

4.3.10.5 The Parallel Combination

Whenever we want to execute two *Utica* statements, say p and q, and do not care in what order they are executed, we compose them by the *parallel combination* construct, as follows:

```
par
  p, q
rap .
```

The semantics of this construct can be defined by the following equation:

$$[\![\mathbf{par}\ p,\ q\ \mathbf{rap}]\!] = ([\![p]\!] \circ [\![q]\!]) \sqcap ([\![q]\!] \circ [\![p]\!]) \ .$$

It stems from this definition that the above statement can actually be considered as a shorthand for:

```
glb
   seq $p$, $q$ qes,
   seq $q$, $p$ qes
blg .
```

The intuition behind this statement is the following: the parallel combination of p and q (assuming they are atomic) consists of the information that both **seq** $p,\ q$ **qes** and **seq** $q,\ p$ **qes** carry in common. Each one of them has more information than the parallel combination: each one has not only the information that p and q have been executed, but also that they have been executed in a specific order; the demonic meet of these two factors carries information to the effect that both have been executed, but carries no information about the order of execution.

The *parallel combination* construct can be used whenever one does not have to specify the ordering of execution of the components, and wishes to postpone the decision to a later step in the design process. This construct can be generalized to an arbitrary number of arguments, by considering all the possible permutations of the components. Clearly, this construct is monotonic with respect to its arguments, as it is defined by the composition of monotonic operations.

4.3.11 The Pascal Connection

In this section we establish that the Pascal-like *extended Utica* statements that we have introduced as shorthands for *pure Utica* statements (i.e., **if-then-else**, **if-then**, and **while-do**) have a semantic definition which is consistent with their traditional relation-based definition [4, 50, 57, 65].

Before carrying on with the proofs, we must first observe the following distinction between Pascal and Utica: Pascal is a deterministic language, and Utica allows nondeterminacy. For a proof of equivalence between Pascal statements and Utica statements to make sense, we must either restrict to deterministic Utica statements, or extend the semantics of Pascal constructs to include nondeterministic statements. The latter is more general. In the sequel, we use a semantics of **if-then-else** which allows for nondeterministic **then** part and **else** part. Similarly, we use a semantics of **while** providing for a nondeterministic loop body. Of course, these semantic definitions reduce to the usual definitions of Pascal when deterministic statements are used.

4.3.11.1 The Alternation

If we consider the traditional relation-based definition of the alternation statement [4, 50, 57, 65], we find the following formula for the semantics of this statement:

$$[\![t]\!] \cap [\![p]\!] \cup \overline{[\![t]\!]} \cap [\![q]\!] \ .$$

We must now prove that under the conditions provided in Section 4.3.10.2, the relation defined by our **if-then-else** statement is indeed equal to this formula.

74 Proposition. *Let t be a logical expression and p, q be Utica statements. Then*

$$[\![\textbf{if } t \textbf{ then } p \textbf{ else } q]\!] = [\![t]\!] \cap [\![p]\!] \cup [\![\overline{t}]\!] \cap [\![q]\!] \ .$$

PROOF.

$$\begin{array}{ll}
 & [\![\textbf{if } t \textbf{ then } p \textbf{ else } q]\!] \\
= & \qquad \{ \text{ semantics of } \textbf{if-then-else} \ \} \\
 & [\![\textbf{lub } t \rightarrow p, \neg t \rightarrow q \textbf{ bul}]\!] \\
= & \qquad \{ \text{ semantics of } \rightarrow \text{ and } \textbf{lub} \ \} \\
 & t \cap P \sqcup \overline{t} \cap Q \\
= & \qquad \{ \text{ Corollary 29 } \} \\
 & t \cap P \cup \overline{t} \cap Q \ . \qquad \square
\end{array}$$

Hence what we have chosen to call **if-then-else** represents Pascal's **if-then-else**. By virtue of Corollary 29, a statement **if** t **then** p **else** q is defined for any t, p, q, and so is any refinement of an **if-then-else** statement.

4.3.11.2 The Conditional

If we consider the traditional relation-based definition of the conditional statement [4, 50, 57, 65], we find the following formula for the semantics of this statement:

$$[\![t]\!] \cap [\![p]\!] \cup \overline{[\![t]\!]} \cap I \ .$$

We must now prove that the relation defined by our **if-then** statement is indeed equal to this formula.

75 Proposition. *Let t be a logical expression and p be a Utica statement. Then*

$$[\![\textbf{if } t \textbf{ then } p]\!] = [\![t]\!] \cap [\![p]\!] \cup \overline{[\![t]\!]} \cap I \ .$$

PROOF.

$$\begin{array}{ll}
 & [\![\textbf{if } t \textbf{ then } p]\!] \\
= & \qquad \{ \text{ semantics of } \textbf{if-then}, \text{ and } \overline{t} \cap I = \overline{t} \cap [\![\textbf{skip}]\!] \ \} \\
 & [\![\textbf{if } t \textbf{ then } p \text{ else } \textbf{skip}]\!] \\
= & \qquad \{ \text{ by proposition above } \} \\
 & t \cap P \cup \overline{t} \cap I \ . \qquad \square
\end{array}$$

Hence what we have chosen to call **if-then** as a shorthand for a *Pure Utica* statement indeed represents Pascal's **if-then**.

4.3.11.3 The While Statement

The semantics of a statement **while** t **do** p is traditionally defined as the least fixpoint under the inclusion ordering ($\subseteq$) of the following function [50, 57, 65]:

$$g.X \triangleq \overline{t} \cap I \cup t \cap PX \ .$$

In a demonic semantics context [4], this function becomes

76 $\quad f.X \triangleq \overline{t} \cap I \sqcup t \cap P \,\square\, X \ ,$

or, equivalently,

77 $f.X \triangleq \overline{t} \cap I \cup t \cap P \,\Box\, X$,

and we select the least fixpoint under $\sqsubseteq$. To see that a least fixpoint exists for all t and p, the reader may observe that f is monotonic with respect to $\preceq$ (laws 60(b,c) and $\sqsubseteq$ (34(c), 56(a)). Theorem 3 shows the existence of a least fixpoint under $\preceq$, since relations under $\preceq$ forms a CPO (Proposition 59). This least fixpoint is also a solution to the inequation $f.X \sqsubseteq X$. By Theorem 2, there exists a least fixpoint for f under $\sqsubseteq$ given by the following formula[§]

$$\bigsqcap \{X | f.X \sqsubseteq X\} \ .$$

When defined as the least fixpoint of f under $\sqsubseteq$, the **while** construct is also monotonic in P with respect to $\sqsubseteq$. Let W_P and W_Q be the least fixpoints of

$$f_P.X \triangleq \overline{t} \cap I \cup t \cap P \,\Box\, X$$

and

$$f_Q.X \triangleq \overline{t} \cap I \cup t \cap Q \,\Box\, X$$

respectively. Then

$$\begin{array}{ll}
 & P \sqsubseteq Q \\
\Rightarrow & \qquad \{ \text{ monotonicity of operations } \} \\
 & \overline{t} \cap I \cup t \cap P \,\Box\, W_Q \ \sqsubseteq\ \overline{t} \cap I \cup t \cap Q \,\Box\, W_Q \\
\Leftrightarrow & \qquad \{ \text{ definition of } f_P \text{ and } f_Q \ \} \\
 & f_P.W_Q \ \sqsubseteq\ f_Q.W_Q \\
\Leftrightarrow & \qquad \{\ W_Q \text{ is a fixpoint of } f_Q \ \} \\
 & f_P.W_Q \ \sqsubseteq\ W_Q \\
\Leftrightarrow & \qquad \{ \text{ definition of } \in \ \} \\
 & W_Q \in \{X | f_P.X \sqsubseteq X\} \\
\Rightarrow & \qquad \{ \text{ property of } \bigwedge \ \} \\
 & \bigsqcap \{X | f_P.X \sqsubseteq X\} \sqsubseteq W_Q \\
\Leftrightarrow & \qquad \{ \text{ definition of } W_P \text{ and Theorem } \ 2 \ \} \\
 & W_P \sqsubseteq W_Q \ .
\end{array}$$

These preliminary remarks being made, we must now establish that under the conditions

1. $[\![t]\!] \subseteq [\![p]\!]L$,
2. $[\![t]\!] \cap [\![p]\!]$ is progressively finite,

the semantics of

```
lub
   clo if $t$ then $p$ olc,
   establish($\neg t$)
bul
```

[§]The reader may now see why we introduced the prerestriction ordering: it simplifies the proof of existence of a fixpoint for a function. We could also include *recursion* in our language, and define its semantics using least fixpoints, in a similar fashion as for the **while** construct. With the prerestriction ordering, it is easy to show that the function associated to a recursion built using **if**, **glb** and $\Box$ has a fixpoint.

is equal to the demonic semantics of the while statement, which is the least fixpoint under $\sqsubseteq$ of f (Equation 77). We conduct our proofs by showing that each expression is equal to

78 $\quad (t \cap P)^* \cap \widehat{\overline{t}},$

under the conditions stated above.

First, we prove that the least fixpoint of f is equal to (78). The following two lemmas are required. The first lemma is used to establish the second lemma. The second lemma provides a lower bound for the domain of the relation in (78).

79 Lemma. *If $t \cap P$ is progressively finite and $\widehat{P}L \subseteq \overline{t} \cup PL$ then $t \cap PL \subseteq (t \cap P)^+\overline{t}$*

Proof.

$t \cap P$ is progressively finite

$\Rightarrow \qquad \{$ 22(a) $\}$

$(t \cap P)^*\overline{(t \cap P)L} = L$

$\Leftrightarrow \qquad \{$ definition of $*$ $\}$

$\overline{t \cap PL} \cup (t \cap P)^+\overline{t \cap PL} = L$

$\Leftrightarrow \qquad \{$ by induction: $((t \cap P)^j)\widehat{\ }\,L \subseteq \widehat{P}L \subseteq \overline{t} \cup PL$ using law 15(v); law 15(u) $\}$

$\overline{t \cap PL} \cup (t \cap P)^+(\overline{t \cap PL} \cap (\overline{t} \cup PL)) = L$

$\Leftrightarrow \qquad \{$ distributivity $\}$

$\overline{t \cap PL} \cup (t \cap P)^+(\overline{t} \cap \overline{t} \cup \overline{t} \cap PL \cup \overline{t} \cap \overline{PL} \cup \overline{PL} \cap PL) = L$

$\Leftrightarrow \qquad \{$ definition of $\overline{}$, Boolean laws (12) $\}$

$\overline{t \cap PL} \cup (t \cap P)^+\overline{t} = L$

$\Leftrightarrow \qquad \{$ 12(h) $\}$

$t \cap PL \subseteq (t \cap P)^+\overline{t}\ . \qquad \Box$

80 Lemma. *If $t \cap P$ is progressively finite and $\widehat{P}L \subseteq \overline{t} \cup PL$ then $\overline{t} \cup PL \subseteq ((t \cap P)^* \cap \widehat{\overline{t}})L$*

Proof.

$\overline{t} \cup PL$

$= \qquad \{$ $t \cup \overline{t} = L$, definition of L and distributivity $\}$

$\overline{t} \cup t \cap PL$

$\subseteq \qquad \{$ Lemma 79 and $\widehat{P}L \subseteq \overline{t} \cup PL$ $\}$

$\overline{t} \cup (t \cap P)^+\overline{t}$

$= \qquad \{$ definition of $*$ $\}$

$(t \cap P)^*\overline{t}$

$= \qquad \{$ definition of L $\}$

$(t \cap P)^*(\overline{t} \cap L)$

$= \qquad \{$ 15(r) $\}$

$((t \cap P)^* \cap \widehat{\overline{t}})L\ . \qquad \Box$

The next lemma is taken from [4]. It provides conditions on the domain of the fixpoints of f.

81 Lemma. *Let $f.X \triangleq \overline{t} \cap I \cup t \cap P \mathbin{\Box} X$, where P is such that $t \cap P$ is progressively finite and $\widehat{P}L \subseteq \overline{t} \cup PL$. Every fixpoint Y of f satisfies $YL = \overline{t} \cup PL$.* $\qquad \Box$

The next proposition provides that the least fixpoint of f, under conditions slightly weaker than the conditions in Section 4.3.10.4, is indeed expression (78).

82 Proposition. *Let* $f.X \triangleq \overline{t} \cap I \cup t \cap P \,\Box\, X$, *where* P *is such that* $t \cap P$ *is progressively finite and* $\widehat{P}L \subseteq \overline{t} \cup PL$. *Then* f *has a unique fixpoint given by*

$$(t \cap P)^* \cap \widehat{\overline{t}} \ .$$

Proof. Function f is monotonic with respect to $\preceq$ (60(b), (c)). From Proposition 59 and Theorem 3, it stems that f has a least fixpoint. Let Y be this least fixpoint and let Z be another fixpoint of f. We show that Y is unique (i.e., $Y = Z$).

$$\begin{array}{ll}
 & Y \preceq Z \\
\Leftrightarrow & \qquad \{ \text{ definition of } \preceq (57) \ \} \\
 & Y = YL \cap Z \\
\Leftrightarrow & \qquad \{ \ YL = ZL \text{ by Lemma 81 } \} \\
 & Y = ZL \cap Z \\
\Leftrightarrow & \qquad \{ \ Z \subseteq ZL \ \} \\
 & Y = Z \ .
\end{array}$$

All that is left to show is that $(t \cap P)^* \cap \widehat{\overline{t}}$ is a fixpoint of f

$$\begin{array}{ll}
 & f.((t \cap P)^* \cap \widehat{\overline{t}}) \\
= & \qquad \{ \text{ definition of } f \ \} \\
 & \overline{t} \cap I \cup t \cap P \,\Box\, ((t \cap P)^* \cap \widehat{\overline{t}}) \\
= & \qquad \{ \ 34(\text{j}), \text{ hypothesis } \widehat{P}L \subseteq \overline{t} \cup PL \text{ and Lemma 80 } \} \\
 & \overline{t} \cap I \cup t \cap P((t \cap P)^* \cap \widehat{\overline{t}}) \\
= & \qquad \{ \ 15(\text{s}), (\text{c}), 16(\text{b}) \ \} \\
 & \overline{t} \cap I \cup (t \cap P)^+ \cap \widehat{\overline{t}} \\
= & \qquad \{ \text{ by } 15(\text{r}) \ \overline{t} \cap I = \widehat{\overline{t}} \cap I, \text{ distributivity } 12(\text{r}) \ \} \\
 & (I \cup (t \cap P)^+) \cap \widehat{\overline{t}} \\
= & \qquad \{ \text{ definition of } {}^* \ \} \\
 & (t \cap P)^* \cap \widehat{\overline{t}} \ . \qquad \Box
\end{array}$$

We now have to prove that the semantics of the *Utica* statement

lub
  clo if t **then** p **olc**,
  establish$(\neg t)$
bul

is equal to the least fixpoint of f under the conditions of Section 4.3.10.4. The two components of the **lub** construct above define the following relations:

$$\begin{array}{l}
[\![\textbf{clo if } t \textbf{ then } p \textbf{ olc}]\!] = \sqcap_{\, i \geq 0} (t \cap P \cup \overline{t} \cap I)^{\boxed{i}} \\
[\![\textbf{establish}(\neg t)]\!] = \widehat{\overline{t}} \quad ,
\end{array}$$

so that the **lub** is interpreted by the following expression, if it is defined:

$$(\sqcap_{\, i \geq 0} (t \cap P \cup \overline{t} \cap I)^{\boxed{i}} \,) \sqcup \widehat{\overline{t}} \ .$$

Before we can prove that this expression is indeed equal to expression (78), we must establish two simple lemmas.

83 Lemma. *If left vector t and relation P satisfy condition $t \subseteq PL$ then*

$$\textstyle\bigsqcap_{\,i\geq 0}(t \cap P \cup \overline{t} \cap I)^{\boxed{i}} = (t \cap P \cup \overline{t} \cap I)^* \ .$$

PROOF.

$$\begin{array}{ll}
 & (t \cap P \cup \overline{t} \cap I)L \\
= & \qquad \{ \text{ distributivity of } \cup \ \} \\
 & (t \cap P)L \cup (\overline{t} \cap I)L \\
= & \qquad \{ \text{ relational identities } (t \text{ and } \overline{t} \text{ are left vectors}) \ \} \\
 & tL \cap PL \cup \overline{t}L \cap IL \\
= & \qquad \{ \text{ relational identities } (t \text{ and } \overline{t} \text{ are left vectors, and } IL = L) \ \} \\
 & t \cap PL \cup \overline{t} \cap L \\
= & \qquad \{ \text{ by hypothesis, and because } L \text{ is maximal } \} \\
 & t \cup \overline{t} \\
= & \qquad \{ \text{ by definition of } ^{-} \ \} \\
 & L \ .
\end{array}$$

Using law 34(j) and the fact that $(t \cap P \cup \overline{t} \cap I)$ is total, we find that

$$(t \cap P \cup \overline{t} \cap I)^{\boxed{i}} = (t \cap P \cup \overline{t} \cap I)^i \ .$$

Using law 15(v) and the fact above, we find

$$(t \cap P \cup \overline{t} \cap I)^i L = L \ .$$

Hence, using the definition of $\sqcap$, we find

$$\textstyle\bigsqcap_{\,i\geq 0}(t \cap P \cup \overline{t} \cap I)^{\boxed{i}} = \displaystyle\bigcup_{i\geq 0}(t \cap P \cup \overline{t} \cap I)^i \ .$$

□

84 Lemma. *If left vector t and relation P satisfy condition $t \subseteq PL$ and are such that $t \cap P$ is progressively finite, then relations $\bigsqcap_{\,i\geq 0}(t \cap p \cup \overline{t} \cap I)^{\boxed{i}}$ and $\widehat{\overline{t}}$ satisfy the consistency condition.*

PROOF. By virtue of Lemma 83, we must show that

$$(t \cap P \cup \overline{t} \cap I)^* L \cap \widehat{\overline{t}}L \subseteq ((t \cap P \cup \overline{t} \cap I)^* \cap \widehat{\overline{t}})L \ .$$

To do so, we content ourselves with establishing that the right hand side is actually L.

$$\begin{array}{ll}
 & ((t \cap P \cup \overline{t} \cap I)^* \cap \widehat{\overline{t}})L \\
= & \qquad \{ \text{ 15(r), definition of } L \ \} \\
 & (t \cap P \cup \overline{t} \cap I)^* \overline{t} \\
\supseteq & \qquad \{ \text{ monotonicity of transitive closure with respect to inclusion } \} \\
 & (t \cap P)^* \overline{t} \\
= & \qquad \{ \text{ because } t \subseteq PL \text{ and } (t \cap P)L = t \cap PL \ \} \\
 & (t \cap P)^* \overline{(t \cap P)L} \\
= & \qquad \{ \text{ hypothesis } t \cap P \text{ is progressively finite and } \ 22(a) \ \} \\
 & L \ . \qquad \square
\end{array}$$

Finally, we have the following proposition.

85 Proposition. *If the logical expression t and the Utica statement p satisfy condition $[\![t]\!] \subseteq [\![p]\!]L$ and are such that $[\![t]\!] \cap [\![p]\!]$ is progressively finite, the semantics of*

```
lub
   clo if $t$ then $p$ olc,
   establish($\neg t$)
bul
```

equals $([\![t]\!] \cap [\![p]\!])^* \cap \widehat{\overline{[\![t]\!]}}$.

Proof.

$$\begin{array}{ll}
 & [\![\textbf{lub clo if } t \textbf{ then } p \textbf{ olc}, \textbf{establish}(\neg t) \textbf{ bul}]\!] \\
= & \qquad\qquad \{ \text{ Lemma 84 } \} \\
 & (\bigsqcap_{i \geq 0} (t \cap P \cup \overline{t} \cap I)^{\boxed{i}}) \sqcup \widehat{\overline{t}} \\
= & \qquad\qquad \{ \text{ Lemma 83, Corollary 30 } \} \\
 & (t \cap P \cup \overline{t} \cap I)^* \cap \widehat{\overline{t}} \\
= & \qquad\qquad \{ \text{ 16(h) } \} \\
 & (t \cap P)^* \cap \widehat{\overline{t}} \ . \qquad \square
\end{array}$$

It is also possible to define another equivalence between a **while** statement and a Utica statement, which is more general, but less intuitive and more complex to prove. We state it without proof.

86 Proposition. *Whenever a* Utica *statement of the form*

```
lub
   clo if $t$ then $p$ olc,
   $u \rightarrow$ establish($\neg t$)
bul
```

satisfies the conditions

1. $\widehat{[\![p]\!]}L \subseteq \overline{[\![t]\!]} \cup [\![p]\!]L = [\![u]\!]$*;*

2. $[\![t]\!] \cap [\![p]\!]$ *is progressively finite,*

it can be written as follows:

```
while $t$ do $p$   .        □
```

Condition u represents the domain of termination of the **while** statement. This completes the definition of Utica.

Chapter 5

Transformation Rules

Utica is designed to support the program construction process, from the specification to the program; in particular, it has the necessary constructs to support the structuring of specifications (based primarily, although not exclusively, on **lub**'s) and the structuring of programs (based on sequence, alternation, conditional, and iteration). The program construction process consists primarily of getting away from **lub**'s and towards the program structuring constructs; hence it revolves around the goal of getting rid of **lub**'s. It is mandatory to remove **lub**'s, because it is not known yet how to generate code of reasonable efficiency to implement the **lub** construct.

The definitions that we have presented in Section 4.3.10, whereby a **lub**-based pattern is transformed into a Pascal-like pattern, are one way to eliminate **lub**'s. However, these transformations do not always apply. If an **lub** cannot be removed, then another way is to propagate the **lub** deeper and deeper into the nesting structure of the program, in order to facilitate its removal. Finally, if an **lub** cannot be eliminated nor propagated, then each component of the **lub** is refined until elimination or propagation is applicable.

The purpose of this chapter is to introduce a number of rules leading to the removal of **lub**'s. Each rule provides a refinement between Utica patterns. When presenting rules, we will give a proof of the refinement, and a mixture of comments, dealing with why the rules are useful, what they mean, etc. Rules are usually expressed with binary constructs (e.g., **lub** p, q **bul** is refined by ...) for simplicity and clarity. Nevertheless, rules can be easily generalized for n-ary constructs by induction using associativity.

5.1 Eliminating Lub

In addition to the extended Utica constructs (Section 4.3.10), which allow the transformation of an **lub** statement into an equivalent Pascal-like statement, we have the following rules to remove **lub**'s from specifications.

Refinement Rule 1 *The statement* **lub** p, p **bul** *is equivalent to* p.

Proof. This stems from the idempotence of joins. □

Refinement Rule 2 *If* $t \wedge u$ *is not equivalent to* **false**, *the statement*

```
    lub
        establish(t),
```

        establish(u)
    bul

is equivalent to the statement **establish**$(t \wedge u)$.

PROOF. Given that $[\![\textbf{establish}(t)]\!]$ and $[\![\textbf{establish}(u)]\!]$ are both right vectors ($\widehat{t}$ and $\widehat{u}$, respectively) their demonic join, when it exists, is their intersection. Note that when $t \wedge u \Leftrightarrow$ **false**, the **lub** in the consequent is undefined. $\square$

Refinement Rule 3 *If statements p and q satisfy the condition*

$$[\![p]\!] \cap [\![q]\!]L \subseteq [\![q]\!] \ ,$$

the statement

lub p, q **bul**

is equivalent to

if t **then** p **else** q

where t is a logical expression satisfying $[\![t]\!] = [\![p]\!]L$.

PROOF. First, we prove the consistency of the two **lub** components:

$$PL \cap QL = (P \cap QL)L = (P \cap QL \cap Q)L = (P \cap Q)L \ .$$

Next, we prove the equivalence

$$\begin{array}{ll}
 & [\![\textbf{lub}\ p,\ q\ \textbf{bul}]\!] \\
= & \qquad \{ \text{ semantics of } \textbf{lub} \ \} \\
 & P \sqcup Q \\
= & \qquad \{ \text{ definition of } \sqcup \ \} \\
 & P \cap \overline{QL} \cup P \cap Q \cup \overline{PL} \cap Q \\
= & \qquad \{ \ Q = Q \cap QL \ \} \\
 & P \cap \overline{QL} \cup P \cap Q \cap QL \cup \overline{PL} \cap Q \\
= & \qquad \{ \text{ hypothesis } P \cap QL \subseteq Q \ \} \\
 & P \cap \overline{QL} \cup P \cap QL \cup \overline{PL} \cap Q \\
= & \qquad \{ \text{ distributivity and definition of } \overline{} \ \} \\
 & P \cup \overline{PL} \cap Q \\
= & \qquad \{ \ P = P \cap PL \ \} \\
 & PL \cap P \cup \overline{PL} \cap Q \\
= & \qquad \{ \ \text{ hypothesis } t = PL, \text{ Corollary 29 and definition of } \textbf{if} \ \} \\
 & [\![\textbf{if}\ t\ \textbf{then}\ p\ \textbf{else}\ q]\!] \ . \qquad \square
\end{array}$$

5.2 Propagating the Lub Deeper

The transformation rules given in this section allow us to push the **lub** further and further inside the nesting structure of the program, until the arguments of the **lub** are sufficiently simple that a trivial refinement can be computed for them; the rules given in Section 5.1 will give us means to get rid of the **lub** altogether, once its arguments are simple enough. Every rule in section 5.2 follows from the following proposition.

87 Proposition. *Let $\Phi(P_1, \ldots, P_n)$ be a monotonic operator with respect to $\sqsubseteq$. Then the following holds if the joins are defined:*

$$\Phi(P_1, \ldots, P_n) \sqcup \Phi(Q_1, \ldots, Q_n) \sqsubseteq \Phi(P_1 \sqcup Q_1, \ldots, P_n \sqcup Q_n) \ .$$

PROOF.

$$\begin{array}{ll} & \Phi(P_1, \ldots, P_n) \sqcup \Phi(Q_1, \ldots, Q_n) \\ \sqsubseteq & \qquad \{ \ P_i \sqsubseteq P_i \sqcup Q_i, \text{ monotonicity of } \Phi \text{ and } \sqcup \ \} \\ & \Phi(P_1 \sqcup Q_1, \ldots, P_n \sqcup Q_n) \sqcup \Phi(Q_1, \ldots, Q_n) \\ \sqsubseteq & \qquad \{ \ Q_i \sqsubseteq P_i \sqcup Q_i, \text{ monotonicity of } \Phi \text{ and } \sqcup \ \} \\ & \Phi(P_1 \sqcup Q_1, \ldots, P_n \sqcup Q_n) \sqcup \Phi(P_1 \sqcup Q_1, \ldots, P_n \sqcup Q_n) \\ = & \qquad \{ \text{ idempotence of } \sqcup \ \} \\ & \Phi(P_1 \sqcup Q_1, \ldots, P_n \sqcup Q_n) \ . \qquad \square \end{array}$$

Since every Utica construct is monotonic with respect to refinement, we can derive one rule for each construct using Proposition 87.

Refinement Rule 4 *If the* **lub***'s are defined, the statement*

```
lub
   seq $p_1$, $p_2$ qes,
   seq $q_1$, $q_2$ qes
bul
```

is refined by

```
seq
   lub $p_1$, $q_1$ bul,
   lub $p_2$, $q_2$ bul
qes .
```

$\square$

Refinement Rule 5 *If the* **lub***'s are defined, the statement*

```
lub
   clo $p_1$ olc,
   clo $p_2$ olc
bul
```

is refined by

```
clo
   lub $p_1$, $p_2$ bul
olc .
```

$\square$

Refinement Rule 6 *If the* **lub***'s are defined, the statement*

```
lub
  glb $p_1$, $p_2$ blg,
  glb $q_1$, $q_2$ blg
bul
```

is refined by

```
glb
  lub $p_1$, $q_1$ bul,
  lub $p_2$, $q_2$ bul
blg .
```

□

Note that because the demonic meet is commutative (hence so is the **glb** construct), other refinements are possible as well (e.g., combining p_1 with q_2 and p_2 with q_1).

Refinement Rule 7 *If the* **lub***'s are defined, the statement*

```
lub
  par $p_1$, $p_2$ rap,
  par $q_1$, $q_2$ rap
bul
```

is refined by

```
par
  lub $p_1$, $q_1$ bul,
  lub $p_2$, $q_2$ bul
rap .
```

□

Refinement Rule 8 *If the* **lub***'s are defined, the statement*

```
lub
  if $t$ then $p_1$ else $p_2$,
  if $t$ then $q_1$ else $q_2$
bul
```

is refined by

```
if $t$ then
  lub $p_1$, $q_1$ bul,
else
  lub $p_2$, $q_2$ bul .
```

□

Refinement Rule 9 *If the* **lub***'s are defined, the statement*

```
lub
  while $t$ do $p$,
  while $t$ do $q$
bul
```

is refined by

```
while $t$ do
   lub $p$, $q$ bul .
```

□

A specifier may apply the rules of this section without checking the consistency condition *a priori* and *a posteriori*. If, after a number of applications of these rules, the specifier ends up with an undefined **lub**, the inconsistency between the components will become evident, because the specifier will not be able to apply the rules of Section 5.1 to remove the undefined **lub**. Indeed, all the rules of Section 5.1 require consistency between the components. On the other hand, if the specifier ends up with defined **lub**'s after applying rules of Section 5.2, he is assured that all the **lub**'s of the intermediate transformations are also defined, as the following proposition provides.

88 Proposition. *Let P_1, P_2, Q_1, Q_2 be relations; let Φ be a monotonic operator wrt $\sqsubseteq$ on relations such that $P_1 \Phi P_2$ and $Q_1 \Phi Q_2$ are defined. Then, we have:*

$$cs(P_1, Q_1) \wedge cs(P_2, Q_2) \Rightarrow cs(P_1 \Phi P_2, Q_1 \Phi Q_2) \ .$$

Proof.

$$\begin{array}{ll} & cs(P_1, Q_1) \wedge cs(P_2, Q_2) \\ \Rightarrow & \qquad \{ \ cs(P_i, Q_i) \text{ implies } P_i \sqcup Q_i \text{ exists } \} \\ & cs((P_1 \sqcup Q_1)\Phi(P_2 \sqcup Q_2), (P_1 \sqcup Q_1)\Phi(P_2 \sqcup Q_2)) \\ \Rightarrow & \qquad \{ \ P_1 \sqsubseteq P_1 \sqcup Q_1, \text{ monotonicity of } \Phi, \text{ Proposition 66 } \} \\ & cs(P_1\Phi(P_2 \sqcup Q_2), (P_1 \sqcup Q_1)\Phi(P_2 \sqcup Q_2)) \\ \Rightarrow & \qquad \{ \ P_2 \sqsubseteq P_2 \sqcup Q_2, \text{ monotonicity of } \Phi, \text{ Proposition 66 } \} \\ & cs(P_1\Phi P_2, (P_1 \sqcup Q_1)\Phi(P_2 \sqcup Q_2)) \\ \Rightarrow & \qquad \{ \ Q_1 \sqsubseteq P_1 \sqcup Q_1, \text{ monotonicity of } \Phi, \text{ Proposition 66 } \} \\ & cs(P_1\Phi P_2, Q_1\Phi(P_2 \sqcup Q_2)) \\ \Rightarrow & \qquad \{ \ Q_2 \sqsubseteq P_2 \sqcup Q_2, \text{ monotonicity of } \Phi, \text{ Proposition 66 } \} \\ & cs(P_1\Phi P_2, Q_1\Phi Q_2) \ . \qquad \square \end{array}$$

It is seldom required to prove consistency from Proposition (28): if one is successful in removing an **lub** using rules of Section 5.1 or Section 4.3.10, then consistency is proved as a byproduct. One would need to prove consistency using the definition only if it seems impossible to eliminate the join, as a means of determining if components were refined up to a point where they can no longer agree.

5.3 Refining Utica Constructs

The rules of this section help the designer in refining the components of an **lub** until they satisfy the conditions for eliminating or propagating the **lub**.

5.3.1 Refining Closure

Refinement Rule 10 *The statement* **clo** p **olc** *is refined by* **skip***.*

Proof. The closure is defined as a demonic meet, hence is refined by any one of the arguments of the meet: one of them is $P^{\boxed{0}}$, which is I. $\square$

Refinement Rule 11 *The statement* **clo** p **olc** *is equivalent to the statement*

```
seq
   clo $p$ olc, clo $p$ olc
qes  .
```

Proof. This rule is an application of law 37(c). $\square$

Refinement Rule 12 *The statement* **clo** p **olc** *is equivalent to the statement*

```
clo
   clo $p$ olc
olc  .
```

Proof. This rule is an application of law 37(d). $\square$

Refinement Rule 13 *The statement*

```
clo $p$ olc
```

is refined by the statement

```
clo if $t$ then $p$ olc
```

for an arbitrary logical expression t.

Proof. This rule is an application of law 37(e). $\square$

5.3.2 Refining Establish

Refinement Rule 14 *Whenever the logical expression t implies the logical expression u, and t is not equivalent to* **false***, the statement* **establish**(t) *refines the statement* **establish**(u)*.*

Proof. Assume $\neg(t \Leftrightarrow \textbf{false})$.

$$
\begin{array}{ll}
 & t \Rightarrow u \\
\Leftrightarrow & \\
 & \{(s,s')|t(s')\} \subseteq \{(s,s')|u(s')\} \\
\Leftrightarrow & \qquad \{ \sqsupseteq \text{ and } \subseteq \text{ coincide on non-empty right vectors } \} \\
 & \{(s,s')|t(s')\} \sqsupseteq \{(s,s')|u(s')\} \\
\Leftrightarrow & \qquad \{ \text{ definition of } \textbf{establish} \ \} \\
 & [\![\textbf{establish}(t)]\!] \sqsupseteq [\![\textbf{establish}(u)]\!] \ . \qquad \square
\end{array}
$$

Refinement Rule 15 *The statement* **establish**(t) *is equivalent to the statement*

seq
  establish(t), **preserve**(t)
qes .

PROOF. Assume $\neg(t \Leftrightarrow \textbf{false})$.

$$\begin{aligned}
&[\![\textbf{seq establish}(t),\ \textbf{preserve}(t)\ \textbf{qes}]\!]\\
=&\qquad \{\text{ definition of } \textbf{establish} \text{ and } \textbf{preserve} \}\\
&\widehat{t} \mathbin{\square} (t \cap \widehat{t})\\
=&\qquad \{\ 34(\text{m})\ \}\\
&\widehat{t} \mathbin{\square} t \cap \widehat{t} \mathbin{\square} \widehat{t}\\
=&\qquad \{\ 34(\text{j})\ \}\\
&\widehat{t}t \cap \widehat{t}\widehat{t}\\
=&\qquad \{\text{ definitions of vector, hypothesis and Tarski rule, } 15(\text{j}),(\text{n})\ \}\\
&L \cap \widehat{t}\\
=&\qquad \{\text{ definition } 7(\text{b})\ \}\\
&\widehat{t}\\
=&\qquad \{\text{ definition of } \textbf{establish} \}\\
&[\![\textbf{establish}(t)]\!]\ .
\end{aligned}$$

On the other hand, if $t \Leftrightarrow \textbf{false}$ then

$$\begin{aligned}
&[\![\textbf{establish}(\text{t})]\!]\\
=&\qquad \{\text{ semantics of } \textbf{establish} \}\\
&\emptyset\\
=&\qquad \{\text{ semantics of } \textbf{preserve} \text{ and } \textbf{seq} \}\\
&[\![\textbf{seq preserve}(t),\ \textbf{establish}(t)\ \textbf{qes}]\!]\ . \qquad \square
\end{aligned}$$

One way to establish a condition is to establish it early in the program, and spend the rest of the program preserving it.

Refinement Rule 16 *The statement* **establish**(t) *is equivalent to the statement*

seq
  establish(**true**), **establish**(t)
qes .

PROOF.

$$\begin{aligned}
&[\![\textbf{seq establish}(\textbf{true}),\ \textbf{establish}(t)\ \textbf{qes}]\!]\\
=&\qquad \{\text{ definition of } \textbf{establish} \}\\
&L \mathbin{\square} \widehat{t}\\
=&\qquad \{\ 34(\text{j})\ \}\\
&L\widehat{t}\\
=&\qquad \{\text{ definition of left vector, } 15(\text{j})\ \}\\
&\widehat{t}\\
=&\qquad \{\text{ definition of } \textbf{establish} \}\\
&[\![\textbf{establish}(t)]\!]\ . \qquad \square
\end{aligned}$$

Establishing a condition can be achieved by first performing an arbitrary state transformation (**establish**(**true**)), and then establishing the condition. Such a rule is most useful when we need to establish a simple condition in two steps. This situation arises when we must match a sequence in other components of a join statement in order to apply Rule 4.

5.3.3 Refining Preserve

Refinement Rule 17 *The statement* **preserve**(t) *is equivalent to*

> **seq**
> **preserve**(t), **preserve**(t)
> **qes** .

PROOF.

$$\begin{array}{ll}
 & [\![\textbf{seq preserve}(t), \textbf{preserve}(t) \textbf{ qes}]\!] \\
= & \qquad \{ \text{ definition of } \textbf{preserve} \} \\
 & (t \cap \widehat{t}) \,\square\, (t \cap \widehat{t}) \\
= & \qquad \{ \text{ 34(j) } \} \\
 & (t \cap \widehat{t})(t \cap \widehat{t}) \\
= & \qquad \{ \text{ 15(r) } \} \\
 & t(t \cap t \cap \widehat{t}) \\
= & \qquad \{ \text{ idempotence of } \cap \text{ (12(a)), 15(s) } \} \\
 & tt \cap \widehat{t} \\
= & \qquad \{ \text{ 15(n) } \} \\
 & t \cap \widehat{t} \\
= & \qquad \{ \text{ definition of } \textbf{preserve} \} \\
 & [\![\textbf{preserve}(t)]\!] \ . \qquad \square
\end{array}$$

To preserve a condition, one may preserve it twice. This rule is useful to match a sequence in other components of a join.

Refinement Rule 18 *The statement* **preserve**(t) *is equivalent to*

> **clo**
> **preserve**(t)
> **olc** .

PROOF. The construct **preserve**(t) is demonically reflexive

$$\begin{array}{ll}
 & [\![\textbf{preserve}(t)]\!] \sqsubseteq I \\
\Leftrightarrow & \qquad \{ \text{ definition of } \textbf{preserve} \} \\
 & t \cap \widehat{t} \ \sqsubseteq \ I \\
\Leftrightarrow & \qquad \{ \text{ definition of } \sqsubseteq \text{ (24) for total relations } \} \\
 & I \cap (t \cap \widehat{t})L \ \subseteq \ t \cap \widehat{t} \\
\Leftrightarrow & \qquad \{ \text{ 15(r), (n) } \} \\
 & I \cap t \ \subseteq \ t \cap \widehat{t} \\
\Leftrightarrow & \qquad \{ \text{ idempotence of } \cap \text{ (12(a)) } \} \\
 & I \cap t \cap t \ \subseteq \ t \cap \widehat{t} \\
\Leftrightarrow & \qquad \{ I \cap t = I \cap \widehat{t} \text{ by 15(r) } \} \\
 & I \cap t \cap \widehat{t} \ \subseteq \ t \cap \widehat{t} \\
\Leftrightarrow & \qquad \{ \text{ 12(e) } \} \\
 & \textbf{true} \ .
\end{array}$$

By virtue of Rule 17, the construct **preserve**(t) is demonically transitive. Hence, by law 37(b), the equivalence follows. $\square$

To preserve a condition, one may do nothing, or keep preserving the condition an indefinite number of times. This is exactly what a **clo** construct does. This rule is useful to match another **clo** statement in a join, in order to apply Rule 5.

5.3.4 Refining Glb

Refinement Rule 19 *The statement* **glb** p_1, p_2 **blg** *is refined by* p_1. □

PROOF. A meet is smaller than its arguments. □

5.3.5 Refining Par

Refinement Rule 20 *The statement*

> **par** p_1, p_2 **rap**

is refined by the statement

> **seq** p_1, p_2 **qes** .

PROOF. A meet is smaller than its arguments. □

5.3.6 Refining Sequence

Refinement Rule 21 *The statement*

```
seq
    if $t$ then $p$ else $q$,
    $r$
qes
```

is equivalent to

```
if $t$ then seq $p$, $r$ qes
    else seq $q$, $r$ qes .
```

PROOF. The rule follows directly from law 34(o). □

5.3.7 Refining Prerestriction

Refinement Rule 22 *The statement* $t \rightarrow p$ *is refined by* p.

PROOF. The rule follows directly from law 56(b). □

Refinement Rule 23 *The statement* $t \rightarrow (u \rightarrow p)$ *is equivalent to* $u \rightarrow (t \rightarrow p)$.

PROOF. The rule follows directly from the asociativity of $\cap$. □

Refinement Rule 24 *The statement*

> $t \rightarrow$ **seq** p, q **qes**

is equivalent to

seq $t \rightarrow p$, q **qes** .

Proof. The rule follows directly from law 34(l). □

To prerestrict a sequence is the same as to prerestrict the first item of the sequence.

Refinement Rule 25 *The statement*

$t \rightarrow$ **lub** p, q **bul**

is equivalent to

lub $t \rightarrow p$, $t \rightarrow q$ **bul** .

Proof. The rule follows directly from law 32(f). □

5.3.8 Refining Alternation

Refinement Rule 26 *The statement*

if t **then** p **else** q

is equivalent to

if t **then** $t \rightarrow p$ **else** $\neg t \rightarrow q$.

Proof.

$$\begin{array}{ll} & [\![\textbf{if } t \textbf{ then } t \rightarrow p \textbf{ else } \neg t \rightarrow q]\!] \\ = & \qquad \{ \text{ semantics of } \textbf{if-then-else} \text{ and } \rightarrow \} \\ & t \cap t \cap P \cup \bar{t} \cap \bar{t} \cap Q \\ = & \qquad \{ \text{ idempotence of } \cap \} \\ & t \cap P \cup \bar{t} \cap Q \\ = & \qquad \{ \text{ Proposition 74 } \} \\ & [\![\textbf{if } t \textbf{ then } p \textbf{ else } q]\!] \ . \end{array}$$

□

5.3.9 Refining Iteration

Refinement Rule 27 *The statement*

while t **do** p

is equivalent to

while t **do** $t \rightarrow p$.

Proof.

$$\begin{array}{ll}
& [\![\textbf{while } t \textbf{ do } t \to p]\!] \\
= & \qquad \{ \text{ semantics of } \textbf{while} \text{ and } \to \} \\
& \bigsqcap \{X | \overline{t} \cap I \cup t \cap t \cap P \,\Box\, X \sqsubseteq X\} \\
= & \qquad \{ \text{ idempotence of } \cap \} \\
& \bigsqcap \{X | \overline{t} \cap I \cup t \cap P \,\Box\, X \sqsubseteq X\} \\
= & \qquad \{ \text{ semantics of } \textbf{while} \text{ and } \to \} \\
& [\![\textbf{while } t \textbf{ do } p]\!] \ . \qquad \Box
\end{array}$$

Refinement Rule 28 *If* $[\![u]\!] = [\![P]\!]L$*, then the statement*

$\textbf{while } t \textbf{ do } p$

is equivalent to

$\textbf{while } t \wedge u \textbf{ do } p \quad .$

PROOF. Let

$$f.X \triangleq \overline{t} \cap I \cup t \cap P \,\Box\, X \ ,$$

$$f'.X \triangleq (\overline{t} \cup \overline{u}) \cap I \cup t \cap u \cap P \,\Box\, X \ ,$$

$$W \triangleq \bigsqcap \{X | f.X \sqsubseteq X\} \ ,$$

$$W' \triangleq \bigsqcap \{X | f'.X \sqsubseteq X\} \ .$$

Note that W and W' respectively represent the semantics of the first and second **while** statements in the rule. We have

$$\begin{array}{ll}
& W \sqsubseteq W' \\
\Leftrightarrow & \qquad \{ \text{ definition of } W \ \} \\
& \bigsqcap \{X | f.X \sqsubseteq X\} \sqsubseteq W' \\
\Leftarrow & \qquad \{ \text{ property of } \bigsqcap \ \} \\
& W' \in \{X | f.X \sqsubseteq X\} \\
\Leftrightarrow & \qquad \{ \text{ property of } \in \} \\
& f.W' \sqsubseteq W' \ .
\end{array}$$

This last identity is proved by the following deductions:

$$\begin{array}{ll}
& f.W' \\
= & \qquad \{ \text{ definition of } f \ \} \\
& \overline{t} \cap I \cup t \cap P \,\Box\, W' \\
= & \qquad \{ \ u = PL \text{ implies } P = u \cap P \ \} \\
& \overline{t} \cap I \cup t \cap u \cap P \,\Box\, W' \\
\sqsubseteq & \qquad \{ \text{ monotonicity } \} \\
& (\overline{t} \sqcup \overline{u}) \cap I \cup t \cap u \cap P \,\Box\, W' \\
= & \qquad \{ \text{ law 32(n) } \} \\
& (\overline{t} \cup \overline{u}) \cap I \cup t \cap u \cap P \,\Box\, W' \\
= & \qquad \{ \text{ definition of } f' \ \} \\
& f'.W' \\
= & \qquad \{ \ W' \text{ is the least fixpoint of } f'\text{, by definition of } W' \ \} \\
& W' \ . \qquad \Box
\end{array}$$

The main use of this rule is to make a while loop total. Knowing the equivalence between a **while** statement and the Utica pattern **lub-clo-establish**, we could also prove a similar rule for an **lub-clo-establish** structure, but we found that the **while** rule is simpler, and equally useful.

5.4 Introducing New Variables

It is quite common in the process of refining a specification that we need to introduce new variables in order to find a suitable refinement for a statement. The following rule may be used for that task.

Refinement Rule 29 *Let p be a Utica statement with context C_p; let w be a variable declaration with associated context $C_w = \Gamma[\![w]\!]$. If C_p and C_w are disjoint, then the statement p is refined by the statement* **var** *w p.*

Proof. We have to prove by induction on the structure of p that

$$[\![\textbf{var}\ w\ p]\!]_{C_p} \sqsupseteq [\![p]\!]_{C_p} \ .$$

The basis of induction is proved on elementary statements of Utica. We can restrict it to statements **skip**(t) and **rel**, since elementary statements **establish**(t) and **preserve**(t) can be rewritten into a **rel** statement. We have to prove the basis of induction for **skip**(t), because **skip**(t), by its semantic definition, refers to the value of the new variables (it preserves them), whereas **rel** does not, by the hypothesis of the transformation rule. The induction step can be restricted to $\,\Box\,$, $\sqcup$, $\sqcap$ and **var**, since the other compound statements of Utica are defined in terms of these operations: **clo** is defined using $\sqcap$ and $\,\Box\,$; the statement $t \rightarrow p$ is equivalent to **seq skip**(t), p **qes**. In the proofs, we simply write Π for Π_{C_p,C_w}.

Before processding with the proof, let us establish the following fact.

(a) $\widehat{\Pi} \,\Box\, (\Pi R) = R$

$$
\begin{array}{ll}
 & \widehat{\Pi} \,\Box\, (\Pi R) \\
= & \qquad \{ \text{ definition of } \Box \ (33) \ \} \\
 & \widehat{\Pi}\Pi R \cap \overline{\widehat{\Pi}\overline{\Pi R L}} \\
= & \qquad \{ \text{ 68(c) } \} \\
 & R \cap \overline{\widehat{\Pi}\overline{\Pi R L}} \\
= & \qquad \{ \text{ by 68(c), } \Pi \text{ deterministic, 17(i) } \} \\
 & R \cap \overline{\widehat{\Pi}(\Pi\overline{RL} \cup \overline{\Pi L})} \\
= & \qquad \{ \text{ 68(a) } \} \\
 & R \cap \overline{\widehat{\Pi}\Pi\overline{RL}} \\
= & \qquad \{ \text{ 68(c) } \} \\
 & R \cap \overline{\overline{RL}} \\
= & \qquad \{ \text{ 12(m) } \} \\
 & R
\end{array}
$$

Basis of Induction for skip

$$
\begin{aligned}
& [\![\mathbf{var}\ w\ \mathbf{skip}(t)]\!]_{C_p} \\
= \quad & \quad \{\text{ semantics of } \mathbf{var} \text{ and } \mathbf{skip} \} \\
& \widehat{\Pi} \circ ([\![t]\!]_{C_p+C_w} \cap I_{C_p+C_w})\Pi \\
= \quad & \quad \{\ [\![t]\!]_{C_p+C_w} \text{ is a left vector, 34(j), 34(m) } \} \\
& (\widehat{\Pi} \circ [\![t]\!]_{C_p+C_w} \cap \widehat{\Pi})\Pi \\
= \quad & \quad \{\ \widehat{\Pi} \circ [\![t]\!]_{C_p+C_w} \text{ is a left vector, 34(j) } \} \\
& \widehat{\Pi} \circ ([\![t]\!]_{C_p+C_w} L_{(C_p+C_w)\times C_p}) \cap \widehat{\Pi}\Pi \\
= \quad & \quad \{\text{ 68(c) } \} \\
& \widehat{\Pi} \circ ([\![t]\!]_{C_p+C_w} L_{(C_p+C_w)\times C_p}) \cap I_{C_p} \\
= \quad & \quad \{\text{ because } C_w \text{ and } C_p \text{ are disjoint, } [\![t]\!]_{C_p+C_w} = [\![t]\!]_{C_p} \otimes L_{C_w} \} \\
& \widehat{\Pi} \circ (([\![t]\!]_{C_p} \otimes L_{C_w}) L_{(C_p+C_w)\times C_p}) \cap I_{C_p} \\
= \quad & \quad \{\text{ 70(f) } \} \\
& \widehat{\Pi} \circ (\Pi [\![t]\!]_{C_p} \widehat{\Pi} L_{(C_p+C_w)\times C_p}) \cap I_{C_p} \\
= \quad & \quad \{\text{ (a) above, 68(c) } \} \\
& [\![t]\!]_{C_p} \widehat{\Pi} L_{(C_p+C_w)\times C_p} \cap I_{C_p} \\
= \quad & \quad \{\text{ 68(b) } \} \\
& [\![t]\!]_{C_p} L_{C_p} \cap I_{C_p} \\
= \quad & \quad \{\text{ definition of left vector (14) } \} \\
& [\![t]\!]_{C_p} \cap I_{C_p} \\
= \quad & \quad \{\text{ semantics of } \mathbf{skip} \} \\
& [\![\mathbf{skip}(t)]\!]_{C_p}\ . \qquad \Box
\end{aligned}
$$

Basis of Induction for rel

$$
\begin{aligned}
& [\![\mathbf{var}\ w\ \mathbf{rel}\ \mathit{plex}\ \mathbf{ler}]\!] \\
= \quad & \quad \{\text{ semantics of } \mathbf{var} \text{ and } \mathbf{rel} \} \\
& \widehat{\Pi} \circ [\![\mathit{plex}]\!]_{C_p+C_w} \circ \Pi \\
= \quad & \quad \{\text{ because } C_w \text{ and } C_p \text{ are disjoint, } [\![\mathit{plex}]\!]_{C_p+C_w} = [\![\mathit{plex}]\!]_{C_p} \otimes L_{C_w} \} \\
& \widehat{\Pi} \circ [\![\mathit{plex}]\!]_{C_p} \otimes L_{C_w} \circ \Pi \\
= \quad & \quad \{\text{ law 70(f) } \} \\
& \widehat{\Pi} \circ (\Pi [\![\mathit{plex}]\!]_{C_p} \widehat{\Pi}) \circ \Pi \\
= \quad & \quad \{\text{ (a) above, 34(j), 68(c) } \} \\
& [\![\mathit{plex}]\!]_{C_p} \\
= \quad & \quad \{\text{ semantics of } \mathbf{rel} \} \\
& \mathbf{rel}\ \mathit{plex}\ \mathbf{ler}\ .
\end{aligned}
$$

Induction Step for seq:

$$
\begin{aligned}
& [\![\mathbf{var}\ w\ \mathbf{seq}\ q,\ r\ \mathbf{qes}]\!]_{C_p} \\
= \quad & \quad \{\text{ semantics of } \mathbf{var}\text{, semantics of } \mathbf{seq} \} \\
& \widehat{\Pi} \circ [\![q]\!]_{C_p+C_w} \circ [\![r]\!]_{C_p+C_w} \circ \Pi \\
\sqsupseteq \quad & \quad \{\text{ 68(d) } \} \\
& \widehat{\Pi} \circ [\![q]\!]_{C_p+C_w} \circ \Pi \circ \widehat{\Pi} \circ [\![r]\!]_{C_p+C_w} \circ \Pi \\
= \quad & \quad \{\text{ semantics of } \mathbf{var} \} \\
& [\![\mathbf{var}\ w\ q]\!]_{C_p} \circ [\![\mathbf{var}\ w\ r]\!]_{C_p} \\
\sqsupseteq \quad & \quad \{\text{ induction hypothesis } \}
\end{aligned}
$$

$$
\begin{array}{ll}
 & \llbracket q \rrbracket_{C_p} \mathbin{\square} \llbracket r \rrbracket_{C_p} \\
= & \qquad \{ \text{ semantics of } \mathbf{seq} \text{ } \} \\
 & \llbracket \mathbf{seq}\ q,\ r\ \mathbf{qes} \rrbracket_{C_p}\ .
\end{array}
$$

Induction Step for lub (assuming the **lub**'s are defined):

$$
\begin{array}{ll}
 & \llbracket \mathbf{var}\ w\ \mathbf{lub}\ q,\ r\ \mathbf{bul} \rrbracket_{C_p} \\
= & \qquad \{ \text{ semantics of } \mathbf{var}\text{, semantics of } \mathbf{lub} \text{ } \} \\
 & \widehat{\Pi} \mathbin{\square} (\llbracket q \rrbracket_{C_p+C_w} \sqcup \llbracket r \rrbracket_{C_p+C_w}) \mathbin{\square} \Pi \\
\sqsupseteq & \qquad \{ \text{ 34(g) } \} \\
 & (\widehat{\Pi} \mathbin{\square} \llbracket q \rrbracket_{C_p+C_w} \sqcup \widehat{\Pi} \mathbin{\square} \llbracket r \rrbracket_{C_p+C_w}) \mathbin{\square} \Pi \\
\sqsupseteq & \qquad \{ \text{ 34(h) } \} \\
 & \widehat{\Pi} \mathbin{\square} \llbracket q \rrbracket_{C_p+C_w} \mathbin{\square} \Pi \sqcup \widehat{\Pi} \mathbin{\square} \llbracket r \rrbracket_{C_p+C_w} \mathbin{\square} \Pi \\
= & \qquad \{ \text{ semantics of } \mathbf{var} \text{ } \} \\
 & \llbracket \mathbf{var}\ w\ q \rrbracket_{C_p} \sqcup \llbracket \mathbf{var}\ w\ r \rrbracket_{C_p} \\
\sqsupseteq & \qquad \{ \text{ induction hypothesis } \} \\
 & \llbracket q \rrbracket_{C_p} \sqcup \llbracket r \rrbracket_{C_p} \\
= & \qquad \{ \text{ semantics of } \mathbf{lub} \text{ } \} \\
 & \llbracket \mathbf{lub}\ q,\ r\ \mathbf{bul} \rrbracket_{C_p}\ .
\end{array}
$$

Induction Step for glb:

$$
\begin{array}{ll}
 & \llbracket \mathbf{var}\ w\ \mathbf{glb}\ q,\ r\ \mathbf{blg} \rrbracket_{C_p} \\
= & \qquad \{ \text{ semantics of } \mathbf{var}\text{, semantics of } \mathbf{glb} \text{ } \} \\
 & \widehat{\Pi} \mathbin{\square} (\llbracket q \rrbracket_{C_p+C_w} \sqcap \llbracket r \rrbracket_{C_p+C_w}) \mathbin{\square} \Pi \\
= & \qquad \{ \text{ 34(e) } \} \\
 & (\widehat{\Pi} \mathbin{\square} \llbracket q \rrbracket_{C_p+C_w} \sqcap \widehat{\Pi} \mathbin{\square} \llbracket r \rrbracket_{C_p+C_w}) \mathbin{\square} \Pi \\
= & \qquad \{ \text{ 34(f) } \} \\
 & \widehat{\Pi} \mathbin{\square} \llbracket q \rrbracket_{C_p+C_w} \mathbin{\square} \Pi \sqcap \widehat{\Pi} \mathbin{\square} \llbracket r \rrbracket_{C_p+C_w} \mathbin{\square} \Pi \\
= & \qquad \{ \text{ semantics of } \mathbf{var} \text{ } \} \\
 & \llbracket \mathbf{var}\ w\ q \rrbracket_{C_p} \sqcap \llbracket \mathbf{var}\ w\ r \rrbracket_{C_p} \\
\sqsupseteq & \qquad \{ \text{ induction hypothesis } \} \\
 & \llbracket q \rrbracket_{C_p} \sqcap \llbracket r \rrbracket_{C_p} \\
= & \qquad \{ \text{ semantics of } \mathbf{glb} \text{ } \} \\
 & \llbracket \mathbf{glb}\ q,\ r\ \mathbf{blg} \rrbracket_{C_p}\ .
\end{array}
$$

Induction Step for var:

$$
\begin{array}{ll}
 & \llbracket \mathbf{var}\ w\ \mathbf{var}\ w'\ q \rrbracket_{C_p} \\
= & \qquad \{ \text{ definition of } \downarrow\text{, semantics of } \mathbf{var} \text{ } \} \\
 & \widehat{\Pi}_{C_p,C_w} \mathbin{\square} \widehat{\Pi}_{C_p+C_w,C_{w'}} \mathbin{\square} \llbracket q \rrbracket_{C_p+C_w+C_{w'}} \mathbin{\square} \Pi_{C_p+C_w,C_{w'}} \mathbin{\square} \Pi_{C_p,C_w} \\
= & \qquad \{\ \ \widehat{\Pi}_{C_p,C_w} \mathbin{\square} \widehat{\Pi}_{C_p+C_w,C_{w'}} = \widehat{\Pi}_{C_p,C_{w'}} \mathbin{\square} \widehat{\Pi}_{C_p+C_{w'},C_w} \\
 & \qquad \Pi_{C_p+C_w,C_{w'}} \mathbin{\square} \Pi_{C_p,C_w} = \Pi_{C_p+C_{w'},C_w} \mathbin{\square} \Pi_{C_p,C_{w'}} \ \} \\
 & \widehat{\Pi}_{C_p,C_{w'}} \mathbin{\square} \widehat{\Pi}_{C_p+C_{w'},C_w} \mathbin{\square} \llbracket q \rrbracket_{C_p+C_w+C_{w'}} \mathbin{\square} \Pi_{C_p+C_{w'},C_w} \mathbin{\square} \Pi_{C_p,C_{w'}} \\
= & \qquad \{ \text{ semantics of } \mathbf{var}\text{, semantics of } \mathbf{var} \text{ } \} \\
 & \llbracket \mathbf{var}\ w'\ \mathbf{var}\ w\ q \rrbracket_{C_p} \\
\sqsupseteq & \qquad \{ \text{ induction hypothesis } \} \\
 & \llbracket \mathbf{var}\ w'\ q \rrbracket_{C_p}\ . \qquad \square
\end{array}
$$

This completes the presentation of transformation rules.

Chapter 6

Illustration: Sorting by Parts

In this chapter we give a brief illustration of the theory we have introduced so far, by presenting three tentative solutions to the sorting problem. First, we write the specification of this problem in *Utica*. The space of this specification is defined by the following declarations:

```
type  itemtype = record
    key : keytype,
    data : datatype
end

var a : array [1..N] of itemtype
```

where *keytype* defines a totally ordered set (hence it makes sense to *sort* an array of *itemtype*). One aspect of the problem is captured by relation *prm*, which expresses that the output array is a permutation of the input array. The typical specification of a permutation [27, 68] uses a function *bag* that converts an array into a bag*:

$$prm \triangleq \mathbf{rel}\ bag.a = bag.a'\ \mathbf{ler} \quad .$$

Another aspect is captured by relation **establish**(*st*), which expresses that the output is sorted:

$$st \triangleq \forall i \in 1..N-1 : a[i].key \leq a[i+1].key \quad .$$

The combined specification is captured by the specification *sort*, which we write as:

```
Specification sort
var a : array [1..N] of itemtype
lub
    prm, establish(st)
bul .
```

6.1 A First Attempt: A Fast Refinement

Because we cannot proceed without highlighting some structure in the specification, we decide to decompose relation *prm* on the basis of the following premise: one obtains a permutation of a given array by performing zero or more swaps between pairs of cells in the array. Using predicate *swp*, we define relation *swap* as follows:

*A *bag* is like a set with repetitions, that is, an element may have multiple occurences

$$swp(a,i,j) \triangleq a'[j] = a[i] \wedge a'[i] = a[j] \wedge \forall k \in 1..N : k \neq i \wedge k \neq j \Rightarrow a'[k] = a[k] \quad ,$$

$$swap \triangleq \mathbf{rel}\ \exists i,j \in 1..N : swp(a,i,j)\ \mathbf{ler} \quad .$$

Thus, relation *prm* is refined by any number of iterations over statement *swap*, that is *prm* is refined by **clo** *swap* **olc**. Hence we let our first refinement be:

```
lub
   clo $swap$ olc,
   establish($st$)
bul .
```

The refinement Rule 10 provides that a closure statement can be refined by a skip statement. Hence our next refinement is:

```
lub
   skip(true),
   establish($st$)
bul .
```

It is quite trivial to prove that relations I and $\widehat{[\![st]\!]}$ do not satisfy the consistency condition, hence no further refinement can be applied. The design process has come to a dead end, as this *Utica* statement is undefined. This failure results from a precipitous refinement, which transformed the closure into a **skip** —hence seriously reducing our design options.

6.2 A Second Attempt: Simple Sort

We reconsider the specification of the sorting algorithm, as it stood after detailing relation *prm*:

```
lub
   clo $swap$ olc,
   establish($st$)
bul .
```

The refinement Rule 13 provides that the statement

clo *swap* **olc**

is refined by

clo if u **then** *swap* **olc**

for any logical condition u. Because we have in mind to introduce a **while** statement, we take $u = \neg st$. Hence our next refinement is:

```
lub
   clo if $\neg st$ then $swap$ olc,
   establish($st$)
bul .
```

In order to transform this into a **while** statement, we must find a refinement q' of *swap* that satisfies two conditions (ref: Section 4.3.10.4):

1. $[\![\neg st]\!] \subseteq [\![q']\!]L$;
2. $[\![\neg st]\!] \cap [\![q']\!]$ is progressively finite.

If we let q' be statement *orient*, where *orient* swaps two arbitrary values that are out of order, or preserve the state otherwise, then both conditions are satisfied.

$$unord(a,i,j) \triangleq i < j \wedge a[i].key > a[j].key$$

$$orient \triangleq \mathbf{rel}\ (\exists i,j : unord(a,i,j) \wedge swp(a,i,j)) \vee \neg(\exists i,j : unord(a,i,j)) \wedge a = a'\ \mathbf{ler}\ .$$

Relation *orient* is total; thus the first condition is trivially satisfied. To see that the second condition is satisfied, one may use the following reasoning. We say that a pair of array cells $a[i]$ and $a[j]$ is unordered if and only if $unord(a,i,j)$. Using a case analysis, one can show that, for $(s,s') \in [\![\neg st]\!] \cap [\![orient]\!]$, the number of unordered pairs of cells in s' is less than the number of unordered pairs of cells in s. In addition, the number of unordered pairs cannot be negative. Hence, there cannot be an infinite sequence

$$s_0, s_1, \ldots, s_k, \ldots$$

such that $(s_k, s_{k+1}) \in [\![\neg st]\!] \cap [\![orient]\!]$. Moreover, *orient* refines *swap*. Hence our next refinement is:

```
lub
   clo if ¬st then orient olc,
   establish(st)
bul .
```

According to the definition of *Utica*'s **while** statement, this can be rewritten as:

```
while ¬st do
   orient .
```

We leave it to the reader to ascertain that this is a legitimate specification of a sort program, however not a very efficient one, since the loop condition requires scanning the array to determine if it is sorted, and the loop body allows only one swap of cells. Instead of pursuing further in this direction, we consider another refinement path.

6.3 A Third Attempt: Selection Sort

We reconsider the specification of the sorting algorithm, as it stood after detailing relation *prm*:

```
lub
   clo swap olc,
   establish(st)
bul .
```

Instead of targeting a **while** structure, as we have done in the previous attempt, we will rather consider a sequential structure. We start by rewriting the logical expression *st* as the conjunction of two expressions: *pst* and $i = N$, where i is a new variable serving as an index on the array, N is a constant equal to the size of the array, and *pst* means that the array is partially sorted between indices 1 and i, and that the contents of these cells is less than the contents of cells $i+1..N$:

$$pst \triangleq (\forall j \in 1..i-1 : a[j].key \leq a[j+1].key) \wedge a[1..i] \leq a[i+1..N] \quad .$$

To rewrite **establish**(st) as **establish**($pst \wedge i = N$), we must first introduce i as a program variable, which is accomplished using the refinement Rule 29. We then use Rule 14 and Rule 2 to decompose the **establish** statement, and invoking the associativity of the join (hence that of **lub**), we derive the following refinement:

```
var $i : 0..N$
lub
    clo $swap$ olc,
    establish($pst$),
    establish($i = N$)
bul .
```

We now attempt to identify a sequence decomposition on all three components of this **lub**, to then push the **lub** inside the sequence structure. We find, using Rule 15, Rule 16 and Rule 11:

```
lub
    seq clo $swap$ olc, clo $swap$ olc qes,
    seq establish($pst$), preserve($pst$) qes,
    seq establish(true), establish($i = N$) qes
bul .
```

We invoke the refinement Rule 4, and get:

```
seq
    lub clo $swap$ olc, establish($pst$), establish(true) bul,
    lub clo $swap$ olc, preserve($pst$), establish($i = N$) bul
qes .
```

The first term of the **seq** provides that we must establish the condition *pst* while preserving a permutation of the array; this constitutes an initialization of the program, and one way to achieve it is to set i to 1 while keeping the array intact. We concentrate on the second term of the sequence, which is:

```
lub
    clo $swap$ olc,
    preserve($pst$),
    establish($i = N$)
bul .
```

Because of the associativity of the **lub**, we can rewrite this as:

```
lub
    lub clo $swap$ olc, preserve($pst$) bul,
    establish($i = N$)
bul .
```

We now apply Rule 18 to refine **preserve**(pst) by its own closure; whence we rewrite this as follows.

```
lub                                                                    (R1)
    lub clo swap olc, clo preserve(pst) olc bul,
    establish(i = N)
bul .
```

By virtue of the refinement Rule 5, this can be refined into:

```
lub                                                                    (R2)
    clo lub swap, preserve(pst) bul olc,
    establish(i = N)
bul .
```

By virtue of the refinement Rule 13, we refine this as:

```
lub
    clo if $i \neq N$ then q olc,
    establish(i = N)
bul ,
```

where q is defined by

```
lub
    swap, preserve(pst)
bul .
```

To transform the above **lub clo** construct into a **while** statement, we let *pf* (stands for: progressively finite) be the following *Utica* statement:

```
rel
    $i' > i$
ler .
```

We propose to refine q by q', where q' is defined by:

```
lub
    swap, preserve(pst), pf
bul .
```

We leave it to the reader to check that this **lub** statement is indeed defined: it is possible to satisfy all three requirements simultaneously. Statement q' is a refinement of statement q since it is obtained by adding a component *pf* to the join. This yields the following refinement:

```
lub
    clo if $i \neq N$ then $q'$ olc,
    establish(i = N)
bul .
```

Since $[\![i \neq N]\!] = [\![pf]\!]L$ and relation $[\![pf]\!]$ is progressively finite, it follows from Proposition 73 that this statement can be refined by:

```
while $i \neq N$ do $q'$ .
```

Replacing q' by its definition, we obtain

```
while $i \neq N$ do
    lub $swap$, preserve($pst$), $pf$ bul .
```

So that the overall *sort* specification can be refined by the following *Utica* statement:

```
var $i : 0..N$                                                    (R3)
seq
    lub clo $swap$ olc, establish($pst$) bul,
    while $i \neq N$ do
        lub $swap$, preserve($pst$), $pf$ bul
qes .
```

We leave it to the reader to contemplate that this is a legitimate *Utica* description of the *Insertion Sort* algorithm: the loop initialization provides that the array must be partially sorted while remaining a permutation of its initial content; the loop body provides that i must be increased, while preserving the property that the array is partially sorted and is a permutation of its initial content. A simple solution for the loop initialization is $i := 0$. In the next chapters, we shall see rules for solving the loop body. For now, the reader may verify that the sequence

```
var $j : 0..N$
seq
    $i := i + 1$, $j := min(a, i, N)$,
    var $temp : itemtype$
    seq $temp := a[i]$, $a[i] := a[j]$, $a[j] := temp$ qes
qes ,
```

where function $min(a, i, N)$ returns the index of the minimum value in $a[i..N]$, is a solution to the loop body. This completes the presentation of the example.

Part III

Extensions

Chapter 7

Software Modification

Software components are not static entities. A program often undergoes a number of modifications to adapt to new user requirements, new performance requirements or a new platform. We have found that a number of activities pertaining to the adaptation to new user requirements can be formulated as the refinement of **lub** statements. It turns out that the same paradigm that we use for program construction applies as well to program modification. We will briefly review these maintenance activities, and discuss how our mathematical model can assist in carrying them out in a correctness-preserving fashion.

7.1 Software Adaptation

We consider a program whose specification is Pgm and a functional feature $Feature$ that we wish to add to Pgm. We assume that both Pgm and $Feature$ are formulated by *Utica* statements, and we model the problem of modifying Pgm to have feature $Feature$ as the refinement of the following *Utica* statement:

lub Pgm, $Feature$ **bul** .

We may solve this specification using the paradigm of program construction by parts. In order to reuse as much as possible of the implementation of Pgm, our strategy consists in decomposing the new feature to match the structure of Pgm, and to push the **lub** inside the structure. The mathematics of construction by parts will then tell us if the decomposition of Pgm can also satisfy $Feature$: if the **lub** exists, the decomposition is acceptable; if the **lub** does not exist, then a new decomposition of Pgm must be worked out. For instance, assume that the first refinement of the Pgm specification is a sequence.

```
 lub
    seq $Pgm_1$, $Pgm_2$ qes,
    $Feature$
 bul
$\sqsubseteq$          { Find sequential decomposition of $Feature$ }
 lub
    seq $Pgm_1$, $Pgm_2$ qes,
    seq $Feature_1$, $Feature_2$ qes
 bul
$\sqsubseteq$          { push lub inside }
```

```
    seq
        lub $Pgm_1$, $Feature_1$ bul,
        lub $Pgm_2$, $Feature_2$ bul
    qes .
```

We repeat this decomposition process until either an undefined **lub** is found or a refinement of the existing component satisfies the new feature. For instance, if $Feature_1$ is refined by Pgm_1, then the next refinement step is

```
seq
    $Pgm_1$,
    lub $Pgm_2$, $Feature_2$ bul
qes .
```

Observe that we can reuse without change the existing refinement steps implementing Pgm_1 to implement the new feature.

The crux of this approach to software modification is to find a decomposition of *Feature* which can accommodate the decomposition of *Pgm*. In the next chapter, we will study this issue, and propose optimal decompositions for various Utica constructs.

7.2 Software Reuse

We consider a situation which arises very often in software reuse: we are interested in finding a component that satisfies some requirements, and the closest we could find in our database is a component that satisfies parts of the requirements, but not all of them. Let *Request* be the specification of what is requested, and let *Available* be the specification of the closest component found (with respect to some metric which we do not discuss here [62]). The problem of adapting *Available* to satisfy *Request* may be formulated as:

lub *Request*, *Available* **bul** ,

if *Request* and *Available* are consistent. As an example, consider the case when we are looking for an optimized Pascal compiler in a database of compilers, and all we find is an unoptimized compiler. We must refine the following *Utica* statement:

lub *OptCompiler*, *Compiler* **bul** .

We recognize that *OptCompiler* can be decomposed as the sequence of *Compiler* by a code optimizer, and we rewrite our problem as follows:

```
lub
    seq Compiler, Optimizer qes,                                   (R4)
    Compiler
bul .
```

To decompose the second term as a sequence, we recognize that any manipulation of the output code that preserves its correctness can be performed downstream of a compiler; this yields

```
lub
    seq Compiler, Optimizer qes,
    seq Compiler, Preserve qes                                     (R5)
bul .
```

After applying the transformation rules we have presented above, and recognizing that *Optimizer* subsumes *Preserve*, we find the following description.

```
seq
    Compiler,
    Optimizer
qes .
```

Hence our problem can be solved by taking the *Compiler* component that we have retrieved and hooking it to an optimizer, which we must develop or retrieve.

For the sake of further illustration we consider the case where the available component is an optimizing compiler, but works only on a subset of the Pascal language; let it be called *SubOptCompiler*. Our problem can then be formulated as:

lub *OptCompiler*, *SubOptCompiler* **bul** .

If we recognize that an optimizing compiler can be obtained by placing an optimizer downstream of a compiler, and that we go through the same process as we just did for the previous example, we find the following expression:

```
lub
    seq Compiler, Optimizer qes,
    seq SubCompiler, Optimizer qes
bul ,
```

where *SubCompiler* is a compiler of a subset of Pascal. Transforming this specification using Rules 4 and 1, we find:

```
seq
    lub Compiler, SubCompiler bul,
    Optimizer
qes .
```

Given that *SubCompiler* is the specification of the retrieved component and *Compiler* is the specification we are trying to satisfy, one can see that the available component could be of some help in satisfying the specification *Compiler*. Further decompositions of *SubCompiler* must be studied in order to determine if some of them can be reused for solving *Compiler*.

Our paradigm for software reuse may apply as well when the **lub** of the available component and the requested component is undefined, which means that the components have conflicting requirements. If the existing component is specified by parts, then a solution is to remove from its specification the subcomponents leading to inconsistency. Because a program satisfying *Available* is also satisfying any subset of the components in *Available*, we can still reuse the development of *Available* in the solution of *Request*.

Another solution is to prerestrict *Available* to the set of states for which consistency is guaranteed:

```
lub
    $t \rightarrow Available$,
    Request
bul ,
```

where t is computed from the formula

$$[\![t]\!] = AL \cap \overline{RL} \cup (A \cap R)L \ ,$$

where $A \triangleq [\![Available]\!]$ and $R \triangleq [\![Request]\!]$. Because a solution to *Available* is also a solution to $t \rightarrow Available$, we can still reuse the development of *Available* in the solution of *Request*.

A typical example of undefined **lub** is when the behavior of the available component for exceptional conditions (e.g., invalid inputs or file server errors) does not match the required behavior of the requested component: the exceptional conditions considered may be different, or error handling may be different.

7.3 Software Merging

It often occurs in a production environment that a given program will evolve in different streams to accommodate slightly different user requirements. We are given a software system V, of which we have two versions: version V_0, which has feature F_0, and version V_1, which has feature F_1. We are interested in creating version V_2, which has both features F_0 and F_1. We formulate this problem as the refinement of the following *Utica* specification:

```
$V_2 =$
   lub
      $V_0$,
      $V_1$
   bul .
```

Transformations of the kind we have shown above will identify which parts of V_0 and V_1 must be preserved, and which parts must be modified to produce V_2. For the sake of illustration, suppose that V_0 and V_1 can be written as follows:

$$V_0 \triangleq \textbf{seq}\ W,\ W_0\ \textbf{qes} \quad ,$$

$$V_1 \triangleq \textbf{seq}\ W,\ W_1\ \textbf{qes} \quad ,$$

for some statement W. Then we can transform the **lub** of V_0 and V_1 to:

```
seq
   lub $W$, $W$ bul,
   lub $W_0$, $W_1$ bul
qes ,
```

which is equal to

```
seq
   $W$,
   lub $W_0$, $W_1$ bul
qes .
```

Following this step, we know that modifications to V_0 will be localized to W_0, and modifications to V_1 will be localized to W_1. We proceed similarly with the **lub** of W_0 and W_1.

7.4 Perspective

In the previous sections, we have briefly illustrated how the paradigm of program construction by parts applies to the modification of programs, in the case of software adaptation, software reuse and software merging. In each case, the complexity of the task is distributed between the new component and the existing one. The designer does not have to grasp the entire specification at once, but rather by pieces, in an orderly manner: first the designer must consider how the new feature can accommodate the structure of the existing component, and then verify its consistency.

Another key feature of our approach is that most refinement steps of the existing component, which include the tedious proof obligations, are reused as is, something which is rarely feasible in a traditional refinement calculus where join has not been defined. For instance, consider the following refinement of specification *sort* of Chapter 6 in a traditional refinement calculus (e.g., [68], but translated in Utica):

$$
\begin{array}{ll}
 & \mathit{sort} \\
\sqsubseteq & \qquad \{ \text{ refinement laws and predicate calculus } \} \\
 & \mathbf{seq} \\
 & \quad \mathbf{rel}\ i' = 0\ \mathbf{rel}, \\
 & \quad \mathbf{rel}\ \mathit{bag}.a = \mathit{bag}.a' \wedge \mathit{pst} \wedge i' = N\ \mathbf{rel} \\
 & \mathbf{seq}\ .
\end{array}
$$

Assume we want to modify *sort* to compute the sum of the array while sorting. It is clear that this refinement step cannot be used to solve the modified specification of *sort*, given by

$$\mathbf{rel}\ \mathit{bag}.a = \mathit{bag}.a' \wedge \mathit{pst} \wedge i' = N \wedge k' = \textstyle\sum_{i=1}^{N} a[i]\ \mathbf{ler}\ ,$$

because the first component of the sequence does not initialize the variable to accumulate the sum, and the second component does not compute the sum. The designer must redo the calculations, relying on his intuition to figure how the existing refinement step can help him find a new decomposition for the modified *sort* specification. Carrying such a modification using program construction by parts would allow him to reuse the refinement steps of *sort* without changing them. We shall illustrate this on a more elaborate example in Chapter 9.

Chapter 8

Structure Coercion

Our approach to program modification described in the previous chapter rests on the ability to find a decomposition of a new specification to match the structure of an existing component. We call this structure matching problem the structure coercion problem. Preferably, the decompositions should be as weak as possible, in order to maximize the chance of obtaining consistent subcomponents, and to provide the easiest components to refine. Our intent in this chapter is to study the conditions of feasibility and optimality for coercion.

We may formulate this structure coercion problem in a general, abstract manner as follows:

89 Definition.[Structure Coercion] Given an n-ary monotonic operator Φ, and relations P_1, …, P_n and Q such that

$$cs(\Phi(P_1, \ldots, P_n), Q) \ ,$$

find relations $X_1, \ldots, X_n$ such that

1. $Q \sqsubseteq \Phi(X_1, \ldots, X_n) \quad ,$
2. $\forall j \in 1..n : cs(P_j, X_j) \quad . \qquad \Box$

In the next sections, we study the problem defined above for the main Utica constructs: sequence, closure, least upper bound, iteration and alternation.

8.1 Sequence Coercion

8.1.1 The General Sequence Coercion Problem

If we instantiate Definition 89 for sequential composition, we obtain the following definition.

90 Definition.[Sequence Coercion] Given relations P_1, P_2 and Q such that $cs(P_1 \circ P_2, Q)$, find relations X_1 and X_2 such that

1. $Q \sqsubseteq X_1 \circ X_2 \quad ,$
2. $cs(P_1, X_1) \quad ,$
3. $cs(P_2, X_2) \quad . \qquad \Box$

An analysis of these constraints leads us to recognize that there exist multiple solutions to this problem. For instance,

$$(X_1 = Q, X_2 = Q \backslash\!\backslash Q) \text{ and } (X_1 = Q /\!/ Q, X_2 = Q)$$

are two potential solutions. In the presence of multiple solutions, one usually resorts to *least* solutions with respect to some ordering. A legitimate ordering in our case is to select the easiest solution to refine, that is, the least solution wrt to $\sqsubseteq$. With two variables, we may express this criterion as finding a solution (X_1, X_2) such that for any solution (Y_1, Y_2), we have

$$X_1 \sqsubseteq Y_1 \text{ and } X_2 \sqsubseteq Y_2 \ .$$

In general, there exists no least solution wrt to this criterion. The following example on the space $\{0, 1\}$ illustrates it.

$$P_1 \triangleq \{(0,0)\} \text{ and } P_2 \triangleq \{(0,0)\},$$

$$Q \triangleq \{(1,0),(1,1)\}$$

The following are two solutions:

$$X_1 \triangleq \{(1,1)\},\ X_2 \triangleq \{(1,0),(1,1)\} \text{ and}$$

$$Y_1 \triangleq \{(1,0),(1,1)\},\ Y_2 \triangleq \{(0,0),(0,1),(1,0),(1,1)\} \quad .$$

There is no solution that is refined by these two solutions simultaneously. The only lower bound common to X_1 and Y_1 (aside from $\emptyset$) is Y_1, but then Y_1 composed with X_2 does not refine Q. The least solution in Z to the inequation $Q \sqsubseteq Y_1 \,\Box\, Z$ is Y_2, which refines X_2. Hence, there is no common lower bound to (X_1, X_2) and (Y_1, Y_2).

8.1.2 The Restricted Sequence Coercion problem

Since there is no least solution in the general case, we now study the sequence coercion problem when one factor of Q is given. Fortunately, a least solution exists for this restricted problem. The next proposition provides a necessary and sufficient condition for the existence of a solution and a formula for the least solution.

91 Proposition. *[Restricted Sequence Coercion] Let P_1, P_2 and Q be relations such that*

$$cs(P_1 \,\Box\, P_2, Q) \ ,$$

and let Q_1 be a demonic left factor of Q (i.e., $\mathbf{def}(Q_1 \backslash\!\backslash Q)$). Then, a solution in X of the inequations

(a) $Q \sqsubseteq Q_1 \,\Box\, X$
(b) $cs(P_2, X)$

exists if and only if

(c) $cs(P_2, Q_1 \backslash\!\backslash Q)$.

Moreover, the least solution wrt $\sqsubseteq$ is given by $Q_1 \backslash\!\backslash Q$.

Proof. Equations (a) and (b) follow easily from (c) and $X = Q_1 \backslash\!\backslash Q$. For the reverse implication, assume that Y is a solution to (a) and (b). By (a) and law 44(b), we have $Q_1 \backslash\!\backslash Q \sqsubseteq Y$. Using this result, (b) and Proposition 66, we have $cs(P_2, Q_1 \backslash\!\backslash Q)$, which completes the proof of the equivalence. In addition, $Q_1 \backslash\!\backslash Q$ is a solution. It is also the least solution, by law 44(b). □

A dual version of this result, where a demonic right factor for Q is used instead, can be proved in a similar fashion.

8.1.2.1 Conditions for Selecting Factors

When selecting a demonic factor (left or right) of Q in the restricted sequence coercion problem (Proposition 91), it is possible to choose one such that the decomposition of Q is "optimal". The next definition describes what is meant by optimal.

92 Definition. The product $P \mathbin{\Box} Q$ is said to be an *optimal sequential decomposition* of relation R if and only if

$$\begin{array}{ll} \text{(a)} & R \sqsubseteq P \mathbin{\Box} Q \\ \text{(b)} & \forall X, Y : X \sqsubseteq P \wedge Y \sqsubseteq Q \wedge R \sqsubseteq X \mathbin{\Box} Y \Rightarrow P \sqsubseteq X \wedge Q \sqsubseteq Y \ . \qquad \Box \end{array}$$

The next example on the space $\{0, 1\}$ illustrates the notion of optimality:

$$Q \triangleq \{(1,0),(1,1)\} \quad .$$

Decompositions (X_1, X_2) and (Y_1, Y_2) of Q are optimal.

$$X_1 \triangleq \{(1,1)\},\ X_2 \triangleq \{(1,0),(1,1)\} \text{ and}$$

$$Y_1 \triangleq \{(1,0),(1,1)\},\ Y_2 \triangleq \{(0,0),(0,1),(1,0),(1,1)\} \quad .$$

The following decompositions are not optimal:

$$X_1' \triangleq \{(0,0),(0,1),(1,1)\},\ X_2 \triangleq \{(1,0),(1,1)\} \text{ and}$$

$$Y_1 \triangleq \{(1,0),(1,1)\},\ Y_2' \triangleq \{(0,0),(1,1)\} \quad .$$

The product $X_1' \mathbin{\Box} X_2$ is equal to Q, but there exists a relation, namely X_1, that is refined by X_1', and whose composition with X_2 is equal to Q. Similarly, the product $Y_1 \mathbin{\Box} Y_2'$ is equal to Q, but Y_2' refines Y_2, and $Y_1 \mathbin{\Box} Y_2$ is equal to Q. Now, a designer faced with the selection of a demonic factor for Q is best served by selecting X_1 or Y_2 instead of X_1' or Y_2', because they are less refined, hence they leave more freedom in design.

The next proposition provides a necessary and sufficient condition for optimality.

93 Proposition. *The product $P \mathbin{\Box} Q$ is an optimal sequential decomposition of relation R if and only if*

$$P = R /\!\!/ Q \wedge Q = P \backslash\!\!\backslash R \ .$$

Proof. Assume the left-hand side of the equivalence. From 44(a), we have

$$R \sqsubseteq P \mathbin{\Box} Q \Leftrightarrow R /\!\!/ Q \sqsubseteq P \ .$$

Applying the hypothesis of optimality of P and Q on $R /\!\!/ Q$ and Q, we obtain $P \sqsubseteq R /\!\!/ Q$. Similarly, from 44(b), we have

$$R \sqsubseteq P \mathbin{\Box} Q \Leftrightarrow P \backslash\!\!\backslash R \sqsubseteq Q \ .$$

Applying the hypothesis of optimality of P and Q on P and $P \backslash\!\!\backslash R$, we obtain $Q \sqsubseteq P \backslash\!\!\backslash R$, which allows us to conclude the right-hand side of the equivalence. Now assume the right-hand side of the equivalence. Let

$$X \sqsubseteq P \wedge Y \sqsubseteq Q \wedge R \sqsubseteq X \mathbin{\square} Y \ .$$

We have

$$\begin{array}{ll} & R \sqsubseteq X \mathbin{\square} Y \\ \Rightarrow & \quad \{ \ Y \sqsubseteq Q \text{ and monotonicity of } \square \ \} \\ & R \sqsubseteq X \mathbin{\square} Q \\ \Leftrightarrow & \quad \{ \ 44(\text{a}) \ \} \\ & R /\!\!/ Q \sqsubseteq X \\ \Leftrightarrow & \quad \{ \ P = R /\!\!/ Q \ \} \\ & P \sqsubseteq X \ . \end{array}$$

Similarly, we have

$$\begin{array}{ll} & R \sqsubseteq X \mathbin{\square} Y \\ \Rightarrow & \quad \{ \ X \sqsubseteq P \text{ and monotonicity of } \square \ \} \\ & R \sqsubseteq P \mathbin{\square} Y \\ \Leftrightarrow & \quad \{ \ 44(\text{b}) \ \} \\ & P \backslash\!\!\backslash R \sqsubseteq Y \\ \Leftrightarrow & \quad \{ \ Q = P \backslash\!\!\backslash R \ \} \\ & Q \sqsubseteq Y \ . \qquad \square \end{array}$$

We now present some results which allow the derivation of an optimal sequential decomposition using either a demonic right factor or a demonic left factor. Propositions are given only for the demonic right factor, since the results for the demonic left factor are easily derived using the duality between $/\!\!/$ and $\backslash\!\!\backslash$.

94 Definition. Relation Q is said to be an *optimal demonic right factor* of relation R if and only if

(a) Q is a demonic right factor of R
(b) $\forall X : \mathbf{def}(R /\!\!/ X) \wedge X \sqsubseteq Q \wedge R /\!\!/ X \sqsubseteq R /\!\!/ Q \Rightarrow Q \sqsubseteq X \ . \qquad \square$

The following example on space $\{0, 1, 2\}$ illustrates the notion of optimal demonic right factor:

$$Q \triangleq \{(1,0),(1,1)\} \quad .$$

Relations X_1 and X_2 are optimal demonic right factors of Q.

$$X_1 \triangleq \{(1,0),(1,1)\},\ Q /\!\!/ X_1 \triangleq \{(1,1)\} \text{ and}$$

$$X_2 \triangleq \{(0,0),(0,1),(1,0),(1,1)\},\ Q /\!\!/ X_2 \triangleq \{(1,0),(1,1)\} \quad .$$

Note that X_2 is optimal even if $X_1 \sqsubseteq X_2$, because $Q /\!\!/ X_2 \sqsubseteq Q /\!\!/ X_1$. Increasing a demonic right factor in the semilattice results in a decrease of the corresponding demonic left residue, and vice-versa, (which is the same as in arithmetic for an equation $x * y = z$), because $Q /\!\!/ P$ and $P \backslash\!\!\backslash Q$ are antitone in P (see Proposition 45). The following demonic right factors with their left residues are not optimal:

$X_3 \triangleq \{(1,0)\}$, $Q /\!\!/ X_3 \triangleq \{(1,1)\}$;

$X_4 \triangleq \{(1,0),(1,1),(2,2)\}$, $Q /\!\!/ X_4 \triangleq \{(1,1)\}$ and

$X_5 \triangleq \{(0,0),(1,0),(1,1)\}$, $Q /\!\!/ X_5 \triangleq \{(1,0),(1,1)\}$.

Relation X_3 is not optimal, because $X_1 \sqsubseteq X_3$ and $Q /\!\!/ X_1 \sqsubseteq Q /\!\!/ X_3$, but $X_3 \not\sqsubseteq X_1$. Similarly, X_4 is not optimal because of X_1, and X_5 is not optimal because of X_2.

We now provide a necessary and sufficient condition for characterizing an optimal demonic right factor.

95 Proposition. *Relation Q is an* optimal demonic right factor *of relation R if and only if*

$$Q = (R /\!\!/ Q) \mathbin{\backslash\!\!\backslash} R \ .$$

Proof. Assume the left-hand side of the equivalence. By 46(a), we have

$$(R /\!\!/ Q) \mathbin{\backslash\!\!\backslash} R \sqsubseteq Q \ .$$

By 46(c), we have

$$R /\!\!/ ((R /\!\!/ Q) \mathbin{\backslash\!\!\backslash} R) \sqsubseteq R /\!\!/ Q \ .$$

Using the hypothesis that Q is an optimal right factor, we obtain

$$Q \sqsubseteq (R /\!\!/ Q) \mathbin{\backslash\!\!\backslash} R \ ,$$

and we conclude the right-hand side. Now, assume the right-hand side. Let X be a relation satisfying the following properties:

$$X \sqsubseteq Q, \\ R /\!\!/ X \sqsubseteq R /\!\!/ Q \ .$$

We have

$$R \sqsubseteq (R /\!\!/ X) \mathbin{\scriptstyle\square} X \sqsubseteq (R /\!\!/ Q) \mathbin{\scriptstyle\square} X \ ,$$

which implies, using 44(b),

$$Q = (R /\!\!/ Q) \mathbin{\backslash\!\!\backslash} R \sqsubseteq X \ . \qquad \Box$$

96 Corollary. *If Q is an optimal demonic right factor of R, then $(R /\!\!/ Q) \mathbin{\scriptstyle\square} Q$ is an optimal sequential decomposition of R.*

Proof. The result follows directly from Proposition 93 and Proposition 95. $\Box$

97 Corollary. *If $P \mathbin{\scriptstyle\square} Q$ is an optimal sequential decomposition of R, then Q is an optimal demonic right factor of R.*

Proof. The result follows directly from Proposition 93 and Proposition 95. $\Box$

8.1.2.2 Heuristics for Deriving Factors

We now provide a number of ways of constructing optimal decompositions. The following proposition provides that, from any demonic right factor Q of a relation R, one can derive an optimal sequential decomposition.

98 Corollary. *If Q is a demonic right factor of R, then*

$$(R/\!\!/Q) \mathbin{\square} ((R/\!\!/Q)\backslash\!\!\backslash R)$$

is an optimal sequential decomposition of R.

Proof. By 46(c), we have $R/\!\!/((R/\!\!/Q)\backslash\!\!\backslash R) = R/\!\!/Q$. Hence, we conclude using Proposition 93. □

The dual proposition for a demonic left factor is proved in a similar manner.

99 Corollary. *The product $(R/\!\!/R) \mathbin{\square} R$ is an optimal sequential decomposition of R.*

Proof. Since $R \sqsubseteq I \mathbin{\square} R$, R is a demonic right factor of R. By Proposition 98, we have that $(R/\!\!/R) \mathbin{\square} ((R/\!\!/R)\backslash\!\!\backslash R)$ is an optimal sequential decomposition of R. By law 46(o), we also have that $(R/\!\!/R) \mathbin{\square} R$ is an optimal sequential decomposition of R. □

The dual corollary for $R \mathbin{\square} (R\backslash\!\!\backslash R)$ is proved in a similar manner.

Since demonic residues are difficult to compute in general, we propose a way of constructing an optimal sequential decomposition when a relation is regular.

100 Proposition. *Let R be a regular relation and let P be a deterministic relation such that $RL \subseteq \widehat{P}L$. Then $(R\widehat{R}\widehat{P}) \mathbin{\square} (PR)$ is an optimal sequential decomposition of R.*

Proof. The following lemma is easily proved using the Dedekind rule and the facts that P is deterministic and $RL \subseteq \widehat{P}L$.

(a) $\widehat{P}PR = R$

Using this lemma, we prove a second lemma indicating that the two expressions are demonic factors of R.

(b) $(R\widehat{R}\widehat{P}) \mathbin{\square} (PR) = (R\widehat{R}\widehat{P})(PR) = R$

$$\begin{array}{ll} & (R\widehat{R}\widehat{P}) \mathbin{\square} (PR) \\ = & \qquad \{\ 34(\mathrm{j})\ \} \\ & (R\widehat{R}\widehat{P})(PR) \\ = & \qquad \{\ \text{Lemma (a)}\ \} \\ & R\widehat{R}R \\ = & \qquad \{\ R \text{ is regular}\ \} \\ & R\ . \end{array}$$

The following lemma is used in the computation of the demonic right residue.

(c) $\widehat{R}R\widehat{R}\widehat{P}PR\widehat{R} = \widehat{R}$

$\widehat{R}R\widehat{R}\widehat{P}PR\widehat{R}$

$=$ { R regular and 50(c) }

$\widehat{R}\widehat{P}PR\widehat{R}$

$=$ { Lemma (a) }

$\widehat{R}R\widehat{R}$

$=$ { R regular and 50(c) }

$\widehat{R}$.

Now we prove the optimality of the decomposition by showing that the proposed demonic right factor is optimal (Proposition 95).

$(R /\!\!/ (PR)) \backslash\!\!\backslash R$

$=$ { Lemma (b) and 39(b) }

$(R\widehat{R}\widehat{P}) \backslash\!\!\backslash R$

$=$ { definition of $\backslash\!\!\backslash$ }

$\kappa(\widehat{R}, (RL \cap R\widehat{R}\widehat{P})^{\frown})^{\frown}$

$=$ { Boolean laws }

$\kappa(\widehat{R}, (R\widehat{R}\widehat{P})^{\frown})^{\frown}$

$=$ { 39(b), Lemma (c) }

$(\widehat{R}R\widehat{R}\widehat{P})^{\frown}$

$=$ { R regular and 50(c) }

$(\widehat{R}\widehat{P})^{\frown}$

$=$ { $^{\frown}$ law }

PR .

By Lemma (b), definition of $/\!\!/$ and Proposition 39(b), we have that $R /\!\!/ (PR) = R\widehat{R}\widehat{P}$. □

Note also that this decomposition produces two regular relations. The next proposition generalizes this fact.

101 Proposition. *If P and Q form an optimal sequential decomposition of a regular relation, then P and Q are regular.*

Proof. It follows from Proposition 93 and Definition 41 that $P = \kappa(R, Q)$ and $Q = \kappa(\widehat{R}, (RL \cap P)^{\frown})^{\frown}$. By Proposition 52, P is regular. By Proposition 52 and law 50(c), Q is regular. □

We now study the connection between optimality and relative regularity (Definition 47).

102 Proposition. *If $R \sqsubseteq R\widehat{Q}Q$ and $Q \sqsubseteq Q\widehat{R}R$, then $R\widehat{Q}$ and Q form an optimal sequential decomposition of R, and $Q\widehat{R}$ and R form an optimal sequential decomposition of Q. In addition R and Q are regular.*

Proof. We only prove the first consequent and the regularity of R. The proof of the second consequent and the regularity of Q are similar. From the hypotheses, Proposition 39 and definition of $\sqsubseteq$, we have

(a) $R /\!\!/ Q = R\widehat{Q}$

(b) $R\widehat{Q}Q \subseteq R$

(c) $Q\widehat{R}R \subseteq Q$

We also need the following lemma, which follows easily from (b) and (c):

(d) $\widehat{R}R\widehat{Q}Q\widehat{R} \subseteq \widehat{R}$

$$\begin{array}{ll}
 & \widehat{R}R\widehat{Q}Q\widehat{R} \\
\subseteq & \qquad \{ \text{(c) and } \widehat{\ } \text{ laws} \} \\
 & \widehat{Q}Q\widehat{R} \\
\subseteq & \qquad \{ \text{(b) and } \widehat{\ } \text{ laws} \} \\
 & \widehat{R} \ .
\end{array}$$

We now prove optimality using Proposition 95.

$$\begin{array}{ll}
 & (R /\!\!/ Q) \backslash\!\!\backslash R \\
= & \qquad \{ \text{(a)} \} \\
 & (R\widehat{Q}) \backslash\!\!\backslash R \\
= & \qquad \{ \text{definition of } \backslash\!\!\backslash \text{ and law 39(b)} \} \\
 & \kappa(\widehat{R}, (RL \cap R\widehat{Q})^{\frown})^{\frown} \\
= & \qquad \{ \text{Boolean laws} \} \\
 & \kappa(\widehat{R}, (R\widehat{Q})^{\frown})^{\frown} \\
= & \qquad \{ \text{(d) and Proposition 39} \} \\
 & (\widehat{R}R\widehat{Q})^{\frown} \\
= & \qquad \{ \text{property of } ^{\frown} \} \\
 & Q\widehat{R}R \\
\sqsupseteq & \qquad \{ \text{hypothesis} \} \\
 & Q \ .
\end{array}$$

By law 46(a), we have $(R /\!\!/ Q) \backslash\!\!\backslash R = Q$. Using Proposition 95, we conclude that the decomposition $(R\widehat{Q}) \,\square\, Q$ is optimal. The regularity of R follows from (d) using (b) and 50(c). $\square$

8.1.3 Transformation Rules

Now that we have determined the feasibility conditions and optimality conditions for coercion into a sequence, we are ready to propose transformation rules. Let us briefly restate the problem.

Given statements p_1, p_2 and q such that the statement

lub seq p_1, p_2 **qes**, q **bul**

is defined, find statements q_1 and q_2 such that

- $q \sqsubseteq$ **seq** q_1, q_2 **qes**,
- the statements **lub** p_1, q_1 **bul** and **lub** p_2, q_2 **bul** are defined.

Because the general sequence coercion problem has no optimal solution, we concentrate on the restricted sequence coercion problem. To this effect, we consider various candidates for a demonic factor of q, the component to coerce. By looking at the problem description, three natural choices arise: q, p_1 and p_2. The mathematics will also highlight a fourth choice: the assignment statement.

8.1.3.1 Structure Coercion in Duet

When selecting q as its own factor, we obtain the following two optimal sequential decompositions (Corollary 99), where the demonic residue operators are extended to Utica statements.

Refinement Rule 30 *The statement q is equivalent to* **seq** *q, $q \backslash\!\!\backslash q$* **qes**. □

Refinement Rule 31 *The statement q is equivalent to* **seq** *$q /\!\!/ q$, q* **qes**. □

Rules 11, 15, 16 and 17 of Chapter 5 are instances of Rule 30 and Rule 31. Hence, they are optimal.

We call this kind of coercion a duet, because it represents the case where the designer wants to implement the new feature (q) with either p_1 or p_2. The next example illustrates our denomination.

```
    lub
       seq p_1, p_2 qes,
       q
    bul
⊑             { Rule 30 }
    lub
       seq p_1, p_2 qes,
       seq q, q\\q qes
    bul
⊑             { push lub: Rule 4 }
    seq                                   (R6)
       lub p_1, q bul,
       lub p_2, q\\q bul
    qes
```

or, if Rule 31 is used instead,

```
    seq                                   (R7)
       lub p_1, q//q bul,
       lub p_2, q bul
    qes .
```

At first glance, expressions like $q /\!\!/ q$ and $q \backslash\!\!\backslash q$ may seem awkward to use and meaningless, given the complex definitions of $/\!\!/$ and $\backslash\!\!\backslash$. In practice, it turns out that $q /\!\!/ q$ and $q \backslash\!\!\backslash q$ are intuitive notions for real specification: they represent the minimal information that must be preserved when q is implemented in duo with another specification. For instance, let q be the following specification:

var x : **Natural**, a : array[1..N] of **Natural**

rel $x' = \Sigma_{i=1}^{N} a[i]$ **ler** .

Specification q is regular. The result of evaluating $q \backslash\!\!\backslash q$ and $q /\!\!/ q$ is given by

rel $x' = x$ **ler**

and

rel $\Sigma_{i=1}^{N} a[i] = \Sigma_{i=1}^{N} a'[i]$ **ler**,

respectively. Refinement R6 (page 106) represents the case where the designer chooses to implement q with p_1, and preserve the value of x while implementing p_2. Refinement R7 (page 106) represents the case where the designer chooses to implement q with p_2, and preserve the sum of the array while implementing p_1.

8.1.3.2 Structure Coercion in Solo

Another use of Rule 30 and Rule 31 is to implement a feature either before of after an existing component. For instance,

```
    lub p, q bul
⊑          { Rule 31 }
    lub
       seq $p /\!\!/ p$, p qes,
       q
    bul
⊑          { Rule 30 }
    lub
       seq $p /\!\!/ p$, p qes,
       seq q, $q \backslash\!\!\backslash q$ qes
    bul
⊑          { push lub: Rule 4 }
    seq
       lub $p /\!\!/ p$, q bul,
       lub p, $q \backslash\!\!\backslash q$ bul
    qes
```

Our denomination "in solo" stems from the following observations: the first component of the sequence implements q while preserving the critical information required by p (given by $p /\!\!/ p$); the second component implements p while preserving the results obtained by q (given by $q \backslash\!\!\backslash q$).

8.1.3.3 Structure Coercion in Refrain

Another choice for a demonic factor of q is either p_1 or p_2. It leads to the following two rules.

Refinement Rule 32 *If* **def**$(p_1 \backslash\!\!\backslash q)$*, then the statement*

```
lub
   seq $p_1$, $p_2$ qes,
   q
bul
```

is refined by

```
lub
   seq $p_1$, $p_2$ qes,
   seq $p_1$, $p_1 \backslash\!\!\backslash q$ qes
bul .     □
```

Our denomination “in refrain” stems from the observation that p_1 is used twice in the solution of components. This rule has the advantage of reducing the number of specifications to solve. If we apply Rule 4 to its consequent, we obtain

```
seq
    lub $p_1$, $p_1$ bul,
    lub $p_2$, $p_1 \backslash\!\!\backslash q$ bul
qes ,
```

if the second join exists. In turn, this is equivalent, by idempotence of join, to

```
seq
    $p_1$,
    lub $p_2$, $p_1 \backslash\!\!\backslash q$ bul
qes .
```

If we select p_2 as a demonic right factor for q, then we obtain the following dual rule.

Refinement Rule 33 *If* **def**$(q /\!\!/ p_2)$*, then the statement*

```
lub
    seq $p_1$, $p_2$ qes,
    $q$
bul
```

is refined by

```
lub
    seq $p_1$, $p_2$ qes,
    seq $q /\!\!/ p_2$, $p_2$ qes
bul .
```

□

As for the previous rule, the consequent reduces to the following statement, after applying Rule 4 and idempotence.

```
seq
    lub $p_1$, $q /\!\!/ p_2$ bul,
    $p_2$
qes .
```

Rules 32 and 33 are not optimal for the decomposition of q. However, it is irrelevant in this case, because optimality is lost when the lub is pushed inside the sequence. For instance, if the optimal demonic right factor $q /\!\!/ (p_1 \backslash\!\!\backslash q)$ is selected instead of p_1, then we obtain:

```
    lub
        seq $p_1$, $p_2$ qes,
        q
    bul
$\sqsubseteq$          { dual of Proposition 98 }
    lub
        seq $p_1$, $p_2$ qes,
        seq $q /\!\!/ (p_1 \backslash\!\!\backslash q)$, $p_1 \backslash\!\!\backslash q$ qes
    bul
$\sqsubseteq$           { push lub: Rule 4 }
    seq
        lub $p_1$, $q /\!\!/ (p_1 \backslash\!\!\backslash q)$ bul,
        lub $p_2$, $p_1 \backslash\!\!\backslash q$ bul
    qes
$\sqsubseteq$           { law 46(b) }
    seq
        $p_1$,
        lub $p_2$, $p_1 \backslash\!\!\backslash q$ bul
    qes
```

which is the same final result as when p_1 is used (Rule 32).

8.1.3.4 Structure Coercion by Independent Steps

Finally, we provide the following rules, which are used when neither p_1, p_2 or q are acceptable candidates.

Refinement Rule 34 *if* **def**$(q /\!\!/ r)$*, then the statement q is refined by the statement*

$$\textbf{seq}\ q /\!\!/ r,\ r\ \textbf{qes}\ . \qquad \Box$$

Refinement Rule 35 *if* **def**$(r \backslash\!\!\backslash q)$*, then the statement p is refined by the statement*

$$\textbf{seq}\ r,\ r \backslash\!\!\backslash q\ \textbf{qes}\ . \qquad \Box$$

The computation of the demonic left residue in Rule 34 is simple when r is an assignment statement. Law 48 provides that the demonic left residue exists and is given by $P\widehat{Q}$ if and only if $P \sqsubseteq P\widehat{Q}Q$. In our case, the expression $\widehat{[\![r]\!]}[\![r]\!]$ is equal to $I \cap \widehat{[\![r]\!]}L$, because an assignment statement is a deterministic relation. This observation gives rise to the following transformation rule.

Refinement Rule 36 *Let p be a Utica statement whose context contains variable x; let e be an expression. If $L[\![p]\!] \subseteq L[\![x := e]\!]$, then*

$$p /\!\!/ (x := e) = p[x' \backslash e']\ ,$$

where e' is defined as $e[w \backslash w']$, and w is the list of variables in the context of p. $\Box$

8.2 Closure Coercion

8.2.1 The General Closure Coercion Problem

If we instantiate Definition 89 for demonic closure, we obtain the following definition.

103 Definition.[Closure Coercion] Given relations P and Q such that $cs(P^{\circledast}, Q)$, find relation X such that

1. $Q \sqsubseteq X^{\circledast}$,

2. $cs(P, X)$.

In general, there are multiple solutions to these two inequations, and there does not exist a least solution with respect to the refinement ordering. The following example on the space $\{0, 1, 2\}$ illustrates this fact:

$$P \triangleq Q \triangleq \{(0,0), (0,1), (1,1), (1,2), (2,2), (2,0)\} \quad .$$

There are three minimal solutions for Q:

$$X_1 \triangleq \{(0,0), (0,1), (1,1), (2,2)\} \quad ,$$

$$X_2 \triangleq \{(0,0), (1,1), (1,2), (2,2)\} \quad ,$$

$$X_3 \triangleq \{(0,0), (1,1), (2,2), (2,0)\} \quad .$$

The identity is also a solution, but it is not minimal. The following proposition provides a necessary and sufficient condition for the existence of a solution to inequation 1 in the closure coercion problem.

104 Proposition. *Inequation 1 in Definition 103 has a solution if and only if $Q \sqsubseteq I$.*

Proof.

$$\begin{array}{ll}
 & \exists X : Q \sqsubseteq X^{\boxast} \\
\Rightarrow & \qquad \{\ X^{\boxast} \sqsubseteq I \text{ for any } X\ \} \\
 & Q \sqsubseteq I \\
\Rightarrow & \qquad \{\ I = I^{\boxast}\ \} \\
 & Q \sqsubseteq I^{\boxast} \\
\Rightarrow & \\
 & \exists X : Q \sqsubseteq X^{\boxast} \quad . \qquad \square
\end{array}$$

8.2.1.1 Heuristics for Deriving Demonic Transitive Roots

The following proposition provides solutions to inequation 1 in the closure coercion problem.

105 Proposition. *If $Q \sqsubseteq I$, then $Q /\!\!/ Q$ and $Q \backslash\!\!\backslash Q$ are solutions to inequation 1 in Definition 103.*

Proof. Assume $Q \sqsubseteq I$. We have:

$$\begin{array}{ll}
 & \textbf{true} \\
\Leftrightarrow & \qquad \{\ \text{law 46(i)}\ \} \\
 & Q \sqsubseteq (Q /\!\!/ Q) \mathbin{\square} Q \\
\Rightarrow & \qquad \{\ \text{hypothesis, monotonicity of } \mathbin{\square}\ \} \\
 & Q \sqsubseteq (Q /\!\!/ Q) \mathbin{\square} I \\
\Leftrightarrow & \qquad \{\ \text{law 34(b), law 46(k)}\ \} \\
 & Q \sqsubseteq (Q /\!\!/ Q)^{\boxast} \quad .
\end{array}$$

The proof of $Q \backslash\!\!\backslash Q$ as a solution is derived in a similar manner using the duality between $/\!\!/$ and $\backslash\!\!\backslash$. $\square$

8.2.2 Transformation Rules

The next two rules stem from Proposition 105.

Refinement Rule 37 *If statement q is refined by* **skip***, then it is refined by* **clo** $q /\!\!/ q$ **olc***.*
□

Refinement Rule 38 *If statement q is refined by* **skip***, then it is refined by* **clo** $q \backslash\!\!\backslash q$ **olc***.*
□

Solutions $Q /\!\!/ Q$ and $Q \backslash\!\!\backslash Q$ are rarely minimal, except in the case where $Q = Q /\!\!/ Q$ (or $Q = Q \backslash\!\!\backslash Q$), which is equivalent to $Q = Q^{\boxed{*}}$ by law 46(k). This case is trivial, but fortunately it often occurs in practice.

In the next example, we see that $Q /\!\!/ Q$ is a maximal solution wrt $\sqsubseteq$. Let $\{0, 1, 2, 3\}$ be the space of the following relation.

$$Q \triangleq \{(1,0),(1,1),(1,2),(2,1),(2,2),(2,3)\} \quad .$$

The optimal solution is

$$\{(1,1),(1,2),(2,1),(2,2)\} \quad ,$$

and the solution $Q /\!\!/ Q$ is given by

$$\{(1,1),(2,2)\} \quad .$$

The next rule is the general mechanism for refining a relation by a closure.

Refinement Rule 39 *If statements p and q satisfy the condition*

$$\forall i \geq 0 : [\![p]\!] \sqsubseteq [\![q]\!]^{\boxed{i}} \quad ,$$

then statement p is refined by statement **clo** q **olc***.*

Proof.

$$\begin{array}{ll}
 & \forall i : i \geq 0 : P \sqsubseteq Q^{\boxed{i}} \\
\Leftrightarrow & \qquad \{ \text{ definition of lower bound } \} \\
 & P \in \{Q^{\boxed{i}} \mid i \geq 0\}^{l} \\
\Leftrightarrow & \qquad \{ \text{ definition of greatest lower bound } \} \\
 & P \sqsubseteq \bigsqcap_{i \geq 0} Q^{\boxed{i}} \\
\Leftrightarrow & \qquad \{ \text{ definition of } \textbf{clo} \text{ } \} \\
 & [\![p]\!] \sqsubseteq [\![\textbf{clo } q \textbf{ olc}]\!] \quad . \qquad \square
\end{array}$$

The next four rules are more specific cases of refinement by a closure.

Refinement Rule 40 *If statement p satisfies the conditions*

$$p \sqsubseteq \textbf{skip} \quad \textit{and} \quad p \sqsubseteq \textbf{seq } p,\ p \textbf{ qes} \quad ,$$

then it is equivalent to the statement **clo** p **olc***.*

Proof. See law 37(b). □

Refinement Rule 41 *The statement $p\backslash\!\backslash p$ is equivalent to the statement* **clo** $p\backslash\!\backslash p$ **olc***.*

Proof. See law 46(l). □

Rule 41 is often used in conjunction with Rule 30 for coercion with a **while** statement.

Refinement Rule 42 *The statement $p /\!/ p$ is equivalent to the statement* **clo** $p /\!/ p$ **olc***.*

Proof. See law 46(k). □

Rule 42 is often used in conjunction with Rule 31 for coercion with a **while** statement. The next rule has a very simple proviso to compute.

Refinement Rule 43 *If $[\![p]\!] = [\![p]\!]\widehat{[\![p]\!]}$ (i.e., $[\![p]\!]$ is a PER by Proposition 54), then the statement p is equivalent to the statement* **clo** p **olc***.*

Proof. See Proposition 55. □

8.3 Other Coercions

The other interesting cases of coercion are the **lub** construct, the **if** construct and the **while** construct.

The coercion of a relation into a **lub** statement may seem useless at first glance, because our paradigm assumes that specifications are structured using **lub**'s. However, it often occurs that components of a specification must be further decomposed in terms of **lub**'s in order to find a suitable solution, because a specifier must promote minimality, completeness and clarity, rather than implementation issues. We provide two rules for **lub** coercion, on the basis of laws 32(i),(j) and (q).

Refinement Rule 44 *If the* **lub** *is defined, the statement*

> **rel** $pred_1 \wedge pred_2$ **ler**

is refined by

> **lub**
>   **rel** $pred_1$ **ler**,
>   **rel** $pred_2$ **ler**
> **bul** . □

Refinement Rule 45 *If the* **lub** *is defined, the statement*

> **rel** $pred_1 \vee pred_2$ **ler**

is refined by

> **lub**
>   **rel** $pred_1$ **ler**,
>   **rel** $pred_2$ **ler**
> **bul** . □

Refinement Rule 46 *The statement p is refined by*

```
lub
    $t \rightarrow p$,
    $\neg t \rightarrow p$
bul .    □
```

The coercion of a relation into an **if** statement is just a special case of Rule 46, by the equivalence between an **if** statement and an **lub** statement.

Refinement Rule 47 *The statement q is equivalent to*

```
if $t$ then $q$ else $q$ .    □
```

The coercion problem for the **while** construct has already been addressed in [61], where it is shown that if a specification is refined by a **while** statement, then the specification is in *first iterative form*. Note that in the construction by parts paradigm, one does not necessarily have to coerce a specification into a **while** construct. Under some conditions, the equivalence between a **while** statement and an **lub** statement (see Section 4.3.10.4, page 63) allows one to transform a specification

```
lub
    while $t$ do $p$,
    $q$
bul
```

into the specification

```
lub
    clo if $t$ then $p$ olc,
    establish($\neg t$),
    $q$
bul ,
```

and then coerce q into a closure. This completes the study of coercion.

Chapter 9

Case Study

In this chapter, we further illustrate our paradigm for program construction/modification on more complex problems. First, we use the construction paradigm to solve the well-known paragraph formatting problem. We shall see that the rules of structure coercion introduced in Chapter 8 are not only useful in program modification, but also in program construction. Second, we use the modification paradigm to adapt the *sort* specification of Chapter 6 for new requirements. In addition, we revisit the compiler example of Chapter 7 to illustrate how the intuitive decompositions used in this example are in fact given by general coercion rules.

9.1 Program Construction

The Naur problem has drawn the interest of several researchers in the past [10, 32, 55, 56, 58, 72, 73, 85] The purpose of this section is not to solve the Naur problem as much as it is to illustrate our method. In order to illustrate the versatility of our notation and method, we will investigate alternative design choices and discuss their merits. If our purpose was to find a solution to the problem, as opposed to illustrate our method, then our discussion would have been a lot shorter.

9.1.1 The User Requirements

First, we present the English version of this problem, as given by [58].

> "Given are a non-negative integer M and a character set which includes two *break* characters, viz. $\not b$ (*blank*) and $\not n$ (*newline*). For a sequence s of characters, we define a *word* as a non-empty sequence of consecutive non-break characters embedded between break characters or the endpoints of s (i.e., the beginning and end of sequence s). The program shall accept as input a finite sequence of characters and produce as output a sequence of characters satisfying the following conditions.
>
> 1. If the input includes at least one break character for any consecutive $M + 1$ characters, then all the words of the input appear in the output, in the same order; all the words of the output appear in the input.
> 2. Furthermore, the output must meet the following conditions:
>   (a) It contains no leading or trailing breaks, nor does it have two consecutive breaks.
>   (b) Any sequence of $M + 1$ consecutive characters includes a newline.

(c) Any sequence made up of no more than M consecutive characters and embedded between (the head of the output or a newline on the left) and (the tail of the output or a break on the right) does not contain a newline character".

9.1.2 The Relational Specification

Before we proceed with writing a formal specification for this problem, we introduce the following notations: we denote by *chr* the set of characters under consideration, including the *newline* symbol; also, we let this set be partitioned into the set of *text* characters, which we denote by *txt* and the set of *breaks*, which we denote by *brk*; we let $x \bullet y$ stand for the concatenation of sequences x and y; we let ϵ denote the empty sequence; we let X^+ denote the closure under "$\bullet$" of the set of sequences X; we let X^* denote $X^+ \cup \{\epsilon\}$; we let $\#.x$ denote the length of sequence x; we let $x \preceq y$ be the predicate "x is a subsequence of consecutive characters of y"; we let *fw.x* denote the first word of sequence x, provided that x contains a word; we let *rw.x* denote the sequence beginning after the first word of x and terminating at the end of x, provided that x contains a word; we let *lw.x* be the Boolean value "sequence x contains a word of length $> M$"; we let *lfw.x* be the Boolean value "the length of the first word of sequence x is $> M$"; we let *sw* be a function that maps a sequence of characters x to the sequence of words of x (e.g., $sw.\langle ab\not{b}cd\rangle = \langle\langle ab\rangle\langle cd\rangle\rangle$). Function *sw* satisfies the following condition:

106 $sw.x = sw.y \bullet sw.z \Rightarrow (lw.x \Leftrightarrow lw.y \vee lw.z)$.

The space of our specification is given by the following declaration:

$$\textbf{var } in, out : \textbf{seq } chr \ .$$

We let relation *binary* represent clause 1 of the user requirements. It can be written as

$$binary \triangleq \textbf{rel } \neg lw.in \wedge sw.in = sw.out' \textbf{ ler} \quad .$$

Clauses 2a, 2b and 2c are conditions on the output only. They are represented by an **establish** statement, for which we define the following predicates:

clause 2a:
$exb \triangleq out \in txt^+ \bullet (brk \bullet txt^+)^* \cup \{\epsilon\}$

clause 2b:
$sht \triangleq \forall x : x \preceq out \wedge \#.x = M + 1 \Rightarrow \not{n} \preceq x$

clause 2c:
$$lng \triangleq \forall x, y, z : out = x \bullet y \bullet z \wedge x \in chr^* \bullet \not{n} \cup \{\epsilon\} \wedge \#.y \leq M \wedge$$
$$z \in brk \bullet chr^* \cup \{\epsilon\} \Rightarrow \not{n} \not\preceq y \ .$$

We define the predicate *unary* as

$$unary \triangleq exb \wedge sht \wedge lng \ .$$

We represent clause 2 by the statement

$$\textbf{establish}(unary) \ .$$

The complete specification of the Naur problem is given by the following:

```
**lub**
    *binary*, *unary*
**bul** .
```

We leave it to the reader to check that the components are consistent, hence the join is defined. The structure of the Naur specification is typical of join-structured specifications: some components represent the relationship between initial states and final states (i.e., *binary*), whereas others, defined by right vectors, represent a condition on the output (i.e., **establish**(*unary*)).

9.1.3 The Solution

Our strategy is to solve *binary* and *unary* independently, and then to progressively combine their solutions to come up with a program satisfying the complete specification.

9.1.3.1 Solving Unary.

We propose an iterative solution for *unary*, which is general enough to preserve consistency with *binary*, but highlights the structure of a solution to *unary*.

```
   establish(unary)
=          { Rule 15 }
   seq
      establish(unary),
      preserve(unary)
   qes
=          { Rule 18 }
   seq                                                    (R8)
      establish(unary),
      clo preserve(unary) olc
   qes
```

We stop the refinement process at this point in order not to overcommit ourselves. However, it is interesting to see, just for the sake of the exercise, that this solution can be developed further to derive a “complete” solution to *unary*, that is, a solution that would generate all possible final values of *out* satisfying the specification, assuming that a nondeterministic implementation of the language is available. The term **establish**(*unary*) may be refined by $out := \epsilon$. The refinement of **preserve**(*unary*) is more complex. One possible solution is to nondeterministically generate a word of length less than or equal to M, and to append it to *out*. In doing so, we must make sure that lines are not too short and not too long, and that only one break separates each word. That requires “scanning” variable *out* to determine if a $\not{b}$ or a $\not{n}$ must be inserted before the word to append. Adding a variable which keeps track of the length of the last line of *out* removes this obligation of scanning *out*. We introduce a new variable k, by virtue of Rule 29, and we define the following predicate:

var k : **Integer**

$$count \triangleq \exists x : \not{n} \not\preceq x \wedge out \in (chr^* \bullet \not{n})^* \bullet x \wedge k = \#.x \quad .$$

We modify the specification of *unary* to include *count*:

$$unary \triangleq exb \wedge sht \wedge lng \wedge count \quad .$$

An adequate solution for **establish**(*unary*) must now take into account *count*. We propose the following **establish** statement:

$$\textbf{lub}\ \textbf{establish}(out = \epsilon),\ \textbf{establish}(k = 0)\ \textbf{bul}\ \ ,$$

which is refined by

$$\textbf{par}\ out := \epsilon,\ k := 0\ \textbf{rap}\ \ .$$

Before proceeding with the refinement of **preserve**(*unary*), let us introduce the following notation:

$$words \triangleq \bigcup_{i=1}^{M} txt^i \quad ,$$

$$aw(x,y) \triangleq out' = out \bullet x \bullet y \wedge (x = \not{\!\mathrm{b}} \Rightarrow k' = \#.y) \wedge (x \neq \not{\!\mathrm{b}} \Rightarrow k' = k + \#.x + \#.y) \quad ,$$

$$filled(x) \triangleq k + \#.x \geq M \quad .$$

The refinement of **preserve**(*unary*) consists of the following steps:

```
   $\textbf{preserve}(unary)$
=          { Rule 46 }
   $\textbf{lub}$
      $out = \epsilon \rightarrow \textbf{preserve}(unary)$,
      $out \neq \epsilon \rightarrow \textbf{preserve}(unary)$
   $\textbf{bul}$
⊑          { refining first case }
   $\textbf{lub}$
      $out = \epsilon \rightarrow \textbf{rel}\ \exists w \in words : aw(\epsilon, w)\ \textbf{ler}$,
      $out \neq \epsilon \rightarrow \textbf{preserve}(\text{unary})$
   $\textbf{bul}$
⊑          { refining second case }
   $\textbf{lub}$                                                                (R9)
      $out = \epsilon \rightarrow \textbf{rel}\ \exists w \in words : aw(\epsilon, w)\ \textbf{ler}$,
      $out \neq \epsilon \rightarrow \textbf{rel}\ \exists w \in words : filled(w) \wedge aw(\not{\!\mathrm{n}}, w) \vee \neg filled(w) \wedge aw(\not{\!\mathrm{b}}, w)\ \textbf{ler}$
   $\textbf{bul}$
```

Further refinements to a complete solution in terms of programming constructs are simple. Assuming that we have available a nondeterministic assignment statement " :∈ " defined as

$$x_i :\in E \triangleq \textbf{rel}\ x_i' \in E \wedge \forall j \in 1..n : i \neq j \Rightarrow x_j' = x_j\ \textbf{ler} \quad ,$$

where E is a set-valued term not containing primed free variables, a complete solution to *unary* is:

```
$\textbf{seq}$
   $\textbf{par}\ out := \epsilon,\ k := 0\ \textbf{rap}$,
   $\textbf{clo}$
      $\textbf{var}\ w : \textbf{seq}\ chr$
      $\textbf{seq}$
         $w :\in \bigcup_{i=1}^{M} txt^i$,
         $\textbf{if}\ out = \epsilon\ \textbf{then}\ \textbf{rel}\ aw(\epsilon, w)\ \textbf{ler}$
```

```
            else if $\mathit{filled}(w)$ then rel $aw(\not{\mathrm{n}}, w)$ ler
                else rel $aw(\not{\mathrm{b}}, w)$ ler
        qes
    olc
qes
```

An implementation of a relation of the type

rel $aw(x, w)$ **ler**

is given by

```
par
    $out := out \bullet x \bullet w$,
    if $x = \not{\mathrm{n}}$ then $k := \#.w$
        else $k := k + \#.x + \#.w$
rap .
```

9.1.3.2 Solving Binary

We are targetting an iterative solution for *binary*. Our strategy consists in decomposing *binary* into a format such that one component allows the introduction of a demonic closure. It is not possible to coerce *binary* into a closure, since *binary* is not refined by **skip** (see Proposition 105). Instead, we coerce binary into a **lub** which will then be decomposed into a sequence and a closure.

```
    binary
=           { def. of binary }
    rel $\neg lw.in \wedge sw.in = sw.out'$ ler
⊑           { $\epsilon$ is the identity of $\bullet$ }
    rel $\neg lw.in \wedge sw.in = sw.out' \bullet sw.in' \wedge sw.in' = \epsilon$ ler
```

Let us introduce the following abbreviations which will be used in the sequel:

$$init \triangleq \textbf{rel } \neg lw.in \wedge sw.in = sw.out' \bullet sw.in' \textbf{ ler} \quad ,$$

$$mw \triangleq sw.in \neq \epsilon \quad .$$

Relation *init* corresponds to the first and second terms of the conjunction in the previous refinement. Condition *mw* is the unprimed negation of the third term of the conjunction; it stands for "more words", i.e., *in* contains at least one word. We pursue the refinement of *binary*.

```
⊑           { abbreviations above, Rule 44 }
    lub $init$, establish($\neg mw$) bul
⊑           { coerce $init$ into a sequence, Rule 30 }
    lub
        seq $init$, $init \backslash\!\backslash init$ qes,
        establish($\neg mw$)
    bul
```

Let us introduce another abbreviation: *init* is regular relative to itself (Definition 47), therefore $[\![init \backslash\!\backslash init]\!] = [\![init]\!]\hat{}[\![init]\!]$ (Definition 41, law 48).

$pw \triangleq init \backslash\!\backslash init = \mathbf{rel}\ \neg lw.in \wedge \neg lw.out \wedge sw.out \bullet sw.in = sw.out' \bullet sw.in'\ \mathbf{ler}$

Abbreviation *pw* stands for "preserve word sequence". We pursue the refinement of *binary*.

```
=          { abbreviation pw }
   lub
      seq init, pw qes,
      establish(¬mw)
   bul
=          { coerce pw into a closure, Rule 41 }
   lub
      seq init, clo pw olc qes,
      establish(¬mw)
   bul
```

We are close to a relational expression equivalent to a **while** loop (Section 4.3.11). The next steps will bring us up to that point.

```
=          { Rule 16 }
   lub
      seq init, clo pw olc qes,
      seq establish(true), establish(¬mw) qes
   bul
⊑          { law 4 }
   seq                                                        (R10)
      lub init, establish(true) bul,
      lub clo pw olc, establish(¬mw) bul
   qes
```

The second lub clearly has a structure appropriate for a conversion into a **while**, given some additional refinements. We postpone these refinements to the next paragraph.

9.1.3.3 Merging Binary with Unary.

We now merge the solutions of *binary* and *unary*. The process is almost a mechanical application of laws, except for the selection of a progressively finite relation in the second-to-last step.

```
   lub binary, unary bul
⊑          { refinements of unary (R8, page 116) and binary (R10, page 119) }
   lub
      seq
         establish(unary),
         clo preserve(unary) olc
      qes,
      seq
         lub init, establish(true) bul,
         lub clo pw olc, establish(¬mw) bul
      qes
   bul
⊑          { law 4 }
```

```
seq
    lub establish($unary$), $init$, establish(true) bul,
    lub
        clo preserve($unary$) olc,
        clo $pw$ olc,
        establish($\neg mw$)
    bul
qes
```

The first component of the sequence in the last step has a simple refinement:

rel $out' = \epsilon \wedge in' = in$ **ler** .

We pursue the refinement for the second component of the sequence.

```
    lub
        clo preserve($unary$) olc,
        clo $pw$ olc,
        establish($\neg mw$)
    bul
$\sqsubseteq$          { associativity of lub, Rule 5 }
    lub
        clo lub preserve($unary$), $pw$ bul olc,
        establish($\neg mw$)
    bul
$\sqsubseteq$          { Rule 13 }
    lub
        clo
            if $mw$ then lub preserve($unary$), $pw$ bul
        olc,
        establish($\neg mw$)
    bul
```

We introduce the following abbreviations.

$pf \triangleq$ **rel** $in' = rw.in$ **ler**

$body \triangleq$ **lub preserve**($unary$), pw, pf **bul**

Pursuing refinement, we obtain

```
$\sqsubseteq$          { definition of $body$ }
    lub
        clo
            if $mw$ then $body$
        olc,
        establish($\neg mw$)
    bul
```

Since $[\![mw]\!] = [\![pf]\!]L$, and $[\![pf]\!]$ is progressively finite, it follows from Proposition 73 that the statement above can be transformed into a **while** statement, according to the equivalence stated in Section 4.3.10.4 (page 63). After transformation, we obtain the following.

```
=
  while $mw$ do
    $body$
=       { Rule 27 }
  while $mw$ do
    $mw \rightarrow body$
```

Let us now pause for a moment and see how the merge was done. The solutions of *unary* and *binary* share the same structure, therefore it is possible to push the join inside the structure. The result is a sequence of two joins. The first one being quite easy to refine, we focus our attention on the second component. Its structure is close to a pattern transformable into a **while** loop. We apply two rules to refine it into a pattern equivalent to a **while-do**. Finally, the loop condition is propagated inside the loop body, to ease further refinements.

9.1.3.4 Solving the Loop Body

Each component in the specification of the loop body seems very close to an assignment statement. For instance, component *pf* can be refined by the statement $in := rw.in$, and component *pw* is partly solved by the statement $out := out \bullet fw.in$, etc. We will make use of the sequence coercion rules to compute the sequence of assignement statements solving the loop body. We start with component *pf*. Because it refers to variable *in* only, *pf* is not affected by statements modifying other variables; hence it can be done safely as the last statement. We decide to refine *pf* by the statement $in := rw.in$. For the sake of clarity, we perform calculations without the prerestriction by *mw*, knowing that we can have recourse to it when it is necessary (for proving refinement).

```
  $body$
=       { definition of $body$ }
  lub preserve($unary$), $pw$, $pf$ bul
⊑       { Rule 34 }
  lub
    seq $pf /\!/ (in := rw.in)$, $in := rw.in$ qes,
    preserve($unary$), $pw$
  bul
⊑       { Rule 33 }
  lub
    seq $pf /\!/ (in := rw.in)$, $in := rw.in$ qes,
    seq preserve($unary$)$/\!/(in := rw.in)$, $in := rw.in$ qes,
    seq $pw /\!/ (in := rw.in)$, $in := rw.in$ qes,
  bul
=       { compute demonic left residues, Rule 36 }
  lub
    seq $pf[in' \backslash rw.in']$, $in := rw.in$ qes,
    seq preserve($unary$)$[in' \backslash rw.in']$, $in := rw.in$ qes,
    seq $pw[in' \backslash rw.in']$, $in := rw.in$ qes,
  bul
⊑       { push lub: Rule 4 }
  seq
```

```
    lub $pf[in'\backslash rw.in']$, preserve$(unary)[in'\backslash rw.in']$, $pw[in'\backslash rw.in']$ bul,
    $in := rw.in$
  qes
```

We now refine the **lub** inside the sequence, which we denote by $body_1$. Analyzing the **lub** component $pw[in'\backslash rw.in']$ given by

$$\textbf{rel}\ \neg lw.in \wedge \neg lw.out \wedge sw.out \bullet sw.in = sw.out' \bullet sw.rw.in'\ \textbf{ler} \quad ,$$

or , by using the equality $sw.in = sw.fw.in \bullet sw.rw.in$,

$$\textbf{rel}\ \neg lw.in \wedge \neg lw.out \wedge sw.out \bullet sw.fw.in \bullet sw.rw.in = sw.out' \bullet sw.rw.in'\ \textbf{ler} \quad ,$$

we find that the assignment statement $out := out \bullet fw.in$ is a simple refinement. Going through the same steps as we did for the refinement of the previous **lub**, we obtain:

```
  $body_1$
$\sqsubseteq$
  seq                                                              (R11)
    lub
      $pf[in'\backslash rw.in'][out'\backslash out' \bullet fw.in']$,
      preserve$(unary)[in'\backslash rw.in'][out'\backslash out' \bullet fw.in']$,
      $pw[in'\backslash rw.in'][out'\backslash out' \bullet fw.in']$
    bul,
    $out := out \bullet fw.in$
  qes .
```

An analysis of the component

$$\textbf{preserve}(unary)[in'\backslash rw.in'][out'\backslash out' \bullet fw.in']$$

reveals that it is rather difficult to solve: when variable *in* starts with a *long* word, this component must terminate in a state such that in' starts with a *short* word, in order to satisfy predicate *sht* of *unary*; therefore a valid refinement must modify variable *in*, whereas it seems possible to solve the component by only modifying *out*. A solution is to backtrack one step, and to proceed with a case analysis.

```
  $body_1$
=          { case analysis: Rule 46 }
  lub
    $lfw.in \rightarrow body_1$
    $\neg lfw.in \rightarrow body_1$
  bul
```

After applying Rule 25, one readily finds that **skip** is a valid refinement of the first case. The second case is solved by refinement R11 (page 122) shown previously. What is left now is the refinement of the following specification:

```
$\neg lfw.in \rightarrow$
  lub
    $pf[in'\backslash rw.in'][out'\backslash out' \bullet fw.in']$,
    preserve$(unary)[in'\backslash rw.in'][out'\backslash out' \bullet fw.in']$,
    $pw[in'\backslash rw.in'][out'\backslash out' \bullet fw.in']$
  bul
```

We face a problem similar to the one encountered in Section 9.1.3.1 when proposing a complete solution for *unary*. To solve component

$$\neg \mathit{lfw.in} \rightarrow \mathbf{preserve}(\mathit{unary})[\mathit{in}' \backslash \mathit{rw.in}'][\mathit{out}' \backslash \mathit{out}' \bullet \mathit{fw.in}']$$

we must add a new variable k; we modify the definition of *unary* to include the condition *count* which requires k to be equal to the length of the last line of *out* (see formal definition in Section 9.1.3.1). The refinements devised so far are still valid for the new version of *unary*. A careful analysis of the **preserve** component exhibits that the following specification is a solution for each component.

```
if $out = \epsilon$ then
    $k := \#.fw.in$
else if $filled(fw.in)$ then
    par $k := \#.fw.in$, $out := out \bullet \not{\mathrm{nl}}$ rap
else
    par $k := k + \#.fw.in + 1$, $out := out \bullet \not{\mathrm{b}}$ rap
```

If we piece all the parts together, we obtain the following program:

```
seq
    par $out := \epsilon$, $k := 0$ rap,
    while $mw$ do
        seq
            if $lfw.in$ then skip
            else
                seq
                    if $out = \epsilon$ then
                        $k := \#.fw.in$
                    else if $filled(fw.in)$ then
                        par $k := \#.fw.in$, $out := out \bullet \not{\mathrm{nl}}$ rap
                    else
                        par $k := k + \#.fw.in + 1$, $out := out \bullet \not{\mathrm{b}}$ rap,
                    $out := out \bullet fw.in$,
                qes,
            $in := rw.in$
        qes
qes .
```

9.1.3.5 Other Solutions

We may also consider other solutions to the Naur problem. Our paradigm is not restrictive with this respect. For instance, it is possible to envisage another loop condition. At refinement step R10 (page 119), we may pursue further the refinement of *binary* before combining it with *unary*. In summary, it consists in refining *binary* into a **while** statement, using the extended **while** equivalence (page 71), and to make the **while** specification total using Rule 28. The structure is then converted back into a **lub-clo-establish** structure, and combined with *unary*. The resulting loop condition is:

while $mw \wedge \neg \mathit{lfw.in}$ **do**…

We obtain the same specification for the loop body; hence the refinement of the previous section is still applicable. Using Rule 27 and the definition of **if**, we can simplify the loop body by removing the top-level **if** statement. We obtain the following solution to the Naur problem:

```
seq
    par $out := \epsilon$, $k := 0$ rap,
    while $mw \wedge \neg lfw.in$ do
        seq
            if $out = \epsilon$ then
                $k := \#.fw.in$
            else if $filled(fw.in)$ then
                par $k := \#.fw.in$, $out := out \bullet \not{\mathrm{n}}$ rap
            else
                par $k := k + \#.fw.in + 1$, $out := out \bullet \not{\mathrm{b}}$ rap,
            $out := out \bullet fw.in$,
            $in := rw.in$
        qes
qes .
```

If we provide an appropriate implementation of the type sequence using external files in Pascal, our solution is comparable to the solution of Myers in [72], both in terms of length and algorithmic complexity. Myers' solution is 63 lines long (in PL/I), and $O(\#.in)$ in complexity, versus 100 lines and $O(\#.in)$ for ours. The difference in length is explained by the fact that we have an additional layer of abstraction by implementing a sequence data type, whereas Myers is directly using files and global variables for composing his solution.

9.2 Software Modification

We use the aggregate of programming notation and refinement rules that we have introduced in Chapters 4, 5, 7 and 8 to modify the *sort* specification of Chapter 6, and the compiler example of Chapter 7.

9.2.1 Software Adaptation

The next example illustrates how existing refinements can be reused for carrying out a modification. We address the problem of adding a new functional feature to an existing program. As we stated it in Section 7.1 of Chapter 7, the **lub** construct provides a simple way of formulating this problem. Given a program whose specification is *Pgm*, and given a feature $Feature$ we wish to add to *Pgm*, we formulate the modification as the refinement of the following Utica specification:

lub *Pgm*, $Feature$ **bul** .

Integration of the new feature proceeds by stepwise decomposition in conformance with the structure of *Pgm*, and then by merging the components using the rules for pushing **lub**'s deeper into the nesting structure (Rule 4, Rule 5, etc).

We use the sort specification *sort* for illustrating the process. We consider the following problem: modify the sort specification so that the order of duplicate keys in the initial array is the same as in the final array; for instance, sorting the following list of pairs (*key,datum*)

$$[(K_1, D_a), (K_2, D_b), (K_1, D_c)]$$

results in

$$[(K_1, D_a), (K_1, D_c), (K_2, D_b)]$$

instead of

$$[(K_1, D_c), (K_1, D_a), (K_2, D_b)] \enspace .$$

To specify this feature, we need the auxiliary function $dupk(a, k, i)$, which returns a sequence of the duplicates of k in a, from position i to N, in their initial order:

```
$dupk(a, k, i) \triangleq$
    if $i = N \wedge a[i].key = k$ then $a[i]$
    if $i = N \wedge a[i].key \neq k$ then $\epsilon$
    if $i \neq N \wedge a[i].key = k$ then $a[i] \bullet dupk(a, k, i+1)$
    if $i \neq N \wedge a[i].key \neq k$ then $dupk(a, k, i+1)$   .
```

The specification of the feature is given by

$$pdup \triangleq \textbf{rel}\ \forall k : dupk(a, k, 1) = dupk(a', k, 1)\ \textbf{ler} \quad .$$

The specification of the modification to *sort* is given by

$$sort' \triangleq \textbf{lub}\ sort,\ pdup\ \textbf{bul}$$

The solution found for sort in Chapter 6 on page 91 is

```
**var** $i : 0..N$
**seq**
    **lub clo** $swap$ **olc**, **establish**$(pst)$ **bul**,
    **while** $i \neq N$ **do**
        **lub** $swap$, **preserve**$(pst)$, $pf$ **bul**
**qes** .
```

Let use the following abbreviations.

$$init \triangleq \textbf{lub clo}\ swap\ \textbf{olc},\ \textbf{establish}(pst)\ \textbf{bul},$$

$$loop \triangleq \textbf{while}\ i \neq N\ \textbf{do lub}\ swap,\ \textbf{preserve}(pst),\ pf\ \textbf{bul}$$

The refinement of $sort'$ proceeds as follows:

```
    **lub** $sort$, $pdup$ **bul**
$\sqsubseteq$            { refinement of $sort$ }
    **lub**
        **seq** $init$, $loop$ **qes**,
        $pdup$
    **bul**
$\sqsubseteq$            { coerce $pdup$ into a sequence: Rule 31 }
    **lub**
        **seq** $init$, $loop$ **qes**,
        **seq** $pdup /\!\!/ pdup$, $pdup$ **qes**
     bul
 ⊑          { $pdup$ is a PER: hence $pdup /\!\!/ pdup = pdup$, by Proposition 55 }
     lub
        seq $init$, $loop$ qes,
        seq $pdup$, $pdup$ qes
     bul
 ⊑          { push lub: Rule 4 }
     seq
        lub $init$, $pdup$ bul,
        lub $loop$, $pdup$ bul
     qes
```

The solution found for *init* in the *sort* specification is the assignement statement $i := 0$. This statement also refines *pdup*. Therefore, we find the following refinement:

```
    seq
       $i := 0$,
       lub $loop$, $pdup$ bul
    qes .
```

To combine *loop* with *pdup*, we convert the **while** statement into a **lub-clo-establish** structure. Since $[\![i \neq N]\!] = [\![i' > i]\!]L$, and $[\![i' > i]\!]$ is progressively finite, it follows from Proposition 73 that the conditions for conversions (Section 4.3.10.4, page 63) are satisfied. Converting the **while** statement, we find

```
 lub
    clo if $i \neq N$ then $body$ olc,
    establish($i = N$),
    $pdup$
 bul ,
```

where *body* stands for the body of the **while** statement. There are two options to consider at this point: coerce relation *pdup* into a closure, or into an **establish** statement. The latter is not possible, and the former is feasible, since we already recognized that *pdup* is a PER, and a PER is equal to its own closure (Rule 43). We refine the last statement as follows:

```
    lub
       clo if $i \neq N$ then $body$ olc,
       establish($i = N$),
       clo $pdup$ olc
    bul
⊑          { Rule 13 }
    lub
       clo if $i \neq N$ then $body$ olc,
       establish($i = N$),
       clo if $i \neq N$ then $pdup$ olc
    bul
⊑          { associativity and Rules 5, 8 }
    lub
       clo if $i \neq N$ then lub $body$, $pdup$ bul olc,
        establish($i = N$),
    bul
```

This structure satisfies the pattern for a conversion into a **while** statement. Unfortunately, if we try to discharge the proof obligations for the conversion, we find that the **lub** of *body* and *pdup* is not defined. If we replace *body* by its definition, we find

lub *swap*, **preserve**(*pst*), *pf*, *pdup* **bul** .

For some initial states, these four components do not agree. For instance, in the following state, it is not possible to swap only two cells, and satisfy **preserve**(*pst*), *pf* and *pdup*:

$i = 0$ and $a = [(K_2, D_a), (K_2, D_b), (K_1, D_c)]$.

To satisfy *pf*, we must increase the value of i by at least 1; to satisfy **preserve**(*pst*), we must put (K_1, D_c) in first position, which implies that it is swapped with (K_2, D_a), thereby breaking the initial sequence of elements with key K_2. We have reached a dead end in the refinement process. The only option is to backtrack in the refinement of $sort$, and determine if it is possible to obtain an **lub** with less refined components.

The example showing the inconsistency between the components is also illustrative in other aspects. If multiple swaps are permitted, it is possible to simultaneously satisfy all components: the first swap occurs between (K_2, D_b) and (K_1, D_c), and the second between (K_2, D_a) and (K_1, D_c). This leads us to investigate if component *swap* can be replaced by another allowing multiple swaps. A simple way to achieve this is to replace *swap* by **clo** *swap* **olc**. If we trace the refinements that lead to have *swap* in the loop body, we find refinement R1 (page 90). If we apply Rule 12, which provides that a closure is equal to its own closure, to refinement R1, we obtain the following specification for the loop body:

body$'$ $\triangleq$ **lub clo** *swap* **olc**, **preserve**(*pst*), *pf*, *pdup* **bul** .

This **lub** is defined, and the structure

```
lub
    clo if $i \neq N$ then body$'$ olc,
    establish($i = N$),
bul
```

satisfies the conditions for a conversion into a **while** statement. The solution of *body* in the sort specification does not refine *body*$'$. However, some of the calculations used to derive *body* are still applicable to *body*$'$ (i.e., the calculations pertaining to *pf*, **preserve**(*pst*) and *swap*). If we use the construction-by-parts paradigm to solve *body*$'$, we get the *Bubble Sort* algorithm. We leave out the details since the development is similar to the development of the loop body in *Naur* (for the sequence decomposition) and to the derivation of the main loop in *sort*.

9.2.2 Software Reuse

We revisit the compiler example of Section 7.2 on page 94. To briefly summarize the problem, we consider the case where we have retrieved from a database of software components the closest program that matches the requirements of a new component. We want to determine how the retrieved component can help in solving the new component. Our example uses Pascal compilers. We are looking for an optimizing Pascal compiler, and all we find is an unoptimizing compiler. The problem of modifying *Compiler* to satisfy *OptCompiler* is formulated as the refinement of the following Utica specification:

lub *OptCompiler*, *Compiler* **bul** .

Our objective in this section is to show that some design decisions that we made in this example follow from general rules of structure coercion. The refinement steps R4 (page 94) and R5 (page 94) are driven by Rule 32 and Rule 30. Here is the same refinement done with structure coercion rules. The order of the steps differ from the initial version, because we can now follow a structured approach guided by the coercion rules.

$$
\begin{array}{ll}
 & \textbf{lub}\ \mathit{OptCompiler}, \mathit{Compiler}\ \textbf{bul} \\
\sqsubseteq & \qquad \{\ \text{decompose existing component: Rule 30}\ \} \\
 & \textbf{lub} \\
 & \quad \mathit{OptCompiler}, \\
 & \quad \textbf{seq}\ \mathit{Compiler}, \mathit{Compiler} \backslash\!\backslash \mathit{Compiler}\ \textbf{qes} \\
 & \textbf{bul} \\
\sqsubseteq & \qquad \{\ \text{compute residue: } \mathit{Compiler} \backslash\!\backslash \mathit{Compiler} = \mathit{Preserve}\ \} \\
 & \textbf{lub} \\
 & \quad \mathit{OptCompiler}, \\
 & \quad \textbf{seq}\ \mathit{Compiler}, \mathit{Preserve}\ \textbf{qes} \\
 & \textbf{bul} \\
\sqsubseteq & \qquad \{\ \text{coerce new component: Rule 32}\ \} \\
 & \textbf{lub} \\
 & \quad \textbf{seq}\ \mathit{Compiler}, \mathit{Compiler} \backslash\!\backslash \mathit{OptCompiler}\ \textbf{qes}, \\
 & \quad \textbf{seq}\ \mathit{Compiler}, \mathit{Preserve}\ \textbf{qes} \\
 & \textbf{bul} \\
\sqsubseteq & \qquad \{\ \text{compute residue: } \mathit{Compiler} \backslash\!\backslash \mathit{OptCompiler} = \mathit{Optimizer}\ \} \\
 & \textbf{lub} \\
 & \quad \textbf{seq}\ \mathit{Compiler}, \mathit{Optimizer}\ \textbf{qes}, \\
 & \quad \textbf{seq}\ \mathit{Compiler}, \mathit{Preserve}\ \textbf{qes} \\
 & \textbf{bul}
\end{array}
$$

It is interesting to note that the decompositions in the initial example, which were selected based on our intuition and programming experience, are the same as those provided by the formal rules, which were derived from an abstract analysis of the coercion problem. It illustrates that programming activities are amenable to mathematical modeling, and that the result is not always foreign to the traditional intuition-based or experience-based programming approach.

The fact that guided us in the selection of the rules was that *Compiler* could only be adequately composed or integrated to *OptCompiler* using a sequential composition: An alternation or an iteration clearly seems of no use for that task. The rest was a matter of determining which of the six sequence coercion rules was applicable. This search can be done in a structured manner, and guided by the designer's experience/intuition, which is now applied in a well-defined scope of six rules, rather than at large on an unlimited number of special decompositions that the designer can imagine.

Conclusion

Summary

In this thesis we have presented a method of separation of concerns in the process of constructing a program from a specification. The method proceeds by identifying subspecifications of the specification at hand, and by deriving partially determined programs that are correct with respect to these specifications; then attempting to combine their partial solutions into a program that satisfies all the components simultaneously. The combination proceeds by negotiating a common program structure that all components *agree with*, then casting it as the outer structure and concentrating on the components.

Fortunately, the method is sufficently general to apply not only to the construction of programs, but also to the modification of programs, which comprises software adaptation, software reuse and software merging. One derives the specification of a modification by taking the least upper bound of the existing specification and the new requirements. A modification is carried out by molding the new component into the structure of the existing component. Along this molding process, undefined **lub**'s identify areas where modifications are required, and the idempotence property of the **lub** circumscribes the parts that are preserved.

The mathematics we have developed for our programming paradigm appear to be satisfactory: first, it enables us to produce arbitrarily nondeterministic solutions to the component specifications; second, it faithfully reflects the stepwise negotiation process whereby two component specifications negotiate a common refinement; third, it does show how the refinement of individual components may lead to failure if it is carried out too independently from neighboring components. We have illustrated our method on various examples, where our emphasis is more on illustrating the method than on solving the example.

We have designed a language to support our method of program construction; this language is based on the premise that programs and specifications are structured orthogonally, and that our language must support both kinds of structure in order to smooth out the stepwise transition from specifications to programs. The language has a demonic relational semantics. Some of the demonic operators we use have been studied independently by other researchers (i.e., $\square$ and $\sqcap$ in [9, 11, 12, 23], $\sqcup$, $/\!\!/$ and $\backslash\!\!\backslash$ in [23]). Other operators are new (i.e., $\boxed{*}$, **establish**, **preserve**). We have derived several new properties for each operator.

Finally, we have derived a number of correctness-preserving transformation rules to guide the programmer in the stepwise refinement process. We favored simple, general rules over complex, specialized rules, to keep the number of rules to a manageable size for the designer. Some rules (coercion rules) are optimal, in the sense that they provide least refined decompositions. We do not claim completeness for the set of transformations rules (viz. the ability to derive any correct program for a given specification using exclusively the transformation rules) nor its minimality. Rather, our confidence in their usefulness comes from experimentation.

Assessment

Among the features highlighted by the application of our paradigm to various examples, we mention an interesting aspect of separation of concerns. In most refinement calculi, program refinement steps can be divided into two categories: viz. creative steps, where the programmer must derive an original solution to a problem; and bookkeeping steps, where the programmer must refine his or her solution within the confines of correctness preservation. We have found that our pattern separates these two steps chronologically and by difficulty: the creative steps are all concentrated at the beginning of the refinement process and are applied to subspecifications, rather than to the overall specification (hence are easier); by contrast, the bookkeeping steps, which consist in negotiating common refinements for various components, are done in later stages and are driven by clerical formal manipulations (requiring less creativity). Hence it relieves the programmer from grasping the whole specification in all its complexity at once.

The refinements of the individual components are not totally independent: while refining a component, we must keep an eye on the neighboring components, with a view to remaining consistent with them. This could be viewed as a constraint; we prefer to view it as a means to guide the programmer, in the sense that the structure of one component is hinted by the structure of its neighboring component(s). At least the programmer can concentrate on the solution of one small component at a time, and then determine if it preserves consistency, instead of searching for a solution to a large component.

Refinement by parts generates some overhead compared to traditional refinement, but we find it a reasonable price to pay to reduce the complexity of deriving a solution. Moreover, the applicability of our paradigm to software modification translates into potentially large reuse of refinement steps and proof obligations. This reuse may offset the overhead cost associated with contructing programs by parts.

Finally, even though we have provided another technique for the programmer to tackle complex, large problems, program construction remains a difficult task. Our contribution is a step further in the support of creative decisions, but the journey to the easy construction of provably correct programs from specifications is still a long one.

Connection to Related Work

Program construction by parts is not new: Hehner discusses it in [41], in the case where specifications are represented by predicates. Taking the least upper bound of programs is not new: Hoare *et al* talk about it in [45], where they argue that join is the most useful operator for structuring large specifications. Von Wright [88] defines a language based on join and meet. Gardiner and Morgan [31] use join for data refinement.

Our work differs from Hehner's work by the representation of specifications (predicates vs. relations) and by their interpretations: termination is implicit in our specifications whereas it is expressed as timing constraints in Hehner's. Our work differs from that of Hoare *et al* [45] by using partial relations and demonic operators, by not using a fictitious state to represent nontermination, and by providing rules to eliminate joins in a specification. Our work differs from the work of von Wright, Gardiner and Morgan by using a different semantics (denotational semantics based on relations, versus axiomatic semantics based on predicate transformers), by not allowing miraculous specifications, and by studying the transformation of join-structured specifications. We share with the work of Sekerinski [83] (and the work on Z [78, 84]) the same specification model where the focus is on input-output pairs for which a program must terminate.

Future Research

Our plan for future research includes two directions: on the practical side, we wish to experiment further with the proposed method, by attempting increasingly complex and increasingly large examples. On the theoretical side, we wish to investigate other models to make the **lub** construct defined for any specifications, not only consistent ones. This may be achieved by introducing artificial elements in our algebra, and use them to represent the join of two inconsistent specifications. The impact of these new elements must be carefully investigated. The benefits would be a more elegant formulation of transformation rules (no restrictions on undefined **lub**'s), and potentially streamlined refinement paths. Also, we wish to pursue the study of structure coercion, to provide heuristics for the general sequence coercion problem, and for the closure coercion problem. Finally, we wish to study the extension of our calculus to handle communication and concurrency. It will be interesting to see if the construction by parts paradigm gives the same leverage for this more complex class of systems.

Bibliography

[1] Abrial, J.R.: The Mathematical Construction of a Program. *Science of Computer Programming* **4** (1984) 45–86.

[2] Abrial, J.R.: A Formal Approach to Large Software Construction. In *Mathematics of Program Construction: First International Conference*, J.L.A. van de Snepscheut, ed., LNCS 375, Springer-Verlag, 1989.

[3] Abrial, J.R.: B-Technology Technical Overview. In *Fifth International Conference on Formal Description Techniques*, M. Diaz and R. Groz, eds, Lannion, France, 1992.

[4] Alikacem, A., S.B.M. Sghaier, J. Desharnais, M. El Ouali, and F. Tchier: From Demonic Semantics to Loop Construction: A Relation Algebraic Approach. In *3rd Maghrebian Conference on Software Engineering and Artificial Intelligence*, April 11–14, Rabat, Morocco, 1994, 239–248.

[5] Back, R.J.R.: On the Correctness of Refinement in Program Development. Ph. D. Thesis Report A-1978-4, Dept. of Computer Science, University of Helsinki, 1978.

[6] Back, R.J.R.: Correctness Preserving Program Refinements: Proof Theory and Applications. Mathematical Center Tracts 131, Mathematical Centre, Amsterdam, 1980.

[7] Backhouse, R.C.: *Program Construction and Verification.* Prentice Hall, 1986.

[8] Backhouse, R.C. *et al*: A Relational Theory of Data Types. Dept. of Math. and Comp. Sc., Eindhoven University of Technology, Netherlands, 1992.

[9] Backhouse, R.C., J. van der Woude. Demonic Operators and Monotype Factors. *Mathematical Structures in Computer Science* **3**(4) (1993) 417–433.

[10] Basili, V.R., R.W. Selby: Comparing the Effectiveness of Software Testing Strategies. University of Maryland, 1985.

[11] Berghammer, R.: Relational Specification of Data Types and Programs. Bericht Nr. 9109, Fakultät für Informatik, Universität des Bundeswehr München, Germany, September 1991.

[12] Berghammer, R., G. Schmidt: Relational Specifications. In: C. Rauszer (editor), *XXXVIII Banach Center Semesters on Algebraic methods in Logic and their Computer Science Applications*, 1993, 167–190.

[13] Boudriga, N., F. Elloumi, A. Mili: On The Lattice of Specifications: Applications to a Specification Methodology. *Formal Aspects of Computing* **4** (1992) 544–571.

[14] Bourbaki, N.: Sur le Théorème de Zorn. *Arch. Math. (Basel)* **2** (1949/1950) 434–437.

[15] Chin, L.H., A. Tarski: Distributive and Modular Laws in the Arithmetic of Relation Algebras. *University of California Publications* **1** (1951) 341–384.

[16] Davey, B.A., H.A. Priestley: *Introduction to Lattices and Order.* Cambridge University Press, 1990.

[17] DaSilva, C., B. Dehbonci, F. Mejia: Formal Specification in Industrial Applications: Subway Speed Control System. In *Formal Description Techniques, V*, M. Diaz, R. Groz, eds, Elsevier, 1993, 199–213.

[18] de Bakker, J.W., W.P. de Roever: A Calculus for Recursive Program Schemes. In *Automata, Languages and Programming*, M. Nivat, ed., North Holland, 1972, 167–196.

[19] de Bakker, J.W.: Semantics and Termination of Nondeterministic Recursive Programs, In *Automata, Languages and Programming*, S. Michaelson, R. Milner, eds, North Holland, 1976, 435–477.

[20] Desharnais, J.: Abstract Relational Semantics. Ph. D. Dissertation, School of Computer Science, McGill University, Canada, 1989.

[21] Desharnais, J., A. Jaoua, N. Belkhiter, F. Tchier: Data Refinement in a Relation Algebra. In *Second Maghrebine Conference on Software Engineering and Artificial Intelligence*, Tunis, April 13–16, 1992, 222–236.

[22] J. Desharnais, A. Jaoua, F. Mili, N. Boudriga, A. Mili: A Relational Division Operator: The Conjugate Kernel. *Theoretical Computer Science* **114** (1993) 247–272.

[23] Desharnais, J. *et al*: Embedding a Demonic Semilattice in a Relation Algebra. *Theoretical Computer Science* (1995), to appear.

[24] Dijkstra, E.W.: Notes on Structured Programming. In *Structured Programming*, O-J Dahl, C.A.R. Hoare, eds, Academic Press, 1971.

[25] Dijkstra, E.W.: Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Communications of the ACM* **18**(8) (1975) 453–457.

[26] Dijkstra, E.W.: *A Discipline of Programming.* Prentice Hall, 1976.

[27] Dijkstra, E.W., W.H.J. Feijen: *A Method of Programming.* Addison-Wesley, 1988.

[28] Dijkstra, E.W., C.S. Scholten: *Predicate Calculus and Program Semantics.* Springer Verlag, 1990.

[29] Dijkstra, R.M.: Relational Calculus and Relational Program Semantics. Computing Science Reports, CS-R9408, Department of Computing Science, University of Groningen, 1994.

[30] Floyd, R.W.: Assigning Meanings to Programs. In *Proc. American Mathematical Society Symposia in Applied Mathematics* **19** (1967) 19–32.

[31] Gardiner, P., C.C. Morgan: Data Refinement of Predicate Transformers. *Theoretical Computer Science* **87** (1991) 143–162.

[32] Goodenough, J.B., S. Gerhart: Towards a Theory of Test Data Selection Criteria. In *Current trends in Programming Methodology, Vol. 2*, R.T. Yeh, ed, Prentice-Hall, 1977.

[33] Gries, D.: *The Science of Programming.* Springer Verlag, 1981.

[34] Gritzner, T.F., R. Berghammer: A Relation Algebraic Model of Robust Correctness. Bericht Mr 9301, Universität des Bundeswehr München, fakultät für Informatik, Neubiberg, Germany, January 1993.

[35] Gunter, C.A., D.S. Scott: Semantic Domains. In *Handbook of Theoretical Computer Science, Vol. B*, J. van Leeuwen, ed., North-Holland, Amsterdam, 1990, 635–674.

[36] Hehner, E.C.R., C.A.R. Hoare: A More Complete Model of Communicating Processes. *Theoretical Computer Science* **26** (1983) 105–120.

[37] Hehner, E.C.R.: Predicative Programming. *Communications of the ACM* **27**(2) (1984) 134–151.

[38] Hehner, E.C.R.: *The Logic of Programming.* Prentice Hall, London, UK, 1986.

[39] Hehner, E.C.R., A. J. Malton: Termination Conventions and Comparative Semantics. *Acta Informatica* **25** (1988) 1–14.

[40] Hehner, E.C.R.: A Practical Theory of Programming. *Science of Computer Programming* **14** (1990) 133–158.

[41] Hehner, E.C.R.: *A Practical Theory of Programming.* Springer Verlag, 1993.

[42] Hoare, C.A.R.: An Axiomatic Basis of Computer Programming. *Communications of the ACM* **12**(10) (1969) 576–580, 583.

[43] Hoare, C.A.R.: Proof of Correctness of Data Representations. *Acta Informatica* **1** (1972) 271–281.

[44] Hoare, C.A.R., H. Jifeng: The Weakest Prespecification, part I and II. *Fundamenta Informaticae* **IX** (1986) 51–84, 217–252.

[45] Hoare, C.A.R. I.J. Hayes, H. Jifeng, C.C. Morgan, A.W. Roscoe, J.W. Sanders, I.H. Sorenson, J.M. Spivey, B.A. Sufrin: Laws of Programming. *Communications of the ACM* **30**(8) (1987) 672–686.

[46] Jaoua, A. N. Boudriga, J-L. Durieux, A. Mili: Pseudo-invertibility, a Measure of Regularity of Relations. *Theoretical Computer Science* **19**(2) (1991) 323–339.

[47] Jones, C.B.: *Software Development: A Rigorous Approach.* Prentice-Hall, 1980.

[48] Jones, G., M. Sheeran: Designing Arithmetic Circuits by Refinement in Ruby. In *Mathematics of program construction: second international conference*, R.S. Bird, C.C. Morgan, J.C.P. Woodcock, eds, Oxford, 1992, LNCS 669, Springer-Verlag, 1993.

[49] Litteck, H.J., P.J.L. Wallis: Refinement Methods and Refinement Calculi. *Software Engineering Journal* **7**(3) (1992) 219–229.

[50] Linger, R.C., H.D. Mills, B.I. Witt: *Structured programming: Theory and Practice.* Addison Wesley, 1979.

[51] Liskov, B., V. Berzins: An Appraisal of Program Specifications. In *Research Directions in Software Technology*, P. Wegner, ed., MIT Press, 1977.

[52] Maddux, R.D.: The Origin of Relation Algebras in the Development and Axiomatization of the Calculus of Relations, *Studia Logica* **50**, (1991) 421–455.

[53] Manna, Z.: *Mathematical Theory of Computation.* McGraw Hill, 1974.

[54] Marchowsky, G.: Chain-complete Posets and Directed Sets with Applications, *Algebra Universalis* **6** (1976) 53–68.

[55] Meyer, B.: On Formalism in Specifications. *IEEE Software* **2**(1) (1985) 6–27.

[56] Meyer, B.: On Oversights in Specifications. *IEEE Software* **2**(4) (1985) 2.

[57] Mili, A.: A Relational Approach to the Design of Deterministic Programs. *Acta Informatica* **20** (1983) 315–328.

[58] Mili, A., Y. Qing, W.X. Yang. A Relational Specification Methodology. *Software-Practice and Experience* **16**(11) (1986) 1003–1030.

[59] Mili, A. J. Desharnais, F. Mili: Relational Heuristics for the Design of Deterministic Programs. *Acta Informatica* **24** (1987) 239–276.

[60] Mili, A, N. Boudriga, F. Mili: *Towards a Discipline of Structured Specifying.* Ellis Horwood, 1989.

[61] Mili, F., A. Mili: Heuristics for Constructing While Loops. *Science of Computer Programming* **18** (1992) 67–106.

[62] Mili, A., R. Mili, R. Mittermeir: Storing and Retrieving Software Components: A Refinement-Based System. In *Proceedings of the Sixteenth International Conference on Software Engineeering*, Sorrento, Italy, 1994.

[63] Mili, A., F. Mili, J. Desharnais: *Computer Program Construction.* Oxford University Press, 1994.

[64] Mills, H.D.: The New Math of Computer Programming. *Communications of the ACM* **18**(1) (1975) 43–48. *Corrigendum* **18**(5) (1975) 280.

[65] Mills, H.D., V.R. Basili, J.D. Gannon, R.G. Hamlet: *Principles of Computer Programming: A Mathematical Approach.* Allyn and Bacon, 1987.

[66] Möller, B.: Relations as a Program Development Language. In *Constructing Programs from Specifications*, Möller, B, ed., Proc. IFIP TC 2/WG 2.1 Pacific Grove, CA, USA, May 13–16, North-Holland (1991) 373–397.

[67] Morgan, C.: The Specification Statement. *ACM Transactions on Programming Languages and Systems* **11**(4) (1988) 517–561.

[68] Morgan, C.: *Programming from Specifications.* Prentice Hall, 1990.

[69] Morris, J.M.: A Theoretical Basis for Stepwise Refinmement and the Programming Calculus. *Science of Computer Programming* **9** (1987) 287–306.

[70] Morris, J.M.: Laws of Data Refinement. *Acta Informatica* **26** (1989) 287–308.

[71] Mosses, P.D.: Denotational Semantics. In *Handbook of Theoretical Computer Science, Vol. B*, J. van Leeuwen, ed., North-Holland, 1990.

[72] Myers, G.J.: A Controlled Experiment in Program Testing and Code Walkthroughs / Inspections. *CACM* **21**(9) (1978) 760–768.

[73] Naur, P.: Programming by Action Clusters. *BIT* **9**(3) (1969) 250–258.

[74] Nelson, G.: A Generalization of Disjktra's Calculus. *ACM Transactions on Programming Languages and Systems* **11**(4) (1989) 517–560.

[75] Nguyen, T.T.: A Relational Model of Demonic Nondeterministic Programs. *International Journal of Foundations of Computer Science* **2**(2) (1991) 101–131.

[76] Parnas, D.L.: On the Criteria to be Used in Decomposing Systems into Modules. *Communications of the ACM* **15**(12) (1972).

[77] Parnas, D.L.: A Generalized Control Structure and Its Formal Definition. *Communications of the ACM* **26**(8) (1983) 572–581.

[78] Potter, B., J. Sinclair, D. Till: *An Introduction to Formal Specification and Z.* Prentice-Hall, 1991.

[79] Pratt, V.R.: Origins of the Calculus of Binary Relations. In *Proc. 7th Annual IEEE Symp. on Logic in Computer Science*, Santa Cruz, CA, USA, 1992, 248-254.

[80] Riguet, J.: Relations Binaires, Fermetures, Correspondances de Galois. *Bulletin de la Société de Mathématiques de France* **76** (1948) 114–155.

[81] Schmidt, G., T. Ströhlein: Relation Algebras: Concept of Point and Representability. *Discrete Mathematics* **54**(1) (1985) 83–92.

[82] Schmidt, G., T. Ströhlein: *Relations and Graphs.* Springer-Verlag, 1993.

[83] Sekerinski, E.: A Calculus for Predicative Programming. In *Mathematics of program construction: second international conference*, R.S. Bird, C.C. Morgan, J.C.P. Woodcock, eds, Oxford, 1992, LNCS 669, Springer-Verlag, 1993.

[84] Spivey, M.: *Understanding Z.* Cambridge University Press, 1988.

[85] Sufrin, B.: Formal System Specification: Notations and Examples. *Lecture Notes in Computer Science*, 1981.

[86] Tarski, A.: On the Calculus of Relations. *Journal of Symbolic Logic* **6**(3) (1941) 73–89.

[87] Tarski, A.: A Lattice-Theoretical Fixpoint Theorem and its Applications. *Pacific Journal of Mathematics* **5** (1955) 285–309.

[88] Von Wright, J.: A Lattice Theoretical Basis for Program Refinement. Ph. D. Thesis, Dept. of Computer Science, Åbo Akademi, Finland, 1990.

[89] Wirth, N.: Program Development by Stepwise Refinement. *Communications of the ACM.* **14**(4) (1971) 221–227.

[90] Woodcock, J.: *Software Engineering Mathematics.* Addison Wesley, 1990.