A REFINEMENT CALCULUS FOR Z

by

Ana Cavalcanti

Technical Monograph PRG-123
ISBN 0-902928-97-X

August 1997

Oxford University Computing Laboratory
Programming Research Group
Wolfson Building, Parks Road
Oxford OX1 3QD
England

Oxford University Computing Laboratory
Programming Research Group
Wolfson Building, Parks Road
Oxford OX1 3QD
England

A REFINEMENT CALCULUS FOR Z

Ana Cavalcanti

Wolfson College

Thesis submitted for the Degree of Doctor of Philosophy
at the University of Oxford

Hilary Term, 1997

Abstract

The lack of a method for developing programs from Z specifications is a difficulty that is now widely recognised. As a contribution to solving this problem, we present ZRC, a refinement calculus based on Morgan's work that incorporates the Z notation and follows its style and conventions. Other refinement techniques have been proposed for Z; ZRC builds upon some of them, but distinguishes itself in that it is completely formalised.

As several other refinement techniques, ZRC is formalised in terms of weakest preconditions. In order to define the semantics of its language, ZRC-L, we construct a weakest precondition semantics for Z based on a relational semantics proposed by the Z standards panel. The resulting definition is not unexpected, but its construction provides evidence for its suitability and, additionally, establishes connections between predicate transformers and two different relational models. The weakest precondition semantics of the remaining constructs of ZRC-L justify several assumptions that permeate the formalisation of Morgan's refinement calculus. Based on the semantics of ZRC-L, we derive all laws of ZRC.

Typically the refinement of a schema in ZRC begins with the application of a conversion law that translates it to a notation convenient for refinement, and proceeds with the application of refinement laws. The conversion laws of ZRC formalise the main strategies and rules of translation available in the literature; its set of refinement laws is extensive and includes support for procedures, parameters, recursion, and data refinement.

Morgan and Back have proposed different formalisations of procedures and parameters in the context of refinement techniques. We investigate a surprising and intricate relationship between these works and the substitution operator that renames the free variables of a program, and reveal an inconsistency in Morgan's calculus. Back's approach does not suffer from this inconsistency, but he does not present refinement laws. We benefit from both works and use a model based on Back's formalism to derive refinement laws similar to those in Morgan's calculus. Furthermore, we derive additional laws that formalise Morgan's approach to recursion.

Three case studies illustrate the application of ZRC. They show that ZRC can be useful as a technique of formal program development, but are by no means enough to ascertain the general adequacy of its conversion and refinement laws. Actually, since Z does not enforce a specific style of structuring specifications, it is likely that new laws will be proved useful for particular system specifications: two of our case studies exemplify this situation. Our hope is that ZRC and its formalisation will encourage further investigation into the refinement of Z specifications and the proper justification of any emerging strategies or techniques.

To Carlos,
my parents, and
my aunt.

Acknowledgments

I would like to express my gratitude to my supervisor, Jim Woodcock, for many valuable discussions and suggestions, and for his expert advice and guidance. He has been a major source of inspiration and encouragement.

My examiners, Paul Gardiner and Bernard Sufrin, have contributed valuable suggestions and corrections. The thesis has improved a lot due to their comments.

Special thanks are due to Will Harwood, Steve King, Carroll Morgan, and Augusto Sampaio, who have read and commented on drafts of parts of this thesis. Will Harwood has suggested the study of the relationship between weakest preconditions and relations in a general framework, rather than in the context of Z. Carroll Morgan has called to my attention the existence of a relationship between positively conjunctive weakest preconditions and relations. The parts of this thesis concerned with procedures and parameters are the result of joint work with Augusto Sampaio. Our many discussions with Carroll Morgan on this subject have been of great help; anonymous referees have also commented on this work.

Thanks to Paulo Borba for many helpful discussions during earlier stages of my work. Many thanks also to Paul Gardiner who has answered my many questions about recursion, and to Andrew Martin and Andrew Stevens for patiently explaining their work to me.

I am also grateful to some members of the Wolfson College staff for their friendship and support in difficult moments. Thanks to Roger Hall, Angela Moore, Jane Scriven, and Heather Waring.

My deepest gratitude goes to my husband for his caring love and dedication. He has left his family, his friends, and his job to come to England with me. I will never forget that.

My parents and my aunt have suffered a great deal from my absence, but never failed to show their support and encouragement. I will never be able to properly acknowledge everything they have done for me.

I am lucky enough to be able to say that I could not possibly name all my friends in this space. Their practical and emotional support has been an invaluable asset. To all of them, a big thanks.

The work described in this thesis has been financially supported by the Brazilian government through one of its research agencies, CNPq, grant 204.527/90-2.

Contents

Chapter 1

Introduction

Z [58, 8] is a well-established formal specification language that has a distinguishing mechanism of modularisation: the schema calculus. Its success is evident: many case studies [25, 27] have already been developed, some of which involve industrial applications; a wide range of tools [57, 31, 24] that support several aspects of its use have been implemented; and several courses and textbooks are at our disposal [52, 16, 65]. In spite of all this, a drawback has been recognised in the use of Z: the absence of a well-defined and provably correct method of moving from the specification phase to the later stages of program development.

Several proposals for solving this problem can be found in the literature. Some results have been achieved on methods of prototyping Z specifications [17, 13, 29, 54] which focus on the production of low cost prototypes with the aim of capturing requirements and validating specifications. In all cases, the prototypes can be generated mainly by translating restricted forms of specifications.

Another line of research considers the use of refinement techniques to develop implementations for Z specifications. As opposed to prototyping methods, a refinement technique aims at the production of efficient imperative programs. Its major features are a unified language of specification, design and programming, a refinement relation, and a stepwise style of program development. In the particular case of a refinement calculus, refinement laws are used to derive programs from specifications.

King proposes in [34] the combined use of Z and Morgan's refinement calculus. In this work the differences between Z and the notation of Morgan's calculus [45] are analysed and, in the light of these considerations, laws that translate schemas and some schema expressions to programs of this refinement calculus are suggested. In [64], Woodcock indicates one additional translation law: a form of promotion is implemented as a call to a procedure with a value-result parameter.

In [66] Wordsworth proposes a refinement technique for Z where schemas themselves are regarded as commands. In this work, refinement is accomplished either by laws that are similar to the translation rules of [34] that apply to schema expressions, or by verification instead of calculation, when none of these laws apply. There is no equivalent to the law of [34] that can translate every schema.

Other proposals that present characteristics of both prototyping and refinement techniques, or that present just some characteristics of one of these approaches, are also available. The objective of the work presented in [61], for instance, is the definition of a method for implementing Z specifications using a functional language. In this case, an executable subset of Z (Z^{--}) is identified, and refinement occurrs in a unified framework for specification and programming. Nevertheless, the exemplified refinement does not follow any formal techniqne and it is hoped that the Z^{--} programs can be employed as final products or that comparison of test results can be used in validating an eventual translation to another programming language.

A notation for documenting the development of Ada programs from Z specifications is defined in [56]. Despite the fact that a language similar to that of Morgan's refinement calculus is used, the proposal consists of designing the programs directly in Ada and then providing an account of their correctness using the notation and literate programming. Refinement laws are not used.

The objectives of producing efficient programs, and of applying a development method that is mathematically sound and allows the use of calculational techniques are best served by the approaches in [34, 66, 64, 65]. In particular, the proposals in [34, 64, 65] distinguish themselves by encouraging and enabling the construction of programs by calculation, instead of verification, to a much greater extent.

In this work, we present ZRC, a refinement calculus for Z whose design is based mainly on [34, 64, 65] and Morgan's calculus. Following the lines of [64, 65], ZRC employs a notation compatible with the Z style at all stages of development. Most of its laws are based on those of [34, 64, 65, 45], but we consider extensions and adaptations, and introduce new laws in order to comply with the Z notation and facilitate the application of ZRC. As Morgan's calculus, ZRC provides support for procedures, recursion, and data refinement. We prove the soundness of all its laws and this is probably its most remarkable feature. As far as we know, no attempt has been made to formalise [34, 64, 65]. Our work uncovers a few mistakes, and indicates simplifications and generalisations. We also point out an inconsistency in [41].

At the moment, there is an effort to standardise Z, and a fairly complete account of the language has already been given in [8]. We assume familiarity with Z as presented in this document, which we use as a basis for our work. A conformant and more accessible presentation can be found in [65].

In the next section we provide an overview of ZRC by means of a very simple example: the birthday book that is specified in [58]. In Section 1.2 we describe the subsequent chapters and appendices.

1.1 A Simple Development

The starting point of a system development in ZRC is, in general, a concrete Z specification which is expressed in terms of data types available in the target programming language. This specification can be, for instance, the result of applying the Z data refinement technique to an abstract specification.

In this section, we apply ZRC to derive an implementation for a birthday book which was first specified and data-refined in [58]. This is a small system that records birthday dates and is able to issue reminders. In what follows, we reproduce its concrete specification as presented in [58].

The birthday book deals with names and dates from the sets $NAME$ and $DATE$, which are introduced as given sets.

$[NAME, DATE]$

People's names and birthday dates are registered in the arrays $names$ and $dates$ which are represented as total functions from $\mathbb{N}_1$; for simplicity, in [58] potentially infinite arrays are assumed

Operation	Precondition
*AddBirthday*1	$\forall i : 1 \mathinner{..} hwm \bullet name? \neq names\ i$
*FindBirthday*1	$\exists i : 1 \mathinner{..} hwm \bullet name? = names\ i$

Table 1.1: Operations Preconditions

to be available in the target programming language. The birthday of the person whose name is recorded at the i-th position of *names* is on the date recorded at the i-th position of *dates*.

*BirthdayBook*1

$$\begin{array}{|l}
names : \mathbb{N}_1 \nrightarrow NAME \\
dates : \mathbb{N}_1 \nrightarrow DATE \\
hwm : \mathbb{N} \\
\hline
\forall i, j : 1 \mathinner{..} hwm \bullet i \neq j \Rightarrow names\ i \neq names\ j
\end{array}$$

The additional state component *hwm* determines the portion of *names* and *dates* that is in use: the positions from 1 to *hwm*. The state invariant establishes that no name is recorded in these positions more than once.

The birthday book has three operations. The first that we specify adds a person's name and birthday, inputs *name*? and *date*?, to *names* and *dates*.

*AddBirthday*1

$$\begin{array}{|l}
\Delta BirthdayBook1 \\
name? : NAME \\
date? : DATE \\
\hline
\forall i : 1 \mathinner{..} hwm \bullet name? \neq names\ i \\
hwm' = hwm + 1 \\
names' = names \oplus \{hwm' \mapsto name?\} \\
dates' = dates \oplus \{hwm' \mapsto date?\}
\end{array}$$

This is a partial operation: *name*? can be added to *names* only if it is not already recorded there in the positions from 1 to *hwm*. The preconditions of *AddBirthday*1 and *FindBirthday*1, the next operation that we specify, are shown in Table 1.1.

The operation *FindBirthday*1 finds the birthday, output *date*!, of the person called *name*?.

*FindBirthday*1

$$\begin{array}{|l}
\Xi BirthdayBook1 \\
name? : NAME \\
date! : DATE \\
\hline
\exists i : 1 \mathinner{..} hwm \bullet name? = names\ i \land date! = dates\ i
\end{array}$$

This operation can be successfully executed only if *name*? is recorded in *names* (see Table 1.1).

The last operation, *Remind*1, retrieves the names of the people that have their birthday on an input date *today*?. The outputs are an array *cardlist*!, which records these names, and a natural number *ncards!* that identifies the section of *cardlist!* that is being used.

$$
\begin{array}{|l}
\textit{Remind1} \\
\hline
\Xi \textit{BirthdayBook1} \\
\textit{today?} : \textit{DATE} \\
\textit{cardlist!} : \mathbb{N}_1 \rightarrow \textit{NAME} \\
\textit{ncards!} : \mathbb{N} \\
\hline
\{\ i : 1 \mathinner{..} \textit{ncards!} \bullet \textit{cardlist!}\ i\ \} = \{\ j : 1 \mathinner{..} \textit{hwm} \mid \textit{dates}\ j = \textit{today?} \bullet \textit{names}\ j\ \} \\
\hline
\end{array}
$$

As opposed to *AddBirthday*1 and *FindBirthday*1, *Remind*1 is a total operation: its precondition is true.

With the assumption that names, dates, arrays, and natural numbers are available in the target programming language, we start the refinement of the operations. In this stepwise process, intermediary and final programs are written in ZRC-L, the language of ZRC. Besides the Z notation. this language embodies specification and programming constructs typically found in refinement techniques; it is an extension of Dijkstra's language of guarded commands [14], which we assume to be known. Final programs are written with the use of executable constructs only. Translating them to an imperative target language should not be difficult, but is not in the scope of ZRC.

The first step of the birthday book development consists of transforming the schemas that specify operations into specification statements. This change of notation is the concern of the so called conversion laws of ZRC. Here, we use that named *bC* (basic conversion).

A specification statement has the form $w : [pre, post]$, where w, the frame, is a list of variables, and *pre*, the precondition, and *post*, the postcondition, are predicates. This program can change only the value of the variables in w and, when executed from a state and with inputs that satisfy *pre*, terminates in a state and with outputs that satisfy *post*. By applying *bC* to *AddBirthday*1 (and then simplifying the precondition of the resulting specification statement), we obtain the specification statement shown below. For clarity, we stack the conjuncts of its pre and postcondition. and those of many other predicates that follow.

$$
\begin{array}{l}
\textit{AddBirthday}1 \\
= bC
\end{array}
$$

$$
\textit{names}, \textit{dates}, \textit{hwm} : \left[\begin{array}{l}
\left(\begin{array}{l}
\forall i, j : 1 \mathinner{..} \textit{hwm} \bullet i \neq j \Rightarrow \textit{names}\ i \neq \textit{names}\ j \\
\forall i : 1 \mathinner{..} \textit{hwm} \bullet \textit{name?} \neq \textit{names}\ i
\end{array} \right), \\
\left(\begin{array}{l}
\forall i, j : 1 \mathinner{..} \textit{hwm}' \bullet i \neq j \Rightarrow \textit{names}'\ i \neq \textit{names}'\ j \\
\forall i : 1 \mathinner{..} \textit{hwm} \bullet \textit{name?} \neq \textit{names}\ i \\
\textit{hwm}' = \textit{hwm} + 1 \\
\textit{names}' = \textit{names} \oplus \{\textit{hwm}' \mapsto \textit{name?}\} \\
\textit{dates}' = \textit{dates} \oplus \{\textit{hwm}' \mapsto \textit{date?}\}
\end{array} \right)
\end{array} \right] \qquad (i)
$$

As *AddBirthday*1, this program can change the values of the state components. Its precondition includes the state invariant and the precondition of *AddBirthday*1. The postcondition comprises the invariant of the after-state and the predicate of *AddBirthday*1. Altogether, the specification statement (*i*) and *AddBirthday*1 specify the same operation. The advantage of writing *AddBirthday*1 as a specification statement is that, even though they are abstract programs which, as schemas, cannot be executed, specification statements are better suited for refinement.

In ZRC, specifications (schemas, specification statements, and other constructs that we introduce in Chapter 3) and designs, which mix programming and specification constructs, are all regarded as programs. In this more general context, we can say that refinement is a relation between programs. Informally, a program p_2 refines a program p_1 when p_2 is acceptable as an implementation (or design) of p_1.

Refinement laws characterise properties of the refinement relation. The refinement process consists of repeatedly applying them to derive an efficient and executable program from a specification. A conversion law is a refinement law that transforms a specification written in Z into a program of ZRC-L. In general, they are applied at the beginning of the refinement process only.

As an example, we present below the refinement law *assigI* (assignment introduction), which can be used to refine a specification statement to an assignment.

Law *assigI* Assignment introduction

$$w, vl : [pre, post]$$

$\sqsubseteq$ *assigI*

$$vl := el$$

provided $pre \Rightarrow post[el/vl'][\_/']$

Syntactic Restrictions

- vl contains no dnplicated variables;
- vl and el have the same length;
- el is well-scoped and well-typed;
- el has no free dashed variables;
- The corresponding variables of vl and expressions of el have the same type.

The symbol $\sqsubseteq$ represents the refinement relation and, as we mentioned above, *assigI* is the law name. Since the assignment $vl := el$ potentially modifies the variables of vl, they must be in the frame of the specification statement. The proviso ensures that, when the precondition of the specification statement holds, its postcondition is satisfied if the after-state variables assume the values established in $vl := el$. To pnt it more simply, it certifies that this assignment really implements the specification statement. The predicate $post[el/vl'][\_/']$ is tbat obtained by substituting the expressions of el for the corresponding variables of vl' and removing the dashes from the free variables of $post$. The syntactic restrictions guarantee that the assignment is well-formed, well-scoped, and well-typed.

The operation *AddBirthday*1 can be implemented by an assignment whose introduction can be justified by an application of *assigI* to the specification statement (i).

$(i) \sqsubseteq$ *assigI*

$$hwm, names, dates := hwm + 1, names \oplus \{hwm + 1 \mapsto name?\}, dates \oplus \{hwm + 1 \mapsto date?\}$$

As required, hwm, $names$, and $dates$ are in the frame of (i). Also, this list has no repetitions and has the same length as $hwm + 1, names \oplus \{hwm + 1 \mapsto name?\}, dates \oplus \{hwm + 1 \mapsto date?\}$, whose expressions refer only to variables in scope: either state or input variables. Finally, these expressions have the same type as the corresponding variables of $hwm, names, dates$.

As to the proof-obligation ensued by the proviso of *assigI*, we observe that the second conjunct of the postcondition of (i) is in its precondition, and the last three conjuncts are the equalities that characterise the assignment. Therefore, the interesting part of this proof-obligation is prompted by the first conjunct and amounts to showing that the assignment maintains the state invariant. Since the array $names \oplus \{hwm + 1 \mapsto name?\}$ coincides with *names* in all positions except $hwm + 1$, the precondition of (i) (or, more precisely, the state invariant) guarantees that it contains no repetitions in the positions from 1 to *hwm*. Our concern is, therefore, only with the introduction of *name?* in the position $hwm + 1$. The precondition, however, also states that *name?* is not recorded in $(1 \mathrel{..} hwm) \lhd names$, and consequently its insertion does not lead to repetitions.

In general, the refinement of an operation comprises several applications of different laws. Programming constructs are introduced gradually and their components are developed independently. The refinement of *FindBirthday*1 provides an example.

The specification statement obtained by applying the conversion law *bC* to *FindBirthday*1, which provides an alternative definition for this operation, is presented in the sequel.

$$
\begin{array}{l}
\mathit{FindBirthday}1 \\
= bC \\
\quad date! : \left[\begin{array}{l} \left(\begin{array}{l} \forall i, j : 1 \mathrel{..} hwm \bullet i \neq j \Rightarrow names\ i \neq names\ j \\ \exists i : 1 \mathrel{..} hwm \bullet name? = names\ i \end{array} \right), \\ \exists i : 1 \mathrel{..} hwm \bullet name? = names\ i \land date! = dates\ i \end{array} \right]
\end{array} \qquad (ii)
$$

Since *FindBirthday*1 cannot change the state, only the output variable is in the frame of this specification statement. Its precondition includes the state invariant and the precondition of *FindBirthday*1. The postcondition, however, is simply the predicate of *FindBirthday*1: by not changing the state, *FindBirthday*1 trivially maintains its invariant which, therefore, does not need to be enforced in the postcondition of (ii).

This operation can be implemented by an iteration that records in an auxiliary variable the position where *name?* occurs in *names*, followed by a proper assignment to *date!*. The declaration of the auxiliary variable can be introduced by applying the *vrbI* (variable introduction) law to (ii). The resulting program is shown below.

$$
\begin{array}{l}
\sqsubseteq vrbI \\
\quad [\![\, \mathbf{var}\ k : \mathbb{N}_1 \bullet \\
\qquad k, date! : \left[\begin{array}{l} \left(\begin{array}{l} \forall i, j : 1 \mathrel{..} hwm \bullet i \neq j \Rightarrow names\ i \neq names\ j \\ \exists i : 1 \mathrel{..} hwm \bullet name? = names\ i \end{array} \right), \\ \exists i : 1 \mathrel{..} hwm \bullet name? = names\ i \land date! = dates\ i \end{array} \right] \\
\quad]\!]
\end{array} \qquad \lhd
$$

This program is a variable block that introduces the auxiliary variable k of type $\mathbb{N}_1$. The scope of k is restricted to the body of the variable block: the specification statement obtained by adding k to the frame of (ii). In the refinement of this program we can rely on the fact that $k \in \mathbb{N}_1$. The symbol on the right margin indicates the program that is refined subsequently: the specification statement as opposed to the variable block as a whole.

The assignment to *date!* can be introduced by the *fassigI* (following assignment introduction) law, which splits a specification statement into the sequential composition of another specification statement with an assignment.

$\sqsubseteq$ *fassigI*

$$k, date! : \left[\begin{pmatrix} \forall i, j : 1 \mathinner{..} hwm \bullet i \neq j \Rightarrow names\ i \neq names\ j \\ \exists i : 1 \mathinner{..} hwm \bullet name? = names\ i \end{pmatrix}, \; \exists i : 1 \mathinner{..} hwm \bullet name? = names\ i \wedge dates\ k' = dates\ i \right] ; \qquad \lhd$$

$date! := dates\ k$

In order to introduce the iteration, we need to identify its invariant: a predicate whose validity is established right before the iteration and that is preserved by it. In our example, a proper invariant is the predicate below.

$$(\exists i : 1 \mathinner{..} hwm \bullet name? = names\ i) \wedge (\forall i : 1 \mathinner{..} k - 1 \bullet name? \neq names\ i)$$

We can introduce this predicate in our specification by splitting the above specification statement into a sequential composition of two other specification statements with the use of *seqcI* (sequential composition introduction). This law introduces an intermediate goal which must be established by the first specification statement in the sequential composition and may be assumed by the second one. The first specification statement may assume the precondition of the original specification statement, and the second specification statement must establish its postcondition. In our example, we take the iteration invariant as the intermediate goal.

$\sqsubseteq$ *seqcI*

$$k, date! : \left[\begin{pmatrix} \forall i, j : 1 \mathinner{..} hwm \bullet i \neq j \Rightarrow names\ i \neq names\ j \\ \exists i : 1 \mathinner{..} hwm \bullet name? = names\ i \end{pmatrix}, \begin{pmatrix} \exists i : 1 \mathinner{..} hwm \bullet name? = names\ i \\ \forall i : 1 \mathinner{..} k' - 1 \bullet name? \neq names\ i \end{pmatrix} \right] ; \qquad \lhd$$

$$k, date! : \left[\begin{pmatrix} \exists i : 1 \mathinner{..} hwm \bullet name? = names\ i \\ \forall i : 1 \mathinner{..} k - 1 \bullet name? \neq names\ i \end{pmatrix}, \; \exists i : 1 \mathinner{..} hwm \bullet name? = names\ i \wedge dates\ k' = dates\ i \right] \qquad (iii)$$

The first specification statement above establishes the invariant. It can be refined to an assignment as shown in the sequel.

$\sqsubseteq$ *assigI*

$k := 1$

The proof-obligation associated to this application of *assigI* is trivial, because the first conjunct of the invariant is in the precondition of the specification statement and, when k' is 1, the second conjunct is a universal quantification over the empty set.

The law *sP* (strengthen postcondition) refines a specification statement by strengthening its postcondition under the assumption that its precondition holds. We apply this law to (*iii*) in order

```
|[ var $k : \mathbb{N}_1$ •
      $k := 1$ ;
      do $name? \neq names\ k \rightarrow k := k + 1$ od ;
      $date! := dates\ k$
]|
```

Figure 1.1: *FindBirthday1* Implementation

to express its postcondition in terms of the iteration invariant.

$(iii) \sqsubseteq sP$

$$k, date! : \left[\begin{pmatrix} \exists i : 1 \mathinner{..} hwm \bullet name? = names\ i \\ \forall i : 1 \mathinner{..} k-1 \bullet name? \neq names\ i \end{pmatrix}, \begin{pmatrix} \exists i : 1 \mathinner{..} hwm \bullet name? = names\ i \\ \forall i : 1 \mathinner{..} k'-1 \bullet name? \neq names\ i \\ name? = names\ k' \end{pmatrix} \right]$$

Applications of *sP* give rise to proof-obligations. In this case, we have to show that the above postcondition implies the postcondition of (iii), when its precondition holds. In order to conclude from $name? = names\ k'$ and $dates\ k' = dates\ k'$ that k' is the i characterised by the existential quantification in the postcondition of (iii), we have just to show that it is in the interval $1 \mathinner{..} hwm$. This follows from the observation that the precondition of (iii) states that *name?* occurs among the first *hwm* elements of *names*, and k' is the first position of *names* where *name?* occurs.

The above specification statement is in a form appropriate for the application of the *itI* (iteration introduction) law, which introduces an iteration that preserves the invariant and, in this case, keeps executing until $name? = names\ k$ holds. In order to guarantee termination, we have to identify a variant: an integer expression whose value is decreased by each step of the iteration, but is bound below by 0. An appropriate variant for our example is $hwm - k$.

$\sqsubseteq itI$

do $name? \neq names\ k \rightarrow$

$$k, date! : \left[\begin{pmatrix} \exists i : 1 \mathinner{..} hwm \bullet name? = names\ i \\ \forall i : 1 \mathinner{..} k-1 \bullet name? \neq names\ i \\ name? \neq names\ k \end{pmatrix}, \begin{pmatrix} \exists i : 1 \mathinner{..} hwm \bullet name? = names\ i \\ \forall i : 1 \mathinner{..} k'-1 \bullet name? \neq names\ i \\ 0 \leq hwm - k' < hwm - k \end{pmatrix} \right]$$

od

The precondition of the specification statement in the body of the iteration includes, besides the iteration invariant, its guard: $name? \neq names\ k$, which certainly holds at that point, as otherwise the iteration would have not proceeded. Under this assumption, the task of this specification

```
|[ var $k : \mathbb{N}$ •
    $ncards!, k := 0, 0$ ;
    do $k \neq hwm \rightarrow$
        $k := k + 1$ ;
        if $dates\ k = today? \rightarrow$
            $cardlist!, ncards! := cardlist! \oplus \{ncards! + 1 \mapsto names\ k\}, ncards! + 1$
        [] $dates\ k \neq today \rightarrow$ skip
        fi
    od
]|
```

Figure 1.2: *Remind*1 Implementation

statement, namely, decrease the variant while preserving the invariant, can be accomplished by the assignment that increases the valne of k by 1.

$\sqsubseteq$ *assigI*
  $k := k + 1$

Since *name*? is not in the first $k - 1$ positions of *names*, and is not in its k-th position either, then obviously *name*? is not in the first k positions of names. By increasing k, we certainly decrease the value of $hwm - k$. And since *name*? is among the first *hwm* elements of *names*, then $hwm > k$, so that $0 \leq hwm - (k + 1)$. These observations account for the proof-obligation that is generated by the above application of *assigI*.

The implementation of *FindBirthday*1 that we have just derived is presented in Figure 1.1. For the sake of conciseness, we do not refine *Remind*1, but in Figure 1.2 we present an implementation for it that can be derived in ZRC.

The conversion and refinement laws that have been used in this section are presented in Appendix D. There we specify precisely the transformations that can be achieved by each of them as well as the proof-obligations that they generate. The main subject of the next chapters is the definition of a model that supports the derivation of these and many other laws.

1.2 Overview

The formalisation of ZRC is based on weakest preconditions. In the next chapter, we present a weakest precondition semantics for Z which we construct from a relational semantics that has been proposed by the Z standardisation committee. Although the weakest precondition semantics is not surprising, its construction gives reassurance as to its adequacy and is itself of interest. To begin with, we establish an isomorphism between predicate transforms and a relational model that has been used elsewhere to formalise the data refinement rules of Z. In second place, we compare this relational model to that used in the Z relational semantics and, finally, we define the weakest precondition semantics. As it consists of a unique definition that considers schemas that specify operations in general, compositional formulations for the weakest precondition of some schema expressions are also provided in Chapter 2.

Chapter 3 concludes the semantics definition of ZRC-L by defining the weakest precondition of its remaining constructs. They are similar to those of Morgan's refinement calculus and, in specifying their weakest precondition, we explain many assumptions of its formalisation. In Chapter 3, we also introduce the definition that we adopt for refinement, and the scope rules of ZRC-L.

The formalisation of ZRC is highly based on that of Morgan's calculus, however, our treatment of procedures, parameters, and recursion follows Back's approach. In Chapter 3, we uncover a rather subtle and unexpected relation between Morgan's and Back's formalisms and the substitution operator that renames the free variables of a program, and show that Morgan's work presents an inconsistency. As a consequence, even though most laws of ZRC concerned with the development of (recursive or parametrised) procedures are similar to those of Morgan's calculus, the model that we present in Chapter 3 to support their derivation is based on Back's work.

Yet in Chapter 3, we present the conversion laws of ZRC and exemplify their application. For the sake of conciseness, we do not discuss each of the refinement laws individually: we concentrate our attention on those that support procedure developments and data refinements. Several of the laws that deal with procedures have no counterpart in Morgan's calculus and formalise its approach to recursion.

The application of ZRC in the refiuement of a small system has already been illustrated in the previous section. Three more sizeable examples are provided in Chapter 4. The first one is a class manager that King has used as a case study for his approach to the refiuement of Z specifications. The second example is part of a text editor for which a C implementation has been obtained using a technique mostly based on verification rules. The third and final example is an Airbus cabin-ilumination system. Its development and that of the text editor suggest the introduction of two additional conversion laws, which we present in Chapter 4 itself.

The last chapter presents our conclusions, discusses some related works, and proposes directions for further research. Finally, four appendices complement the material presented in Chapters 2 and 3. Appendix A explains the less familiar symbols of the mathematical notation employed in Chapter 2, and Appeudix B presents proofs for some of the theorems introduced in this same chapter. The weakest precondition semantics of ZRC-L is summarised in Appendix C. Lastly, Appendix D presents the conversion and the refinement laws of ZRC along with their derivation.

Chapter 2

A Weakest Precondition Semantics for Z

In the same way as a number of other refinement techniques [47, 48, 4, 45], ZRC is formalised in terms of weakest preconditions (*wp*) [14], which are used to define both the meaning of ZRC-L and the refinement relation. This chapter is concerned with the *wp* semantics of ZRC-L or, more specifically, of Z. The remaining constructs of ZRC-L, which are not part of the Z notation, are considered in Chapter 3.

In the next section we reproduce part of the Z relational semantics; this work, which is presented in [8] by the standardisation committee, is the responsibility of Brien. In Section 2.2 we provide an equivalent *wp* semantics for Z which is constructed with basis on the relational semantics itself and on an isomorphism between weakest preconditions and relations. The *wp* semantics is composed of a single definition that contemplates schemas that specify operations in general. In order to facilitate its application, in Section 2.3 we derive compositional formulations for the weakest preconditions of some schema expressions. Finally, Section 2.4 discusses some aspects of the *wp* semantics and examines some related works.

2.1 The Relational Semantics

The part of the Z relational semantics presented in this section is that concerned with the definition of schemas. As a *wp* semantics considers only the meaning of operations, and these are specified in Z by schemas, we concentrate here on their definition. Basically, we introduce the definitions used in Section 2.2. The complete specification is presented in [8].

The relational semantics is defined in a denotational style. It is based on an abstract syntax and on semantic functions which map schemas, declarations, or predicates, for instance, to values of a semantic universe. These functions are specified compositionally.

2.1.1 Syntax

The abstract syntax of Z is partially defined below using a BNF-like notation. We write terminal symbols enclosed in quotation marks and non-terminal symbols in italics.

$$
\begin{array}{rcl}
\mathit{Schema} & ::= & \mathit{SConstruction} \\
\mathit{SConstruction} & ::= & `(\text{'}\ \mathit{Decl}\ `|\text{'}\ \mathit{Pred}\ `)\text{'} \\[1ex]
\mathit{Decl} & ::= & \mathit{SimpleDecl} \\
 & | & \mathit{CompndDecl} \\
\mathit{SimpleDecl} & ::= & \mathit{VarName}, \mathit{VarName}, \ldots, \mathit{VarName}\ `:\text{'}\ \mathit{Exp} \\
\mathit{CompndDecl} & ::= & \mathit{Decl}\ `;\text{'}\ \mathit{Decl} \\[1ex]
\mathit{SchemaText} & ::= & \mathit{SimpleScT} \\
 & | & \mathit{CompndScT} \\
\mathit{SimpleScT} & ::= & \mathit{Decl} \\
\mathit{CompndScT} & ::= & \mathit{Decl}\ `|\text{'}\ \mathit{Pred}
\end{array}
$$

The syntactic categories *Pred*, *VarName*, and *Exp* correspond to the Z predicates, variable names, and expressions, respectively.

2.1.2 Semantic Universe

The semantic universe is based on ZF set theory. It comprises denotations for names, types, values, and specifications as a whole.

The language used in the specification of the semantic universe (and of the semantic functions) is defined in [8]. It consists mainly of conventional mathematical or Z notation together with some set and relational operators. The unusual operations used here and in Section 2.2 are enumerated and briefly explained in Appendix A.

Names and Types

The paragraphs of a Z specification introduce names and associate with each of them a type. We can distinguish different sorts of names: schema, variable, and constant names. Therefore, the set *Name*, which contains all names that can be used in a specification, is partitioned by the sets *SchemaName*, *Variable*, and *Constant*, which contain, respectively, all valid schema, variable, and constant names. The set of given set names, which is called *GivenSetName*, is a subset of *Constant*.

A type is either a given set, a power set, a cartesian product or a schema type. As a cousequence, *Type*, the set of all valid types, is partitioned into the sets *Gtype*, *Ptype*, *Ctype* and *Stype*. Tbe structure of *Type* is defined by the constructors *givenT*, *powerT*, *cproductT* and *schemaT*.

$$
\begin{array}{l}
\mathit{givenT} : \mathit{GivenSetName} \pinj \mathit{Gtype} \\
\mathit{powerT} : \mathit{Type} \pinj \mathit{Ptype} \\
\mathit{cproductT} : \mathit{Type}^{+} \pinj \mathit{Ctype} \\
\mathit{schemaT} : \mathit{Signature} \pinj \mathit{Stype}
\end{array}
$$

A given set type is constructed out of its name by *givenT*; a power set type is constructed by *powerT* from its base type; *cproductT* constructs a cartesian product type out of the tuple composed of its base types; finally, *schemaT* takes a signature and constructs a schema type. A signature is a finite partial function from *Variable* to *Type*.

Each type is associated with a set of values, which is called its carrier set. This association is established by a function named *Carrier*.

Elements

A pair formed by a type and a value of its carrier set is called an element. The set *Elm*, that contains all these pairs, is defined as a relation between types and values.

Definition 2.1 $Elm == Carrier \mathbin{⨾} \ni$

The membership relation for elements ($\exists$) associates an element whose value is a set with elements whose values belong to this set. The type in a set-valued element is a power set.

Definition 2.2 $\exists == (powerT^{-1} \times \ni)$

In this definition, $(\_ \times \_)$ is not used as the traditional ZF operator; $(powerT^{-1} \times \ni)$ relates a pair formed by a power set type *pt* and a set *s* (a set-valued element) to the pairs formed by the base type of *pt* and a member of *s*. The definition of $(\_ \times \_)$ adopted here is presented in Appendix A.

An association of variable names with elements is called a situation. The set *Situation* contains all finite partial mappings from *Variable* to *Elm*.

Definition 2.3 $Situation == Variable \mathrel{+\!\!\!\!\twoheadrightarrow} Elm$

The typing and value constraints in a generic definition (schema or constant) may be specified in terms of its parameters. As a consequence, generic types and generic elements have to be considered.

Generics

A generic type is either a type itself or a function. The type of a generic schema or constant is represented by a function which defines the type of each of the schema or constant instantiations. The type of an instantiation is determined by the value it ascribes to the parameters. The set of all generic types is called *GenType*.

Definition 2.4 $GenType == Type \cup \bigcup_{n>0}(Ptype^n \rightarrow Type)$

Similarly, a generic element can be an ordinary element or a function from tuples of set-valued elements to elements. The set containing all set-valued elements is *Pelm*. In its definition, *Elm* is viewed as a relation.

Definition 2.5 $Pelm == Ptype \lhd Elm$

The set of generic elements is *GenElm*. Its definition is very much like that of *GenType*.

Definition 2.6 $GenElm == Elm \cup \bigcup_{n>0}(Pelm^n \rightarrow Elm)$

A declaration, predicate or schema, for instance, can be defined in terms of names that have been previously introduced in the specification. As a result, their meaning depends, in general, on the types and values of these names or, in other words, on the environment.

Environments

An environment records a particular association of types and values with names. The set of all environments, *Env*, contains all finite partial functions from names to generic elements.

Definition 2.7 $Env == Name \twoheadrightarrow GenElm$

Environments and situations play a major role in the relational semantics of schemas. In what follows, we reproduce its definition.

2.1.3 The Semantics of Schemas

The semantic function that defines the meaning of schemas is $(\!| \_ |\!)^{\mathcal{M}s}$, which maps schemas to relations between environments and situations. For a schema S, the relation $(\!| S |\!)^{\mathcal{M}s}$ associates an environment with each of the situations that assign elements to the components of S according to its definition.

$$(\!| S |\!)^{\mathcal{M}s} : Env \leftrightarrow Situation$$

As the definition of S may depend on the environment, different situations may be associated with different environments.

The meaning of a schema $\langle D \mid P \rangle$ is defined in terms of the meaning of its declaration — $(\!| D |\!)^{\mathcal{M}}$ — and of its schema text — $(\!| D \mid P |\!)^{\mathcal{M}}$. These are defined in the sequel.

$$(\!| \langle D \mid P \rangle |\!)^{\mathcal{M}s} = (\!| D |\!)^{\mathcal{M}} \cap ((\!| D \mid P |\!)^{\mathcal{M}} \operatorname{⨾} \supseteq)$$

The relation $(\!| D \mid P |\!)^{\mathcal{M}}$ associates an environment ρ with each of its enrichments that include the variables declared in D and satisfying the restrictions in D and P. The composition $(\!| D \mid P |\!)^{\mathcal{M}} \operatorname{⨾} \supseteq$ relates ρ to all subsets of these enrichments. The intersection rules out the subsets that are not situations that assign values to precisely the variables declared in D.

The Semantics of Declarations

The function $(\!| \_ |\!)^{\mathcal{M}}$ establishes the meaning of declarations. For a declaration D, $(\!| D |\!)^{\mathcal{M}s}$ is the relation that associates an environment with all situations that assign values to exactly those variables declared in D in accordance with its restrictions.

$$(\!| D |\!)^{\mathcal{M}} : Env \leftrightarrow Situation$$

If the type definitions of D rely on the environment, then in general $(\!| D |\!)^{\mathcal{M}}$ relates different sets of situations to different environments.

The definition of $(\!| \_ |\!)^{\mathcal{M}}$ is given by recursion over *Decl*. The base case is a simple declaration of the form $n_1, \ldots, n_m : s$.

$$(\!| n_1, \ldots, n_m : s |\!)^{\mathcal{M}} \;=\; [\![s]\!]^{\mathcal{M}} \operatorname{⨾} \langle (n_1{}^{\circ}, \ni), \ldots, (n_m{}^{\circ}, \ni) \rangle \operatorname{⨾} \{\ldots\}$$

The relation $[\![s]\!]^{\mathcal{M}}$ defines the meaning of the set expression s: a function that associates an environment with the element that represents the type and the value of s in that environment. This element is related by $\langle (n_1{}^{\circ}, \ni), \ldots, (n_m{}^{\circ}, \ni) \rangle$ to every m-tuple of pairs of the form (n_i, x), where i is the position of the pair in the tuple and x, an element of s. Finally, $\{\ldots\}$ associates each of these tuples with the corresponding situation (set of pairs). Altogether, $(\![n_1, \ldots n_m : s]\!)^{\mathcal{M}}$ relates an environment to all situations that associate any value of s to each n_i.

A compound declaration has its meaning specified as follows.

$$(\![D_1;\ D_2]\!)^{\mathcal{M}} = \langle (\![D_1]\!)^{\mathcal{M}}, (\![D_2]\!)^{\mathcal{M}} \rangle \mathbin{;} \sqcup$$

The relation $\langle (\![D_1]\!)^{\mathcal{M}}, (\![D_2]\!)^{\mathcal{M}} \rangle$ associates an environment ρ with the pairs of situations that relate elements to the variables declared in D_1 and D_2 in a way that respects their definitions. The pairs that are compatible as functions (in the sense that variables that belong to the domain of both of them are associated with the same value) are related to their union by $\sqcup$.

The set of names introduced by a declaration is known as its alphabet. This set is specified by the function α, which is defined as shown below. Application of this particular function to a declaration D is represented simply by juxtaposition: αD.

$$\alpha(n_1, \ldots, n_m : s) = \{n_1, \ldots, n_m\}$$
$$\alpha(D_1;\ D_2) = \alpha D_1 \cup \alpha D_2$$

The alphabet of a simple declaration contains exactly the variable names $n_1, \ldots, n_m$ that it introduces. In the case of a compound declaration, its alphabet is the union of the alphabets of its components.

The Semantics of Schema Texts

The meaning of schema texts is defined by $\{\!\_\}^{\mathcal{M}}$. This function associates a schema text with a relation between environments: for a schema text St, $\{\!St\}^{\mathcal{M}}$ associates an environment with all its enrichments by situations that assign elements to the components of St in accordance with its definition.

$$\{\!St\}^{\mathcal{M}} : Env \leftrightarrow Env$$

As in the case of schemas and declarations, the definition of St may depend on the environment and, that being so, different environments may determine different sets of situations.

The definition of $\{\!\_\}^{\mathcal{M}}$ is by recursion over *SchemaText*. The base case is a declaration.

$$\{\!D\}^{\mathcal{M}} = \langle 1, (\![D]\!)^{\mathcal{M}} \rangle \mathbin{;} \oplus$$

The relation $\{\!D\}^{\mathcal{M}}$ associates an environment ρ with each of the environments that can be obtained by enriching ρ with a situation that is related to it in $(\![D]\!)^{\mathcal{M}}$.

The semantics of a compound schema text is defined as follows.

$$\{\!D \mid P\}^{\mathcal{M}} = \{\!D\}^{\mathcal{M}} \rhd [\![P]\!]^{\mathcal{M}}$$

The set $[\![P]\!]^{\mathcal{M}}$ contains the environments that satisfy the predicate P. While $\{\!D\}^{\mathcal{M}}$, as mentioned above, relates an environment to each of its enrichments that takes the declaration D into account, $\{\!D \mid P\}^{\mathcal{M}}$ relates that environment just to those of these enrichments that satisfy P.

The Semantics of Predicates

A predicate P has its meaning defined by the set $\{\!|P|\!\}^{\mathcal{M}}$, which, as already said, contains the environments in which P holds. This set is defined as the intersection of the set of environments in which P is well-typed and the set of environments in which P is supported. A predicate is supported in an environment if it is true in that environment. For example, $\neg\ (x \in x)$ is supported in all environments, since the axiom of regularity ensures that $x \in x$ is false. Nonetheless, $\neg\ (x \in x)$ is not well-typed in any environment, so that its meaning is the empty set of environments.

Here, we present only the definition of $\{\!|P|\!\}^{\mathcal{V}}$, the set of environments in which P is supported, and actnally restrict ourselves to negations, conjunctions, implications, and existential and universal quantifications. These are the definitions used in Section 2.2.

An environment supports a negation $\neg\ P$ if it does not support P.

$$\{\!|\neg\ P|\!\}^{\mathcal{V}} = Env \setminus \{\!|P|\!\}^{\mathcal{V}}$$

The set of environments in which a conjunction $P_1 \wedge P_2$ is snpported is the intersection of the set of environments in which P_1 is supported with the set of environments in which P_2 is supported.

$$\{\!|P_1 \wedge P_2|\!\}^{\mathcal{V}} = \{\!|P_1|\!\}^{\mathcal{V}} \cap \{\!|P_2|\!\}^{\mathcal{V}}$$

An implication $P_1 \Rightarrow P_2$ is supported in any environment that snpports $\neg\ P_1$ or P_2.

$$\{\!|P_1 \Rightarrow P_2|\!\}^{\mathcal{V}} = \{\!|\neg\ P_1|\!\}^{\mathcal{V}} \cup \{\!|P_2|\!\}^{\mathcal{V}}$$

Existential quantifications have the form $\exists\, St \bullet P$, where St is a schema text and P is a predicate. An environment ρ supports a predicate $\exists\, St \bullet P$ if P is snpported in some enrichment of ρ that takes the definition of St into account.

$$\{\!|\exists\, St \bullet P|\!\}^{\mathcal{V}} = \mathrm{dom}(\{\!|St|\!\}^{\mathcal{M}} \rhd \{\!|P|\!\}^{\mathcal{V}})$$

As previonsly explained, $\{\!|St|\!\}^{\mathcal{M}}$ associates an environment ρ with all its enrichments that consider St. The relation $\{\!|St|\!\}^{\mathcal{M}} \rhd \{\!|P|\!\}^{\mathcal{V}}$ associates ρ only to those enrichments that support P; its domain contains exactly the environments that support $\exists\, St \bullet P$. Universal quantifications are defined in terms of existential quantifications.

$$\{\!|\forall\, St \bullet P|\!\}^{\mathcal{V}} = \{\!|\neg\ \exists\, St \bullet \neg\ P|\!\}^{\mathcal{V}}$$

This definition relies on de Morgan's law; $\forall\, St \bullet P$ is snpported in an environment ρ if P is supported in all enrichments of ρ that reflect St.

This relational semantics is, as pointed out before, a subset of that specified in [8]. In the next section we present an equivalent weakest precondition semantics for Z.

2.2 A Weakest Precondition Semantics

In [14], where weakest preconditions were first introdnced, they are used to define the semantics of a programming language. A semantics based on weakest preconditions consists of the definition of a function called, in general, *wp*. This function determines, when applied to a program P and to a predicate ψ, the weakest precondition that guarantees that P terminates in a state that satisfies ψ. The predicate ψ is called a postcondition.

In this section we construct a *wp* semantics for Z or, more precisely, based on the relational semantics presented in the previous section, we determine the result of applying *wp* to a schema that specifies an operation. This is a predicate transformer: a function from predicates to predicates. We establish a correspondence between relations and predicate transformers, and use $(\!|\_|\!)^{\mathcal{M}_S}$, the semantic function that defines the relational model of a schema, to specify its weakest precondition. The *wp* function so defined specifies a weakest precondition semantics for Z that is equivalent to its relational semantics in the sense precisely defined by the correspondence betweeu relations and predicate transformers.

Firstly, we consider an alternative relational model where initial states and inputs are related to final states and outputs. The correspondence hetween this model and predicate transformers is examined in a general setting rather than in the particnlar context of Z. Secondly, we present a way of expressiug the relational model defined by $(\!|\_|\!)^{\mathcal{M}_S}$ for schemas that specify operations in terms of the alternative relational model. Finally, we define *wp*.

2.2.1 Predicate Transformers and Relations

Weakest precondition semantics is based on the principle that the meaning of a program is properly characterised only if, for every postcondition ψ, the preconditions that guarantee termination in a state that satisfies ψ can be identified. In other words, *wp* is supposed to be well-defined for all postconditions ψ. For this reason, we impose no restriction over their sets of free variables, which, in the context of Z, may include those that represent the final state and the outputs, and those representing the initial state and the inputs as well.

At this stage, we represent predicates as sets. We consider a set I of all possible initial states, and a set F of all final states. Inputs are regarded as part of the initial states, and outputs, as part of the final states. Predicates over initial states (and inputs) are elements of $\mathbb{P}\,I$, with $\varnothing$ representing false and I, true, for instance. Postconditions, which are predicates over the initial and final states, are represented as elements of $\mathbb{P}(I \times F)$, or rather, as relations between initial and final states. Altogether, the domaiu of predicate transformers that we consider is the set of total functions $\mathbb{P}(I \times F) \rightarrow \mathbb{P}\,I$.

In contrast with Dijkstra's *wp*, the postconditions to which these predicate transformers can be applied represent state transitions instead of fiual states. They determine, when applied to a postcondition ψ, the weakest precondition that guarantees that the program that it represents perform the state transition specified by ψ.

As a matter of fact, we identify two healthiness conditions and consider only the predicate transformers that satisfy them. The first of these healthiness conditions is positive conjunctivity. A predicate transformer *pt* is positively conjunctive if it distributes over non-empty intersections:

$$pt.(\bigcap\{\, i \bullet S_i \,\}) = \bigcap\{\, i \bullet pt.S_i \,\} \text{ provided } \{\, i \bullet S_i \,\} \neq \varnothing \tag{2.1}$$

Except when traditional mathematical notation is used, function application is represented by a period, so that, for instance, $pt.(\bigcap\{\, i \bullet S_i \,\})$ is the application of *pt* to the postcondition $\bigcap\{\, i \bullet S_i \,\}$. At this point, we depart from [8], where function application is represented by subscription. This notation is not convenient for our purposes because, in many cases, we apply this operator repeatedly and to lengthy expressions. Positively conjunctive predicate transformers correspond to operations that do not present angelic nondeterminism [6]. Monotonicity with respect to $\subseteq$ is a consequence of positive conjunctivity [15].

The second healthiness condition is concerned with the specification of initial states in postconditions:

$$i \in pt.((\{i\} \times F_1) \cup S) \Rightarrow i \in pt.(\{i\} \times F_1) \text{ for any } F_1 \subseteq F, S \subseteq (I \setminus \{i\}) \times F \qquad (2.2)$$

If i belongs to $pt.((\{i\} \times F_1) \cup S)$, then we can conclude that pt either is miraculous at i or, when executed in i, is guaranteed to lead to a final state in F_1. In both cases, i must belong to $pt.(\{i\} \times F_1)$. An operation is miraculous at a state i if it can achieve whatever postcondition is required, including false, when executed from i [47]. Monotonicity implies that, since $\{i\} \times F_1 \subseteq (\{i\} \times F_1) \cup S$, we can actually strengthen (2.2) to an equivalence.

The lemma below identifies a property of the predicate transformers that satisfy both (2.1) and (2.2). This result is used later on in this section.

Lemma 2.1 *For every predicate transformer pt that satisfies both (2.1) and (2.2), initial state $i \in I$, and set of final states $F_1 \subseteq F$,*

$$i \in pt.(\{i\} \times F_1) \equiv i \in pt.(I \times F_1)$$

Proof

($\Rightarrow$) By $\{i\} \times F_1 \subseteq I \times F_1$ and monotonicity.

($\Leftarrow$) By $I \times F_1 = (\{i\} \times F_1) \cup ((I \setminus \{i\}) \times F_1)$ and (2.2).

□

The relational model that we consider at this point is $I_\perp \leftrightarrow F_\perp$, the set of relations between $I_\perp$ and $F_\perp$, where $I_\perp$ is the set $I \cup \{\perp\}$ and, likewise, $F_\perp$ is $F \cup \{\perp\}$. In this model, a relation associates an initial state i with a final state f when the execution of the operation that it represents may lead to state f from state i. The distinguished state $\perp$ represents nontermination: it is the state reached when an operation fails to terminate. A partial relation represents an operation that is miraculous at the states that are not in its domain.

An operation that (for a particular initial state) always fails to terminate is not regarded as being any worse than another one that may fail to terminate just sometimes. Consequently, there is no interest in distinguishing these cases and we further restrict our model by assuming that, when an operation may fail to terminate, it may also terminate and establish any arbitrary result. Formally, we assume that the relations R of our model satisfy the following healthiness condition.

$$\forall i : I_\perp \bullet \perp \in R(\!| \{i\} |\!) \Rightarrow R(\!| \{i\} |\!) = F_\perp \qquad (2.3)$$

In words, if, when executed in a state i, R may lead to $\perp$, then it may lead to any final state whatsoever.

When an operation fails to terminate, it is not possible to execute another operation and recover from this situation. Therefore we impose yet another restriction on the relations of our model:

$$\perp R \perp \qquad (2.4)$$

This healthiness condition guarantees that, whenever an operation is executed after some other operation has failed to terminate, it also leads to nontermination.

The domain of predicate transformers and the relational model we have just presented are isomorphic. In order to prove this result, we define the weakest precondition of a relation; define a function that gives a relational semantics for predicate transformers; and show that these functions are each other's inverse.

The function $r2wp$ can be applied to a relation R and to a postcondition S to determine the weakest precondition that guarantees that R establishes S. Its definition is as follows.

Definition 2.8 *For every relation R and postcondition S,*

$$r2wp.R.S = \overline{\text{dom}\ (R \setminus \overline{S})}$$

By considering $R \setminus \overline{S}$, we identify all possible ways in which R may fail to establish S. Therefore, the complement of the domain of this relation contains exactly those initial states in which the execution of R is guaranteed to achieve S. Whatever postcondition S is considered, the states that are not in the domain of R are always included in $\overline{\text{dom}(R \setminus \overline{S})}$. This is in accordance with our previous observation that R is miraculous at these states and therefore can achieve any postcondition required.

By way of illustration, we consider $I = \{i_1, i_2, i_3\}$ and $F = \{f_1, f_2, f_3\}$. If R is the relation

$$\begin{array}{llll} \{ & (\perp,\perp), & (\perp,f_1), & (\perp,f_2), & (\perp,f_3), \\ & (i_1,f_1), & (i_1,f_2), & & \\ & (i_3,f_3) & & & \} \end{array}$$

we can deduce that $r2wp.R.\{(i_1,f_2),(i_3,f_3)\}$ is equal to $\{i_2, i_3\}$. Indeed, if executed from $\perp$, R is not even guaranteed to terminate, and from i_1, it may achieve f_1 as well as f_2. On the other hand, R is miraculous at i_2 and, if executed from i_3, it is guaranteed to reach f_3.

The theorem presented below shows that the predicate transformers defined by $r2wp$ satisfy the healthiness conditions we proposed above.

Theorem 2.1 *For every relation R, rw2p.R satisfies the healthiness conditions (2.1) and (2.2).*

Proof

Healthiness condition (2.1):

$r2wp.R.(\bigcap\{\ i \bullet S_i\ \})$

$= \overline{\text{dom}(R \setminus \overline{(\bigcap\{\ i \bullet S_i\ \})})}$ [by definition of $r2wp$]

$= \overline{\text{dom}\ \bigcup\{\ i \bullet R \setminus \overline{S_i}\ \}}$ [by a property of sets]

$= \overline{\bigcup\{\ i \bullet \text{dom}(R \setminus \overline{S_i})\ \}}$ [by a property of dom]

$= \bigcap\{\ i \bullet \overline{\mathrm{dom}(R \setminus S_i)}\ \}$ [by a property of sets]

$= \bigcap\{\ i \bullet r2wp.R.S_i\ \}$ [by definition of $r2wp$]

Healthiness condition (2.2):

$i \in r2wp.R.((\{i\} \times F_1) \cup S)$

$\equiv i \in \overline{\mathrm{dom}(R \setminus ((\{i\} \times F_1) \cup S))}$ [by definition of $r2wp$]

$\equiv i \notin \mathrm{dom}(R \setminus ((\{i\} \times F_1) \cup S))$ [by a property of sets]

$\equiv \{i\} \lhd R \subseteq (\{i\} \times F_1) \cup S$ [by a property of sets]

$\equiv \{i\} \lhd R \subseteq \{i\} \times F_1$ [by $S \subseteq (I \setminus \{i\}) \times F$]

$\equiv i \in r2wp.R.(\{i\} \times F_1)$ [by definition of $r2wp$ and the previous steps]

□

The relation corresponding to a predicate transformer is determined by $wp2r$. The definition of this function is presented in the sequel.

Definition 2.9 *For every predicate transformer pt,*

$$wp2r.pt = \{\ i : I_\perp;\ f : F_\perp \mid i \in \overline{pt.(I \times \overline{\{f, \perp\}})}\ \}$$

The postcondition $I \times \overline{\{f, \perp\}}$ simply specifies all final states different from f, since no particular initial state is determined. The predicate $\overline{pt.(I \times \overline{\{f, \perp\}})}$ characterises the initial states in which execution of pt is not guaranteed to avoid f or, to put it more simply, the initial states in which execution of pt may lead to f.

For every monotonic predicate transformer pt, the relation $wp2r.pt$ satisfies the healthiness conditions (2.3) and (2.4). This can be proved by relying on the definition of $wp2r$ and by observing that $pt.(I \times \overline{\{\perp\}}) = pt.(I \times F)$ and $\perp$ is not in the range of predicate transformers. For the sake of conciseness, we do not present the details here.

The theorems that follow establish that $r2wp$ and $wp2r$ are each other's inverse, and therefore establish an isomorphism between the relational and the predicate transformer model.

Theorem 2.2 *For every predicate transformer pt that satisfies the healthiness conditions (2.1) and (2.2),*

$$r2wp.(wp2r.pt) = pt$$

Proof For every postcondition S,

$r2wp.(wp2r.pt).S$

$= \overline{\mathrm{dom}((wp2r.pt) \setminus S)}$ [by definition of $r2wp$]

$= \overline{\mathrm{dom}(\{\ i : I_\perp;\ f : F_\perp \mid i \in \overline{pt.(I \times \overline{\{f, \perp\}})}\ \} \setminus S)}$ [by definition of $wp2r$]

$= \overline{\mathrm{dom}\{\ i : I_\perp;\ f : F_\perp \mid (i, f) \notin S \wedge i \in \overline{pt.(I \times \overline{\{f, \perp\}})}\ \}}$ [by a property of sets]

$$
\begin{aligned}
&= \overline{\{\, i : I_\perp \mid (\exists f : \overline{S\llparenthesis \{i\} \rrparenthesis} \bullet i \in \overline{pt.(I \times \overline{\overline{\{f,\perp\}}})}) \,\}} && \text{[by definition of dom]}\\
&= \{\, i : I_\perp \mid (\forall f : \overline{S\llparenthesis \{i\} \rrparenthesis} \bullet i \in pt.(I \times \overline{\{f,\perp\}})) \,\} && \text{[by a property of sets]}\\
&= \{\, i : I_\perp \mid i \in \bigcap\{\, f : \overline{S\llparenthesis \{i\} \rrparenthesis} \bullet pt.(I \times \overline{\{f,\perp\}}) \,\} \,\} && \text{[by a property of sets]}\\
&= \{\, i : I_\perp \mid i \in pt.(\bigcap\{\, f : \overline{S\llparenthesis \{i\} \rrparenthesis} \bullet I \times \overline{\{f,\perp\}} \,\}) \,\} && \text{[by } \perp \in \overline{S\llparenthesis \{i\} \rrparenthesis} \neq \varnothing \text{ and (2.1)]}\\
&= \{\, i : I_\perp \mid i \in pt.(I \times S\llparenthesis \{i\} \rrparenthesis) \,\} && \text{[by a property of sets]}\\
&= \{\, i : I_\perp \mid i \in pt.(\{i\} \times S\llparenthesis \{i\} \rrparenthesis) \,\} && \text{[by Lemma 2.1]}\\
&= \{\, i : I_\perp \mid i \in pt.S \,\} && \text{[by } S = (\{i\} \times S\llparenthesis \{i\} \rrparenthesis) \cup (\{i\} \ntriangleleft S)\text{, and (2.2)]}\\
&= pt.S && \text{[by a property of sets]}
\end{aligned}
$$

□

Theorem 2.3 *For every relation R that satisfies the healthiness conditions (2.3) and (2.4),*

$$wp2r.(r2wp.R) = R$$

Proof

$$
\begin{aligned}
&wp2r.(r2wp.R)\\
&= \{\, i : I_\perp;\ f : F_\perp \mid i \in \overline{r2wp.R.(I \times \overline{\{f,\perp\}})} \,\} && \text{[by definition of } wp2r\text{]}\\
&= \{\, i : I_\perp;\ f : F_\perp \mid i \in \overline{\overline{\operatorname{dom}(R \setminus (I \times \overline{\{f,\perp\}}))}} \,\} && \text{[by definition of } r2wp\text{]}\\
&= \{\, i : I_\perp;\ f : F_\perp \mid i \in \operatorname{dom}(R \setminus (I \times \overline{\{f,\perp\}})) \,\} && \text{[by a property of sets]}\\
&= \{\, i : I_\perp;\ f : F_\perp \mid i \in \operatorname{dom}((\{\perp\} \lhd R) \cup (R \rhd \{f,\perp\})) \,\} && \text{[by } R \subseteq I_\perp \times F_\perp\text{]}\\
&= \{\, i : I_\perp;\ f : F_\perp \mid i = \perp \vee i\ R\ f \vee i\ R\ \perp \,\} && \text{[by properties of relations]}\\
&= \{\, i : I_\perp;\ f : F_\perp \mid i = \perp \vee i\ R\ f \,\} && \text{[by } i\ R\ \perp \Rightarrow i\ R\ f\text{, by (2.3)]}\\
&= \{\, i : I_\perp;\ f : F_\perp \mid i\ R\ f \,\} && \text{[by } i = \perp \Rightarrow i\ R\ f\text{, by (2.4) and (2.3)]}\\
&= R && \text{[by a property of sets]}
\end{aligned}
$$

□

In the next section, we show that the model used for schemas that specify operations in the relational semantics of Z is isomorphic to a model defined in terms of an instance of the relational model we have presented above.

2.2.2 The Different Relational Models

The Z relational semantics models schemas as relations between environments and situations. In particular, schemas that specify operations are modelled by relations M_R that have the property below, where St, St', $Inp?$, and $Out!$ are sets which, together, contain all the schema components. The set St contains the names of the variables that represent the state components, and St', the variable names that are formed by suffixing a dash ($'$) to the names of St: they represent the components of the after-state. The sets *Inp*? and *Out*! contain the names of the input and output variables, respectively.

$$\forall s : \operatorname{ran} M_R \bullet \operatorname{dom} s = St \cup St' \cup Inp? \cup Out! \tag{2.5}$$

A relation between environments and situations models a schema Op by associating, with an environment ρ, the situations that represent possible assignments of types and values to the components of Op, according to its own definition. The domain of these situations is always the same: the components of Op. This is basically the property asserted by (2.5), which considers schemas whose components are the variables in $St \cup St' \cup Inp? \cup Out!$.

The alternative model that we propose for these schemas consists of functions from environments to relations. In this model, the function that represents a schema Op associates, with an environment ρ, the relation that models the operation defined by Op in ρ. The relational model used is an instance of that presented in Section 2.2.1. The particular sets of initial and final states that we consider are *IStInp* and *FStOut*, which we define below.

$$\begin{aligned}&IStInp = \{\, s : Situation \bullet \operatorname{dom} s = St \cup Inp? \,\}\\ &FStOut = \{\, s : Situation \bullet \operatorname{dom} s = St' \cup Out! \,\}\end{aligned}$$

Altogether, the model that we suggest is a subset of $Env \rightarrow (IStInp_\perp \leftrightarrow FStOut_\perp)$, where Env is the set of environments defined in Section 2.1.2. The functions that we consider have only relations that satisfy the healthiness conditions (2.3) and (2.4) in their range. Moreover, since miraculous operations cannot be specified in Z [42], these relations are total as well.

In what follows, we show that this model is isomorphic to that used in the relational semantics of Z. Firstly, we define a function $r2f$ that transforms a relation between environments and situations into a corresponding function from environments to relations between situations. Secondly, we define $f2r$, which transforms a function from environments to relations back into a relation between environments and situations. Finally, we prove that $r2f$ and $f2r$ are inverse to each other, if applied to relations or functions that satisfy the previously mentioned restrictions.

The definition of $r2f$ is presented below.

Definition 2.10 *For every relation M_R between environments and situations, and environment ρ,*

$$\begin{aligned} r2f.M_R.\rho \;=\; \{\; & isi : IStInp_\perp;\ fso : FStOut_\perp \bullet \\ & (\exists s : Situation \bullet \rho\ M_R\ s \wedge s = isi \cup fso) \vee && (2.6)\\ & \neg\,(\exists s : Situation \bullet \rho\ M_R\ s \wedge isi \subseteq s) && (2.7)\\ \}\; & \end{aligned}$$

provided $\rho \in \operatorname{dom} M_R$.

The domain of the function $r2f.M_R$ is that of M_R. For every environment ρ in the domain of M_R, the relation $r2f.M_R.\rho$ is defined by considering the situations related to ρ in M_R. Each of them assigns types and values to the before and after-state, input, and output variables, and describes a possible behaviour of the operation, when executed in the initial state and with the inputs defined. All pairs of situations from *IStInp* and *FStOut* that can be obtained by splitting these situations are associated in $r2f.M_R.\rho$ – disjunct (2.6) in Definition 2.10. Moreover, if a particular situation isi of $IStInp_\perp$ is not included in any of them or, in other words, the operation aborts (does not terminate or terminates in an arbitrary state) if executed from the state and with the inputs specified by isi, then it is associated in $r2f.M_R.\rho$ to all situations of $FStOut_\perp$ – disjunct (2.7).

The relation $r2f.M_R.\rho$ is total since, for any situation isi of $IStInp_\perp$, either it is included in a situation of $M_R(\!| \{\rho\} |\!)$ – disjunct (2.6) – or it is not – disjunct (2.7). If it is included, it is related in $r2f.M_R.\rho$ to situations fso that are also included in situations of $M_R(\!| \{\rho\} |\!)$. Therefore, isi cannot possibly be related to $\perp$. On the other hand, if isi is not included in any situation of $M_R(\!| \{\rho\} |\!)$, then, as already remarked, it is related to all situations of $FStOut_\perp$. In conclusion, $r2f.M_R.\rho$ satisfies the healthiness condition (2.3). Finally, since $\perp$ is not included in any situation of $M_R(\!| \{\rho\} |\!)$, we conclude that it is related to $\perp$ in $r2f.M_R.\rho$, and so, the healthiness condition (2.4) is satisfied by this relation as well.

The function $f2r$ is defined in the sequel.

Definition 2.11 *For every function M_F from environments to relations, environment ρ, and situation s,*

$$\rho\ (f2r.M_F)\ s \equiv$$

$$\perp \notin (M_F.\rho)(\!| \{(St \cup Inp?) \lhd s\} |\!) \wedge \tag{2.8}$$

$$\exists\, isi : IStInp;\ fso : FStOut \mid (isi, fso) \in M_F.\rho \bullet s = isi \cup fso \tag{2.9}$$

provided $\rho \in \operatorname{dom} M_F$.

An environment ρ in the domain of M_F may be related by $f2r.M_F$ to a situation s only if the initial state and inputs defined by s are not related to $\perp$ in $M_F.\rho$ or, to put it another way, the operation is guaranteed to terminate when executed in this state and with these inputs – conjunct (2.8) of Definition 2.11. In this case, if s can be obtained by combining situations isi and fso related in $M_F.\rho$ – conjunct (2.9), then $\rho\ (f2r.M_F)\ s$ holds.

A direct consequence of the definitions of $IStInp$ and $FStOut$ is that all situations in the range of $f2r.M_F$ have domain $St \cup Inp? \cup St' \cup Out!$. In other words, $f2r.M_F$ satisfies the healthiness condition (2.5).

As we have already hinted, $r2f$ and $f2r$ are each other's inverse. This is proved by the theorems below.

Theorem 2.4 *For every function M_F from environments to total relations between situations of $IStInp_\perp$ and $FStOut_\perp$ that satisfy the healthiness conditions (2.3) and (2.4),*

$$r2f.(f2r.M_F) = M_F$$

Proof For every environment ρ in the domain of M_F,

$$\begin{aligned}
& r2f.(f2r.M_F).\rho \\
& = \{\ isi : IStInp_\perp;\ fso : FStOut_\perp \bullet && \text{[by definition of } r2f] \\
& \qquad (\exists\, s : Situation \bullet \rho\ (f2r.M_F)\ s \wedge s = isi \cup fso) \vee \\
& \qquad \neg\, (\exists\, s : Situation \bullet \rho\ (f2r.M_F)\ s \wedge isi \subseteq s) \\
& \quad \}
\end{aligned}$$

$$\begin{array}{ll}
= \{\ isi : IStInp_\perp;\ fso : FStOut_\perp \bullet & \text{[by definition of } f2r\text{]} \\
\quad (\exists s : Situation \bullet & \\
\qquad \perp \notin (M_F.\rho)(\!|\ \{(St \cup Inp?) \lhd s\}\ |\!) \wedge & \\
\qquad (\exists isi_1 : IStInp;\ fso_1 : FStOut \mid (isi_1, fso_1) \in M_F.\rho \bullet s = isi_1 \cup fso_1) \wedge & \\
\qquad s = isi \cup fso) \vee & \\
\quad \neg (\exists s : Situation \bullet & \\
\qquad \perp \notin (M_F.\rho)(\!|\ \{(St \cup Inp?) \lhd s\}\ |\!) \wedge & \\
\qquad (\exists isi_1 : IStInp;\ fso_1 : FStOut \mid (isi_1, fso_1) \in M_F.\rho \bullet s = isi_1 \cup fso_1) \wedge isi \subseteq s) & \\
\} &
\end{array}$$

$$\begin{array}{ll}
= \{\ isi : IStInp_\perp;\ fso : FStOut_\perp \bullet & \text{[by } (isi \cup fso = isi_1 \cup fso_1) \equiv (isi = isi_1 \wedge fso = fso_1)\text{]} \\
\quad (\perp \notin (M_F.\rho)(\!|\ \{isi\}\ |\!) \wedge (isi, fso) \in M_F.\rho \wedge isi \neq \perp \wedge fso \neq \perp) \vee & \\
\quad \neg (\exists s : Situation \bullet & \\
\qquad \perp \notin (M_F.\rho)(\!|\ \{(St \cup Inp?) \lhd s\}\ |\!) \wedge & \\
\qquad (\exists isi_1 : IStInp;\ fso_1 : FStOut \mid (isi_1, fso_1) \in M_F.\rho \bullet s = isi_1 \cup fso_1) \wedge isi \subseteq s) & \\
\} &
\end{array}$$

$$\begin{array}{ll}
= \{\ isi : IStInp_\perp;\ fso : FStOut_\perp \bullet & \text{[by } s = isi_1 \cup fso_1 \Rightarrow (isi \subseteq s \equiv isi_1 = isi)\text{]} \\
\quad (\perp \notin (M_F.\rho)(\!|\ \{isi\}\ |\!) \wedge (isi, fso) \in M_F.\rho \wedge isi \neq \perp \wedge fso \neq \perp) \vee & \\
\quad \neg (\perp \notin (M_F.\rho)(\!|\ \{isi\}\ |\!) \wedge (\exists fso_1 : FStOut \bullet (isi, fso_1) \in M_F.\rho) \wedge isi \neq \perp) & \\
\} &
\end{array}$$

$$\begin{array}{ll}
= \{\ isi : IStInp_\perp;\ fso : FStOut_\perp \bullet & \text{[by predicate calculus]} \\
\quad ((isi, fso) \in M_F.\rho \wedge fso \neq \perp) \vee \perp \in (M_F.\rho)(\!|\ \{isi\}\ |\!) \vee & \\
\quad \neg (\exists fso_1 : FStOut \bullet (isi, fso_1) \in M_F.\rho) \vee isi = \perp & \\
\} &
\end{array}$$

$$\begin{array}{ll}
= \{\ isi : IStInp_\perp;\ fso : FStOut_\perp \bullet & \text{[by } (isi, fso) \in M_F.\rho \wedge fso = \perp \Rightarrow \perp \in (M_F.\rho)(\!|\ \{isi\}\ |\!)\text{]} \\
\quad (isi, fso) \in M_F.\rho \vee \perp \in (M_F.\rho)(\!|\ \{isi\}\ |\!) \vee & \\
\quad \neg (\exists fso_1 : FStOut \bullet (isi, fso_1) \in M_F.\rho) \vee isi = \perp & \\
\} &
\end{array}$$

$$\begin{array}{ll}
= \{\ isi : IStInp_\perp;\ fso : FStOut_\perp \bullet & \text{[by } isi = \perp \Rightarrow \perp \in (M_F.\rho)(\!|\ \{isi\}\ |\!)\text{, by (2.4)]} \\
\quad (isi, fso) \in M_F.\rho \vee \perp \in (M_F.\rho)(\!|\ \{isi\}\ |\!) \vee \neg \exists fso_1 : FStOut \bullet (isi, fso_1) \in M_F.\rho & \\
\} &
\end{array}$$

$$\begin{array}{ll}
= \{\ isi : IStInp_\perp;\ fso : FStOut_\perp \bullet (isi, fso) \in M_F.\rho \vee \perp \in (M_F.\rho)(\!|\ \{isi\}\ |\!)\ \} & \\
 & \text{[by } M_F.\rho \text{ is total]}
\end{array}$$

$$\begin{array}{ll}
= \{\ isi : IStInp_\perp;\ fso : FStOut_\perp \bullet (isi, fso) \in M_F.\rho\ \} & \\
 & \text{[by } \perp \in (M_F.\rho)(\!|\ \{isi\}\ |\!) \Rightarrow (isi, fso) \in M_F.\rho\text{, for every } fso\text{, by (2.3)]}
\end{array}$$

$$\begin{array}{ll}
= M_F.\rho & \text{[by a property of sets]}
\end{array}$$

□

Theorem 2.5 *For every relation M_R between environments and situations that satisfies (2.5),*

$$f2r.(r2f.M_R) = M_R$$

Proof For every environment ρ in the domain of M_R, and situation s,

$$
\begin{array}{ll}
\rho\ (f2r.(r2f.M_R))\ s & \\
\equiv \bot \notin (r2f.M_R.\rho)(\!|\ \{(St \cup Inp?) \lhd s\}\ |\!) \land & \text{[by definition of } f2r] \\
\quad \exists\, isi : IStInp;\ fso : FStOut \mid (isi, fso) \in r2f.M_R.\rho \bullet s = isi \cup fso & \\
\equiv \neg\ ((\exists\, s_1 : Situation \bullet \rho\ M_R\ s_1 \land s_1 = (St \cup Inp?) \lhd s \cup \bot) \lor & \text{[by definition of } r2f] \\
\qquad \neg\ (\exists\, s_1 : Situation \bullet \rho\ M_R\ s_1 \land (St \cup Inp?) \lhd s \subseteq s_1)) \land & \\
\quad \exists\, isi : IStInp;\ fso : FStOut \bullet & \\
\qquad ((\exists\, s_1 : Situation \bullet \rho\ M_R\ s_1 \land s_1 = isi \cup fso) \lor & \\
\qquad\ \ \neg\ (\exists\, s_1 : Situation \bullet \rho\ M_R\ s_1 \land isi \subseteq s_1)) \land & \\
\qquad s = isi \cup fso & \\
\equiv (\exists\, s_1 : Situation \bullet \rho\ M_R\ s_1 \land (St \cup Inp?) \lhd s \subseteq s_1) \land & \text{[by } \bot \text{ is included in no situation]} \\
\quad \exists\, isi : IStInp;\ fso : FStOut \bullet & \\
\qquad ((\exists\, s_1 : Situation \bullet \rho\ M_R\ s_1 \land s_1 = isi \cup fso) \lor & \\
\qquad\ \ \neg\ (\exists\, s_1 : Situation \bullet \rho\ M_R\ s_1 \land isi \subseteq s_1)) \land & \\
\qquad s = isi \cup fso & \\
\equiv (\exists\, s_1 : Situation \bullet \rho\ M_R\ s_1 \land (St \cup Inp?) \lhd s \subseteq s_1) \land & \\
\quad (\rho\ M_R\ s \lor \neg\ \exists\, s_1 : Situation \bullet \rho\ M_R\ s_1 \land (St \cup Inp?) \lhd s \subseteq s_1) & \\
\qquad \text{[by } s = isi \cup fso \equiv (isi = (St \cup Inp?) \lhd s \land fso = (St' \cup Out!) \lhd s)] & \\
\equiv (\exists\, s_1 : Situation \bullet \rho\ M_R\ s_1 \land (St \cup Inp?) \lhd s \subseteq s_1) \land \rho\ M_R\ s & \text{[by predicate calculus]} \\
\equiv \rho\ M_R\ s & \text{[by } \rho\ M_R\ s \Rightarrow \rho\ M_R\ s \land (St \cup Inp?) \lhd s \subseteq s]
\end{array}
$$

□

In the next section, we use $r2f$ and $r2wp$ to determine the weakest precondition of a schema that specifies an operation in an arbitrary environment ρ.

2.2.3 The Definition of *wp*

Every schema that specifies an operation may be written in the form $\langle d;\ d';\ di?;\ do! \mid p\rangle$, where d declares the variables that represent the state components, d', the corresponding dashed variables, $di?$, the input variables, $do!$, the output variables, and p is a predicate. In order to determine the weakest precondition of these schemas or, in other words, the weakest precondition semantics of Z, we consider initially the model assigned to them by $(\!|\_|\!)^{\mathcal{M}s}$. If Op is a schema that specifies an operation, $r2f.(\!|Op|\!)^{\mathcal{M}s}$ defines it as a function from environments to relations between situations. For an environment ρ in the domain of this function, $r2f.(\!|Op|\!)^{\mathcal{M}s}.\rho$ is the relation between situations that represents Op in this environment. Finally, $r2wp.(r2f.(\!|Op|\!)^{\mathcal{M}s}.\rho)$ is the weakest precondition of Op in ρ.

The expression $r2wp.(r2f.(\!|Op|\!)^{\mathcal{M}s}.\rho)$ is a function from sets to sets: a predicate transformer as defined in Section 2.2.1. Nonetheless, for practical reasons, we want to define wp as a function from Z predicates (elements of the syntactic category *Pred*) to Z predicates. With this aim, we define an interpretation for them as sets: a function $[\![\_]\!]$, which specifies a set representation for Z predicates in ρ.

In order to define this function, first we define a set interpretation for declarations: another function which we also name $[\![\_]\!]$.

$$[\![d]\!] = \{\ s : Situation \bullet \rho\ (\!|d|\!)^{\mathcal{M}} s\ \}$$

The relation $(\!|d|\!)^{\mathcal{M}}$ associates ρ with all situations that assign types and values to the variables declared by d according to its definition. These are the situations that characterise d.

In defining a set representation for the Z predicates, we consider only those predicates that are relevant here. A predicate p over the variables defined in ρ and the alphabet of a declaration $d;\ d';\ di?;\ do!$ is represented by a set of pairs of situations from $[\![d;\ di?]\!]$ and $[\![d';\ do!]\!]$.

$$[\![p]\!] = \{\ isi : [\![d;\ di?]\!];\ fso : [\![d';\ do!]\!] \mid \rho \oplus isi \oplus fso \in \{\!|p|\!\}^{\mathcal{M}}\ \}$$

As explained in Section 2.1.3, $\{\!|p|\!\}^{\mathcal{M}}$ contains all the environments in which p is satisfied. A pair (isi, fso) of situations belongs to $[\![p]\!]$ exactly when the environment $\rho \oplus isi \oplus fso$ belongs to $\{\!|p|\!\}^{\mathcal{M}}$ or, in words, exactly when p is satisfied in $\rho \oplus isi \oplus fso$.

For a predicate p over the variables in ρ and the alphabet of $d;\ di?$, we have the following very similar definition.

$$[\![p]\!] = \{\ isi : [\![d;\ di?]\!] \mid \rho \oplus isi \in \{\!|p|\!\}^{\mathcal{M}}\ \}$$

In this case, p is represented by a set of situations instead of a set of pairs of situations.

The set representations of conjunctions and implications involving predicates over the variables in ρ and those in the alphabet of $d;\ d';\ di?;\ do!$ can be expressed compositionally in the usual way. This is established by the lemma below.

Lemma 2.2 *For all predicates p and q over the variables in the domain of ρ and those in $\alpha(d;\ d';\ di?;\ do!)$,*

$$\begin{aligned}
&[\![p \wedge q]\!] = [\![p]\!] \cap [\![q]\!] \\
&[\![p \Rightarrow q]\!] = \overline{[\![p]\!]} \cup [\![q]\!]
\end{aligned}$$

Proof For the sake of brevity, we consider just implications; the proof for conjunctions is similar. As mentioned in Section 2.1.3, $\{\!|p|\!\}^{\mathcal{M}}$ is the conjunction of the set of environments in which p is well-typed with the set of environments in which p is supported. Here, however, we consider only predicates that are well-typed in the environment ρ and in view of the declarations $d;\ d';\ di?;\ do!$. More precisely, we consider only predicates that are well-typed in any environment $\rho \oplus isi \oplus fso$, where isi and fso are situations that belong to the set representation of $d;\ di?$ and $d';\ do!$, respectively. For such a predicate, $\rho \oplus isi \oplus fso \in \{\!|p|\!\}^{\mathcal{M}}$ is equivalent to $\rho \oplus isi \oplus fso \in \{\!|p|\!\}^{\mathcal{V}}$, where, as explained in Section 2.1.3, $\{\!|p|\!\}^{\mathcal{V}}$ is the set of environments that support p. This fact is used below.

$$\begin{aligned}
&[\![p \Rightarrow q]\!] \\
&= \{\ isi : [\![d;\ di?]\!];\ fso : [\![d';\ do!]\!] \mid \rho \oplus isi \oplus fso \in \{\!|p \Rightarrow q|\!\}^{\mathcal{M}}\ \} && \text{[by definition of } [\![\_]\!]\text{]}
\end{aligned}$$

$$= \{\ isi : [d;\ di?];\ fso : [d';\ do!] \mid \rho \oplus isi \oplus fso \in \{\!|\neg\, p|\!\}^{\mathcal{V}} \cup \{\!|q|\!\}^{\mathcal{V}}\ \}$$

[by the above comment and the definition of $\{\!|\_|\!\}^{\mathcal{V}}$]

$$= \{\ isi : [d;\ di?];\ fso : [d';\ do!] \mid \rho \oplus isi \oplus fso \in Env \setminus \{\!|\neg\, p|\!\}^{\mathcal{V}} \cup \{\!|q|\!\}^{\mathcal{V}}\ \}$$

[by definition of $\{\!|\_|\!\}^{\mathcal{V}}$]

$$= \overline{\{\ isi : [d;\ di?];\ fso : [d';\ do!] \mid \rho \oplus isi \oplus fso \in \{\!|p|\!\}^{\mathcal{V}}\ \}} \cup$$ [by a property of sets]

$$\{\ isi : [d;\ di?];\ fso : [d';\ do!] \mid \rho \oplus isi \oplus fso \in \{\!|q|\!\}^{\mathcal{V}}\ \}$$

$$= \overline{\llbracket p \rrbracket} \cup \llbracket q \rrbracket$$ [by definition of $\llbracket\_\rrbracket$]

□

The next lemma provides a compositional formulation for the set representation of existential quantifications of the form $\exists\, d';\ do! \bullet p$ and universal quantifications of the form $\forall\, d';\ do! \bullet p$.

Lemma 2.3 *For every predicate p over the variables in ρ and those in $\alpha(d;\ d';\ di?;\ do!)$,*

$$\llbracket \exists\, d';\ do! \bullet p \rrbracket = \{\ isi : [d;\ di?] \mid (\exists\, fso : [d';\ do!] \bullet (isi, fso) \in \llbracket p \rrbracket)\ \}$$
$$\llbracket \forall\, d';\ do! \bullet p \rrbracket = \{\ isi : [d;\ di?] \mid (\forall\, fso : [d';\ do!] \bullet (isi, fso) \in \llbracket p \rrbracket)\ \}$$

Proof For the sake of brevity, we consider just universal qnantifications; the proof for existential quantifications is simpler.

$$\llbracket \forall\, d';\ do! \bullet p \rrbracket$$

$$= \{\ isi : [d;\ di?] \mid \rho \oplus isi \in \{\!|\forall\, d';\ do! \bullet p|\!\}^{\mathcal{M}}\ \}$$ [by definition of $\llbracket\_\rrbracket$]

$$= \{\ isi : [d;\ di?] \mid \rho \oplus isi \in \{\!|\forall\, d';\ do! \bullet p|\!\}^{\mathcal{V}}\ \}$$

[by $\forall\, d';\ do! \bullet p$ is well-typed (see comment in the proof of Lemma 2.2]

$$= \{\ isi : [d;\ di?] \mid \rho \oplus isi \in \{\!|\neg\ \exists\, d';\ do! \bullet \neg\, p|\!\}^{\mathcal{V}}\ \}$$ [by definition of $\{\!|\_|\!\}^{\mathcal{V}}$]

$$= \{\ isi : [d;\ di?] \mid \neg\, \rho \oplus isi \in \mathrm{dom}(((1, \{\!|d';\ do!|\!\}^{\mathcal{M}}) \mathbin{⨟} \oplus) \rhd \{\!|\neg\, p|\!\}^{\mathcal{V}})\ \}$$ [by definition of $\{\!|\_|\!\}^{\mathcal{V}}$]

$$= \{\ isi : [d;\ di?] \mid \neg\ \exists\, fso : Situation \bullet \rho \oplus isi \{\!|d';\ do!|\!\}^{\mathcal{M}} fso \wedge \rho \oplus isi \oplus fso \in \{\!|\neg\, p|\!\}^{\mathcal{V}}\ \}$$

[by properties of relations]

$$= \{\ isi : [d;\ di?] \mid \neg\ \exists\, fso : Situation \bullet \rho\ \{\!|d';\ do!|\!\}^{\mathcal{M}} fso \wedge \rho \oplus isi \oplus fso \in \{\!|\neg\, p|\!\}^{\mathcal{V}}\ \}$$

[by the variables of $\alpha(d;\ di?)$ are not free in $d;\ do!$]

$$= \{\ isi : [d;\ di?] \mid \neg\ \exists\, fso : [d';\ do!] \bullet \rho \oplus isi \oplus fso \in \{\!|\neg\, p|\!\}^{\mathcal{V}}\ \}$$ [by definition of $\llbracket\_\rrbracket$]

$$= \{\ isi : [d;\ di?] \mid \neg\ \exists\, fso : [d';\ do!] \bullet \neg\, \rho \oplus isi \oplus fso \in \{\!|p|\!\}^{\mathcal{V}}\ \}$$ [by definition of $\{\!|\_|\!\}^{\mathcal{V}}$]

$$= \{\ isi : [d;\ di?] \mid \neg\ \exists\, fso : [d';\ do!] \bullet \neg\, (isi, fso) \in \llbracket p \rrbracket\ \}$$ [by definition of $\llbracket\_\rrbracket$]

$$= \{\ isi : [d;\ di?] \mid \forall\, fso : [d';\ do!] \bullet (isi, fso) \in \llbracket p \rrbracket\ \}$$ [by predicate calculus]

□

The two lemmas above are used in the sequel in the proof of Theorem 2.6.

Below we consider a postcondition ψ expressed as a Z predicate and define (the set representation of) $wp.Op.\psi$, for an arbitrary schema Op that specifies an operation.

Definition 2.12 *For every schema $\langle d;\ d';\ di?;\ do! \mid p\rangle$, environment ρ, and postcondition ψ,*

$$[\![wp.\langle d;\ d';\ di?;\ do! \mid p\rangle.\psi]\!] = r2wp.(r2f.(\!|\langle d;\ d';\ di?;\ do! \mid p\rangle|\!)^{\mathcal{M}s}.\rho).[\![\psi]\!]$$

The environment is an implicit parameter of wp. As a consequence of Definition 2.12, for every environment ρ, the predicate transformer that wp associates with a schema $\langle d;\ d';\ di?;\ do! \mid p\rangle$ that specifies an operation is equivalent, in the sense precisely defined by $r2wp$ and $r2f$, to the relational model of this schema specified in the relational semantics of Z.

Theorem 2.6 in the sequel presents a definition of wp in terms of Z predicates. Its proof relies on Lemma 2.4, which identifies properties that characterise the representation of a schema that specifies an operation in the relational semantics of Z.

Lemma 2.4 *For every schema $\langle d;\ d';\ di?;\ do! \mid p\rangle$, environment ρ and situation s,*

$$\rho\ (\!|\langle d;\ d';\ di?;\ do! \mid p\rangle|\!)^{\mathcal{M}s}\ s \equiv \rho\ (\!|d;\ d';\ di?;\ do!|\!)^{\mathcal{M}}\ s \wedge (\rho \oplus s) \in \{\![p]\!\}^{\mathcal{M}}$$

Proof

$\rho\ (\!|\langle d;\ d';\ di?;\ do! \mid p\rangle|\!)^{\mathcal{M}s}\ s$

$\equiv (\rho, s) \in (\!|d;\ d';\ di?;\ do!|\!)^{\mathcal{M}} \cap (\{\!|d;\ d';\ di?;\ do! \mid p|\!\}^{\mathcal{M}} \mathbin{;} \supseteq)$ [by definition of $(\!|\_|\!)^{\mathcal{M}s}$]

$\equiv \rho\ (\!|d;\ d';\ di?;\ do!|\!)^{\mathcal{M}}\ s\ \wedge \exists\, \sigma : Env \bullet (\rho, \sigma) \in \{\!|d;\ d';\ di?;\ do! \mid p|\!\}^{\mathcal{M}} \wedge s \subseteq \sigma$

[by properties of sets and relations]

$\equiv \rho\ (\!|d;\ d';\ di?;\ do!|\!)^{\mathcal{M}}\ s\ \wedge$ [by definition of $\{\!|\_|\!\}^{\mathcal{M}}$]
$\exists\, \sigma : Env \bullet (\rho, \sigma) \in ((\mathrm{I}, (\!|d;\ d';\ di?;\ do!|\!)^{\mathcal{M}}) \mathbin{;} \oplus) \rhd \{\![p]\!\}^{\mathcal{M}} \wedge s \subseteq \sigma$

$\equiv \rho\ (\!|d;\ d';\ di?;\ do!|\!)^{\mathcal{M}}\ s\ \wedge$ [by properties of relations]
$\exists\, \sigma : Env \bullet (\exists\, v : Situation \bullet \rho\ (\!|d;\ d';\ di?;\ do!|\!)^{\mathcal{M}}\ v \wedge \sigma = \rho \oplus v) \wedge \sigma \in \{\![p]\!\}^{\mathcal{M}} \wedge s \subseteq \sigma$

$\equiv \rho\ (\!|d;\ d';\ di?;\ do!|\!)^{\mathcal{M}}\ s\ \wedge \exists\, \sigma : Env \bullet \sigma = \rho \oplus s \wedge \sigma \in \{\![p]\!\}^{\mathcal{M}}$

[by $\operatorname{dom} s = \operatorname{dom} v = \alpha(d;\ d';\ di?;\ do!)$ and so $\sigma = \rho \oplus v \Rightarrow (s \subseteq \sigma \equiv s = v)$]

$\equiv \rho\ (\!|d;\ d';\ di?;\ do!|\!)^{\mathcal{M}}\ s \wedge (\rho \oplus s) \in \{\![p]\!\}^{\mathcal{M}}$ [by predicate calculus]

□

The operation $\langle d;\ d';\ di?;\ do! \mid p\rangle$ is guaranteed to terminate exactly when there is a final state and outputs that satisfy p. Furthermore, it is guaranteed to establish ψ upon termination if, whenever p holds, so does ψ. In the theorem below, termination is captured by $(\exists\, d';\ do! \bullet p)$. Correctness or, more precisely, the establishment of ψ, is captured by $(\forall\, d';\ do! \bullet p \Rightarrow \psi)$.

Theorem 2.6 *For every schema $\langle d;\ d';\ di?;\ do! \mid p\rangle$, and postcondition ψ,*

$$wp.\langle d;\ d';\ di?;\ do! \mid p\rangle.\psi \equiv (\exists\, d';\ do! \bullet p) \wedge (\forall\, d';\ do! \bullet p \Rightarrow \psi)$$

Proof We assume that $\langle d;\ d';\ di?;\ do! \mid p\rangle$ is named Op.

$[\![wp.Op.\psi]\!]$

$= r2wp.(r2f.(\!|Op|\!)^{\mathcal{M}s}.\rho).[\![\psi]\!]$ [by definition of wp]

$= \overline{\operatorname{dom}((r2f.(\!|Op|\!)^{\mathcal{M}s}.\rho) \setminus [\![\psi]\!])}$ [by definition of $r2wp$]

$$
= \overline{\begin{array}{l}
\mathrm{dom}(\{\ isi : IStInp_\bot;\ fso : FStOut_\bot \bullet \\
\qquad\qquad (\exists\, s : Situation \bullet \rho\ \llparenthesis Op \rrparenthesis^{\mathcal{M}s}\ s\ \ \land s = isi \cup fso) \lor \\
\qquad\qquad \neg\ (\exists\, s : Situation \bullet \rho\ \llparenthesis Op \rrparenthesis^{\mathcal{M}s}\ s\ \ \land isi \subseteq s) \\
\qquad \} \setminus \llbracket \psi \rrbracket)
\end{array}}
\qquad \text{[by definition of } r2f\text{]}
$$

$$
\begin{array}{l}
= \overline{\begin{array}{l}
\mathrm{dom}(\{\ isi : IStInp_\bot;\ fso : FStOut_\bot \bullet \\
\qquad\qquad \neg\ (\exists\, s : Situation \bullet \rho\ \llparenthesis Op \rrparenthesis^{\mathcal{M}s}\ s\ \ \land isi \subseteq s) \\
\qquad \} \setminus \llbracket \psi \rrbracket)
\end{array}} \cap \\
\quad \overline{\begin{array}{l}
\mathrm{dom}(\{\ isi : IStInp_\bot;\ fso : FStOut_\bot \bullet \\
\qquad\qquad (\exists\, s : Situation \bullet \rho\ \llparenthesis Op \rrparenthesis^{\mathcal{M}s}\ s\ \ \land s = isi \cup fso) \\
\qquad \} \setminus \llbracket \psi \rrbracket)
\end{array}}
\end{array}
\qquad \text{[by properties of sets and relations]}
$$

$$
\begin{array}{l}
= \mathrm{dom}\ \overline{\begin{array}{l}
(\{\ isi : IStInp_\bot;\ fso : FStOut_\bot \bullet \\
\qquad\quad \neg\ (\exists\, s : Situation \bullet \rho\ \llparenthesis Op \rrparenthesis^{\mathcal{M}s}\ s\ \ \land isi \subseteq s) \\
\ \} \setminus \llbracket \psi \rrbracket \fatsemi (FStOut_\bot \times FStOut_\bot))
\end{array}} \cap \\
\quad \mathrm{dom}\ \overline{\begin{array}{l}
(\{\ isi : IStInp_\bot;\ fso : FStOut_\bot \bullet \\
\qquad\quad (\exists\, s : Situation \bullet \rho\ \llparenthesis Op \rrparenthesis^{\mathcal{M}s}\ s\ \ \land s = isi \cup fso) \\
\ \} \setminus \llbracket \psi \rrbracket \fatsemi (FStOut_\bot \times FStOut_\bot))
\end{array}}
\end{array}
$$

[by $\overline{\mathrm{dom}\, R} = \mathrm{dom}\, \overline{(R \fatsemi U)}$, where U is the universe relation]

$$
\begin{array}{l}
= \mathrm{dom}\ \overline{\begin{array}{l}
\{\ isi : IStInp_\bot;\ fso : FStOut_\bot \bullet \\
\qquad \exists\, ifso : FStOut_\bot \bullet \\
\qquad\qquad \neg\ (\exists\, s : Situation \bullet \rho\ \llparenthesis Op \rrparenthesis^{\mathcal{M}s}\ s\ \ \land isi \subseteq s) \land (isi, ifso) \notin \llbracket \psi \rrbracket \\
\}
\end{array}} \cap \\
\quad \mathrm{dom}\ \overline{\begin{array}{l}
\{\ isi : IStInp_\bot;\ fso : FStOut_\bot \bullet \\
\qquad \exists\, ifso : FStOut_\bot \bullet \\
\qquad\qquad (\exists\, s : Situation \bullet \rho\ \llparenthesis Op \rrparenthesis^{\mathcal{M}s}\ s\ \ \land s = isi \cup ifso) \land (isi, ifso) \notin \llbracket \psi \rrbracket \\
\}
\end{array}}
\end{array}
\qquad \text{[by properties of sets and relations]}
$$

$$
\begin{array}{l}
= \mathrm{dom}\, \overline{\{\ isi : IStInp_\bot;\ fso : FStOut_\bot \bullet \neg\ \exists\, s : Situation \bullet \rho\ \llparenthesis Op \rrparenthesis^{\mathcal{M}s}\ s\ \ \land isi \subseteq s\ \}} \cap \\
\quad \mathrm{dom}\ \overline{\begin{array}{l}
\{\ isi : IStInp_\bot;\ fso : FStOut_\bot \bullet \\
\qquad \exists\, ifso : FStOut_\bot \bullet \\
\qquad\qquad (\exists\, s : Situation \bullet \rho\ \llparenthesis Op \rrparenthesis^{\mathcal{M}s}\ s\ \ \land s = isi \cup ifso) \land (isi, ifso) \notin \llbracket \psi \rrbracket \\
\}
\end{array}}
\end{array}
\qquad \text{[by } (isi, \bot) \notin \llbracket \psi \rrbracket\text{, for any } isi\text{]}
$$

$$
\begin{array}{l}
= \{\ isi : IStInp_\bot \bullet \exists\, s : Situation \bullet \rho\ \llparenthesis Op \rrparenthesis^{\mathcal{M}s}\ s\ \ \land isi \subseteq s\ \} \cap \\
\quad \{\ isi : IStInp_\bot \bullet \\
\qquad \forall\, ifso : FStOut_\bot;\ s : Situation \bullet \rho\ \llparenthesis Op \rrparenthesis^{\mathcal{M}s}\ s\ \ \land\ \ s = isi \cup ifso \Rightarrow (isi, ifso) \in \llbracket \psi \rrbracket \\
\quad \}
\end{array}
\qquad \text{[by a property of sets and relations]}
$$

$$
\begin{array}{l}
= \{\ isi : IStInp \bullet \exists\, s : Situation \bullet \rho\ \llparenthesis Op \rrparenthesis^{\mathcal{M}s}\ s\ \ \land isi \subseteq s\ \} \cap \\
\quad \{\ isi : IStInp \bullet \\
\qquad \forall\, ifso : FStOut_\bot;\ s : Situation \bullet \rho\ \llparenthesis Op \rrparenthesis^{\mathcal{M}s}\ s\ \ \land\ \ s = isi \cup ifso \Rightarrow (isi, ifso) \in \llbracket \psi \rrbracket \\
\quad \}
\end{array}
\qquad \text{[by } \bot \not\subseteq s\text{, for any situation } s\text{]}
$$

$$
\begin{array}{ll}
= & \{\ \mathit{isi} : \mathit{IStInp} \bullet \exists\, s : \mathit{Situation} \bullet \rho\ (\!|\mathit{Op}|\!)^{\mathcal{M}S}\ s\ \wedge \mathit{isi} \subseteq s\ \} \cap \\
& \{\ \mathit{isi} : \mathit{IStInp} \bullet \\
& \qquad \forall\, s : \mathit{Situation} \bullet \\
& \qquad\qquad \rho\ (\!|\mathit{Op}|\!)^{\mathcal{M}S}\ s\ \wedge\ \mathit{isi} = (\alpha d \cup \alpha di?) \lhd s \Rightarrow (\mathit{isi}, (\alpha d' \cup \alpha do!) \lhd s) \in [\![\psi]\!] \\
& \}\qquad\qquad\qquad [\text{by } s = \mathit{isi} \cup \mathit{ifso} \equiv \mathit{isi} = (\alpha d \cup \alpha di?) \lhd s \wedge \mathit{ifso} = (\alpha d' \cup \alpha do!) \lhd s] \\
= & \{\ \mathit{isi} : \mathit{IStInp} \bullet \\
& \qquad \exists\, s : \mathit{Situation} \bullet \rho\ (\!|d;\ d';\ di?;\ do!|\!)^{\mathcal{M}}\ s\ \wedge (\rho \oplus s) \in \{\!|p|\!\}^{\mathcal{M}} \wedge\ \mathit{isi} \subseteq s \\
& \} \cap \\
& \{\ \mathit{isi} : \mathit{IStInp} \bullet \\
& \qquad \forall\, s : \mathit{Situation} \bullet \\
& \qquad\qquad \rho\ (\!|d;\ d';\ di?;\ do!|\!)^{\mathcal{M}}\ s\ \wedge (\rho \oplus s) \in \{\!|p|\!\}^{\mathcal{M}} \wedge\ \mathit{isi} = (\alpha d \cup \alpha di?) \lhd s \Rightarrow \\
& \qquad\qquad\qquad (\mathit{isi}, (\alpha d' \cup \alpha do!) \lhd s) \in [\![\psi]\!] \\
& \} \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad [\text{by Lemma 2.4}] \\
= & \{\ \mathit{isi} : \mathit{IStInp} \bullet \\
& \qquad \exists\, s_1 : \mathit{IStInp};\ s_2 : \mathit{FStOut} \bullet \\
& \qquad\qquad \rho\ (\!|d;\ d';\ di?;\ do!|\!)^{\mathcal{M}}\ (s_1 \oplus s_2)\ \wedge (\rho \oplus s_1 \oplus s_2) \in \{\!|p|\!\}^{\mathcal{M}} \wedge\ \mathit{isi} = s_1 \\
& \} \cap \\
& \{\ \mathit{isi} : \mathit{IStInp} \bullet \\
& \qquad \forall\, s_1 : \mathit{IStInp};\ s_2 : \mathit{FStOut} \bullet \\
& \qquad\qquad \rho\ (\!|d;\ d';\ di?;\ do!|\!)^{\mathcal{M}}\ (s_1 \oplus s_2)\ \wedge (\rho \oplus s_1 \oplus s_2) \in \{\!|p|\!\}^{\mathcal{M}} \wedge\ \mathit{isi} = s_1 \Rightarrow \\
& \qquad\qquad\qquad (\mathit{isi}, s_2) \in [\![\psi]\!] \\
& \} \qquad\qquad\qquad [\text{by } \rho\ (\!|d;\ d';\ di?;\ do!|\!)^{\mathcal{M}}\ s \text{ implies } \mathrm{dom}\, s = \alpha d;\ d';\ di?;\ do!] \\
= & \{\ \mathit{isi} : \mathit{IStInp} \bullet \\
& \qquad \exists\, s_2 : \mathit{FStOut} \bullet \rho\ (\!|d;\ di?|\!)^{\mathcal{M}}\ \mathit{isi}\ \wedge \rho\ (\!|d';\ do!|\!)^{\mathcal{M}}\ s_2 \wedge (\rho \oplus \mathit{isi} \oplus s_2) \in \{\!|p|\!\}^{\mathcal{M}} \\
& \} \cap \\
& \{\ \mathit{isi} : \mathit{IStInp} \bullet \\
& \qquad \forall\, s_2 : \mathit{FStOut} \bullet \\
& \qquad\qquad \rho\ (\!|d;\ di?|\!)^{\mathcal{M}}\ \mathit{isi}\ \wedge \rho\ (\!|d';\ do!|\!)^{\mathcal{M}}\ s_2\ \wedge (\rho \oplus \mathit{isi} \oplus s_2) \in \{\!|p|\!\}^{\mathcal{M}} \Rightarrow (\mathit{isi}, s_2) \in [\![\psi]\!] \\
& \} \qquad\qquad [\text{by } \rho (\!|d;\ d';\ di?;\ do!|\!)^{\mathcal{M}}\ (s_1 \oplus s_2) \equiv \rho\ (\!|d;\ di?|\!)^{\mathcal{M}}\ s_1 \wedge \rho\ (\!|d';\ do!|\!)^{\mathcal{M}}\ s_2] \\
= & \{\ \mathit{isi} : [\![d;\ di?]\!] \bullet (\exists\, s_2 : [\![d';\ do!]\!] \bullet (\rho \oplus \mathit{isi} \oplus s_2) \in \{\!|p|\!\}^{\mathcal{M}}\ )\} \cap \\
& \{\ \mathit{isi} : [\![d;\ di?]\!] \bullet (\forall\, s_2 : [\![d';\ do!]\!] \bullet (\rho \oplus \mathit{isi} \oplus s_2) \in \{\!|p|\!\}^{\mathcal{M}} \Rightarrow (\mathit{isi}, s_2)\ \in [\![\psi]\!])\ \} \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad [\text{by definition of } [\![\_]\!] \text{ for declarations}] \\
= & [\![(\exists\, d';\ do! \bullet p) \wedge (\forall\, d';\ do! \bullet p \Rightarrow \psi)]\!] \qquad\qquad\qquad [\text{by definition of } [\![\_]\!] \text{ for predicates}]
\end{array}
$$

□

In order to rule out the possibility of scope conflict, we assume that the before and after-state, the input, and the output variables are not free in the declarations. The free variables of a declaration are those that occur free in the type definitions.

In the next section we consider a few healthiness conditions that are satisfied by *wp*.

2.2.4 Healthiness Conditions

In [14] four properties of *wp* that reflect characteristics of programming languages are pointed out: law of excluded miracle, monotonicity, $\wedge$-distributivity, and continuity. As we show in the sequel, from these, just continuity is not satisfied by the *wp* function that we have defined. The law of excluded miracle holds because miraculous operations cannot be specified in Z, and $\wedge$-distributivity, because angelic operations cannot be specified either. On the other hand, continuity does not necessarily hold because operations of unbounded nondetermiuism can be defined.

Theorem 2.7 *Law of Excluded Miracle.*

$$wp.(d;\ d';\ di?;\ do! \mid p).\mathrm{false} \equiv \mathrm{false}$$

Proof

$wp.(d;\ d';\ di?;\ do!).\,\mathrm{false}$

$\equiv (\exists\, d';\ do! \bullet p) \wedge (\forall\, d';\ do! \bullet p \Rightarrow \mathrm{false})$ [by definition of *wp*]

$\equiv (\exists\, d';\ do! \bullet p) \wedge (\forall\, d';\ do! \bullet \neg\, p)$ [by predicate calculus]

$\equiv (\exists\, d';\ do! \bullet p) \wedge \neg\, (\exists\, d';\ do! \bullet p)$ [by predicate calculus]

$\equiv \mathrm{false}$ [by predicate calculus]

$\square$

Monotonicity and $\wedge$-distribntive are a direct consequence of Theorem 2.1, since *wp* is defined in terms of *r2wp*.

A predicate transformer *pt* is continuous if, for every indexed family $\{\ i : \mathbf{N} \bullet p_i\ \}$ of predicates such that $p_i \Rightarrow p_{i+1}$ for all $i \in \mathbf{N}$, we have that $pt.(\exists\, i : \mathbf{N} \bullet p_i) \equiv (\exists\, i : \mathbf{N} \bullet pt.p_i)$. As mentioned before, *wp* does not necessarily defines a continuous predicate transformer. A counterexample can be provided if we consider the operation that chooses an arbitrary positive integer and the family of predicates $\{\ i : \mathbf{N} \bullet p_i(x')\ \}$ where $p_i(x') \equiv x' < i$. This operation can be specified as follows.

$$\begin{array}{|l}
\hline
CH \\
x : \mathbb{Z} \\
x' : \mathbb{Z} \\
\hline
x' > 0 \\
\hline
\end{array}$$

According to Theorem 2.6, $wp.CH.\psi \equiv (\forall\, x' : \mathbb{Z} \bullet x' > 0 \Rightarrow \psi)$. The family of predicates considered satisfies the property alluded in the characterisation of continuity: $p_i(x') \Rightarrow p_{i+1}(x')$, for all $i \in \mathbf{N}$. Nevertheless, $wp.CH$ does not satisfy the corresponding property. First of all, as we show below, for every i, $wp.CH.p_i(x') \equiv \mathrm{false}$.

$wp.CH.p_i(x')$

$\equiv (\forall\, x' : \mathbb{Z} \bullet x' > 0 \Rightarrow p_i(x'))$ [by definition of *wp*]

$$\equiv (\forall x' : \mathbb{Z} \bullet x' > 0 \Rightarrow x' < i) \qquad \text{[by definition of } p_i(x')\text{]}$$
$$\equiv \text{false} \qquad \text{[by a property of } <\text{]}$$

In conclusion, $(\exists i : \mathbb{N} \bullet wp.CH.p_i(x'))$ is false. On the other hand, $wp.CH.(\exists i : \mathbb{N} \bullet p_i(x'))$ is true. This is proved below.

$$wp.CH.(\exists i : \mathbb{N} \bullet p_i(x'))$$
$$\equiv (\forall x' : \mathbb{Z} \bullet x' > 0 \Rightarrow \exists i : \mathbb{N} \bullet p_i(x')) \qquad \text{[by definition of } wp\text{]}$$
$$\equiv (\forall x' : \mathbb{Z} \bullet x' > 0 \Rightarrow \exists i : \mathbb{N} \bullet x' < i) \qquad \text{[by definition of } p_i(x')\text{]}$$
$$\equiv \text{true} \qquad \text{[by a property of } <\text{]}$$

So, $wp.CH.(\exists i : \mathbb{N} \bullet p_i(x'))$ is not equivalent to $(\exists i : \mathbb{N} \bullet wp.CH.p_i(x'))$ and therefore, $wp.CH$ is not continuous.

In the next section we introduce a number of theorems that help in calculating the weakest precondition of some schema expressions.

2.3 Schema Expressions

A Z schema can be specified by an expression of the schema calculus and, although we can calculate the weakest precondition of every schema that specifies an operation by first expanding it to the form $\langle d;\ d';\ di?;\ do! \mid p\rangle$, ideally we should be able to express and calculate the weakest precondition of a schema expression compositionally. Unfortunately, wp does not distribute nicely through most schema operators. In what follows, we present a number of results that can be applied in some particular cases.

The theorem below presents a compositional formulation of wp which can be obtained in the case of a schema disjunction, if the disjuncts are operations over the same state and with the same inputs and outputs.

Theorem 2.8 *For all schemas Op_1 and Op_2 that specify operations over the same state and with the same inputs and outputs, and for every postcondition ψ,*

$$\begin{aligned} & wp.(Op_1 \lor Op_2).\psi \equiv \\ & \quad (wp.Op_1.\text{true} \lor wp.Op_2.\text{true}) \land \\ & \quad (wp.Op_1.\text{true} \Rightarrow wp.Op_1.\psi) \land (wp.Op_2.\text{true} \Rightarrow wp.Op_2.\psi) \end{aligned}$$

The operation $Op_1 \lor Op_2$ terminates if either Op_1 or Op_2 does. When Op_1 (similarly Op_2) terminates, $Op_1 \lor Op_2$ can behave just like it and, in this case, $Op_1 \lor Op_2$ is guaranteed to establish ψ only if $Op_1(Op_2)$ is. Of course, when both Op_1 and Op_2 terminate, then $Op_1 \lor Op_2$ can behave like either of them and therefore both have to guaranteedly establish ψ. A proof for Theorem 2.8 and for some other theorems we introduce in this section can be found in Appendix B.

As implication can be expressed in terms of disjunction, we can, based on Theorem 2.8, formulate the weakest precondition of a schema implication compositionally.

Theorem 2.9 *For all schemas Op_1 and Op_2 that specify operations over the same state and with the same inputs and outputs, and for every postcondition ψ,*

$$\begin{aligned}&wp.(Op_1 \Rightarrow Op_2).\psi \equiv\\ &\quad (wp.\neg\, Op_1.\text{true} \vee wp.Op_2.\text{true}) \wedge\\ &\quad (wp.\neg\, Op_1.\text{true} \Rightarrow wp.\neg\, Op_1.\psi) \wedge (wp.Op_2.\text{true} \Rightarrow wp.Op_2.\psi)\end{aligned}$$

provided Op_1 is normalised.

Requiring Op_1 to be normalised is necessary as, otherwise, the schema expressions $Op_1 \Rightarrow Op_2$ and $\neg\, Op_1 \vee Op_2$ are not equivalent, as expected.

Since $p_1 \Leftrightarrow p_2$ is equivalent to $(p_1 \wedge p_2) \vee \neg\,(p_1 \vee p_2)$, we can express $wp.(Op_1 \Leftrightarrow Op_2)$ in terms of $wp.(Op_1 \wedge Op_2)$ and $wp.\neg\,(Op_1 \vee Op_2)$, if Op_1 and Op_2 are normalised.

Theorem 2.10 *For all schemas Op_1 and Op_2 that specify operations over the same state and with the same inputs and outputs, and for every postcondition ψ,*

$$\begin{aligned}&wp.(Op_1 \Leftrightarrow Op_2).\psi \equiv\\ &\quad (wp.(Op_1 \wedge Op_2).\text{true} \vee wp.\neg\,(Op_1 \vee Op_2).\text{true}) \wedge\\ &\quad (wp.(Op_1 \wedge Op_2).\text{true} \Rightarrow wp.(Op_1 \wedge Op_2).\psi) \wedge\\ &\quad (wp.\neg\,(Op_1 \vee Op_2).\text{true} \Rightarrow wp.\neg\,(Op_1 \vee Op_2).\psi)\end{aligned}$$

provided Op_1 and Op_2 are normalised.

Existential quantifications that are applied to and yield schemas that define operations are considered in the next theorem.

Theorem 2.11 *For every schema Op that specifies an operation, all declarations d, d', $di?$, and $do!$ that introduce components of Op, and every postcondition ψ,*

$$\begin{aligned}&wp.(\exists\, d;\ d';\ di?;\ do! \bullet Op).\psi \equiv\\ &\qquad (\exists\, d;\ di? \bullet wp.Op.\text{true}) \wedge (\forall\, d;\ di? \bullet wp.Op.\text{true} \Rightarrow wp.Op.\psi)\end{aligned}$$

provided the variables of αd, $\alpha d'$, $\alpha di?$, and $\alpha do!$ do not occur free in ψ.

Hiding is a special form of existential quantification. The schema $Op \backslash \alpha d, \alpha d', \alpha di?, \alpha do!$ is equivalent to $\exists\, d;\ d';\ di?;\ do! \bullet Op$, so that, under the restrictions imposed on d; d'; $di?$; $do!$, and ψ in Theorem 2.11, $wp.(Op \backslash \alpha d, \alpha d', \alpha di?, \alpha do!).\psi \equiv wp.(\exists\, d;\ d';\ di?;\ do! \bullet Op).\psi$. Actually, αd, $\alpha d'$, $\alpha di?$, and $\alpha do!$ are sets of variable names and what the hiding operator takes as argument is a comma-separated list of variable names. Nonetheless, we allow ourselves this minor abuse of notation and assume that $\alpha d, \alpha d', \alpha di?, \alpha do!$ does denote a list of the variables declared in d, d', $di?$, and $do!$.

Schema projection can be defined in terms of conjunction and hiding. The schema defined by $Op_1 \upharpoonright Op_2$ is equivalent to $(Op_1 \wedge Op_2) \backslash \alpha d, \alpha d', \alpha di?, \alpha do!$, where d, d', $di?$, and $do!$ declare the components of Op_1 that are not components of Op_2.

Theorem 2.12 *For all schemas Op_1 and Op_2 that specify operations, and every postcondition ψ,*

$$\begin{aligned}&wp.(Op_1 \upharpoonright Op_2).\psi \equiv\\ &\quad(\exists\, d;\ di? \bullet wp.(Op_1 \wedge Op_2).\mathrm{true}) \wedge\\ &\quad(\forall\, d;\ di? \bullet wp.(Op_1 \wedge Op_2).\mathrm{true} \Rightarrow wp.(Op_1 \wedge Op_2).\psi)\end{aligned}$$

where d and $di?$ declare the state components and the input variables of Op_1 that are not components of Op_2. We assume that all components of Op_1 that are not components of Op_2 are not free in ψ.

The form of a schema renaming is $S[nv/ov]$, where S is a schema, and nv and ov are lists of variables. The schema $S[nv/ov]$ is obtained from S by substituting the variables of nv for the corresponding ones in ov. We consider the case $Op[ns, ns', ni?, no!/os, os', oi?, oo!]$, where renaming is applied to a schema that specifies an operation and produces another schema that specifies an operation. As expected, os' (ns') is the list of variables obtained by dashing the variables of os (ns).

Theorem 2.13 *For every schema Op that specifies an operation, all lists of variables os, $oi?$, $oo!$, ns, $ni?$, and $no!$ without duplicates, and every postcondition ψ where the variables of os, os', $oi?$, and $oo!$ do not occur free,*

$$\begin{aligned}&wp.Op[ns, ns', ni?, no!/os, os', oi?, oo!].\psi \equiv\\ &\quad(wp.Op.\psi[os, os', oi?, oo!/ns, ns', ni?, no!])[ns, ni?/os, oi?]\end{aligned}$$

We assume that the variables of ns, ns', $ni?$, and $no!$ are not components of Op; and that the variables of os, $oi?$, ns, ns', $ni?$, and $no!$ do not occur as global variables in Op.

If we rename the components of a schema, we can calculate its weakest precondition with respect to a postcondition ψ by expressing ψ in terms of the original component names and calculating the weakest precondition of the original schema with respect to this postcondition. The resulting predicate is expressed in terms of the original state and input variables, which then have to be renamed.

The schema expression called generic schema designator has the form $S[e_1, e_2, \ldots, e_n]$, where S is the name of a generic schema with n parameters and $e_1, e_2, \ldots, e_n$ are set-valued expressions. Since the parameters of a generic definition are used as ordinary given sets, we can ignore the parameters of a generic schema Op that specifies an operation and calculate its weakest precondition as if it were an ordinary schema. The next theorem shows how this result can help in calculating the weakest precondition of $Op[e_1, e_2, \ldots, e_n]$.

Theorem 2.14 *For every generic schema designator $Op[e_1, e_2, \ldots, e_n]$, where Op is a generic schema that specifies an operation and has parameters $x_1, x_2, \ldots, x_n$; and for every postcondition ψ where $x_1, x_2, \ldots, x_n$ do not occur free.*

$$wp.Op[e_1, e_2, \ldots, e_n].\psi \equiv (wp.Op.\psi)[e_1, e_2, \ldots, e_n/x_1, x_2, \ldots, x_n]$$

provided the components of Op are not free in $e_1, e_2, \ldots, e_n$.

If the weakest precondition of Op is known, the weakest precondition of $Op[e_1, e_2, \ldots, e_n]$ can be obtained simply by substituting $e_1, e_2, \ldots, e_n$ for the occurrences of the corresponding parameters of Op in this predicate.

2.4 Conclusions

With tbe objective of formalising ZRC, we have presented a weakest precondition semantics for Z equivalent to the relational semantics defined in [8], which is an official document of the Z standardisation committee. Actnally, we have constructed a *wp* semantics for Z based on this relational semantics. The outcoming definition is neither complex nor surprising, but its calculation provides evidence for its adequacy and is itself of interest.

An isomorphism between a relational model and weakest preconditions has been established. This relational model is along the lines of that presented in [28], and is used in [65] to formalise the data refinement rules of Z. A connection between it and the relational model assigned to schemas that specify operations in [8] has been presented as well. This is the link to the standard Z semantics that is missing in [65].

In [26], an isomorphism between a relational model and a predicate transformer model based on weakest preconditions and weakest liberal preconditions (*wlp*) is established. In this work, the behaviour of operations in states where they may fail to terminate is of interest, hencc the nse of weakest liberal preconditions. Correspondingly, the relational model adopted there does not satisfy our healthiness condition (2.3). Moreover, $\perp$ (or ∞, as it is called in [26]) is not in the domain of the relations that are considered there, so that (2.4) is not necessary. As far as predicate transformers are concerned, the healthiness conditions imposed in [26] restrict the model to universally conjunctive weakest liberal preconditions, and relate *wp* and *wlp*. Together, these healthiness conditions imply that *wp* is positively conjunctive, which is our healthiness condition (2.1). Since the postconditions of [26] specify states, as opposed to state transitions, our healthiness condition (2.2) is not an issue there.

An innovative aspect of our work is to formalise the Oxford-style of specifying operations using *wp*. The only other formalisation that we are aware of is the relational work in [65]. On the otber hand, as opposed to the relational semantics, the weakest precondition semantics of Z does not define it completely: while the relational semantics ascribes a meaning to all its syntactic strnctures, the *wp* semantics is restricted to a subset of the syntactic category *Schema*. As already noted, however, the motivation for the definition of a *wp* semantics for Z was not the provision of an alternative account of its semantics, but the formalisation of ZRC.

Chapter 3

ZRC

In this chapter we present ZRC: its language (ZRC-L), its conversion and refinement laws, and its formalisation. Most conversion laws are based on those of [34, 64]; we give them a uniform presentation in a style closer to the Z notation. The refinement laws are, on the whole, based on those of Morgan's calculus. Again, in order to conform to the Z style, adjustments have been necessary and, in several cases, we adopt refinement laws similar to those of [65]. Furthermore, we introduce additional conversion and refinement laws.

Our main enterprise, however, has been the formalisation of ZRC. Apparently, there has been no effort to establish the soundness of the translation rules of [34, 64], and so formalisation is a distinctive attribute of ZRC.

Most of this work is based on the formalisation of Morgan's calculus [47, 45]. Nonetheless, due to an inconsistency we have found in [41], we adopt Back's approach [3] in our treatment of proeedures and parameters. Additionally, we formalise the use of variants presented in [45]; the refinement laws of ZRC concerned with the development of recursive procedures support the technique suggested in Morgan's calculus and have no equivalent there. Our approach to data refinement is based on that of [46], which, as shown in [39], is more general than that of [45], which is based on the auxiliary variable technique.

Section 3.1 provides an informal description of ZRC-L; its weakest precondition semantics is presented in Appendix C. In the previous chapter we have considered the semantics of Z. In Section 3.2, we discuss the semantics of several constructs of ZRC-L that are not part of the Z notation, and in Section 3.3 we formalise our notion of refinement.

The semantics of procedures, parameters, and recursion is considered separately in Section 3.4. There we examine a connection between Morgan's and Back's formalisms and the substitution operator that renames the free variables of a program, and unveil an inconsistency in Morgan's work. Furthermore, we define the semantics of the variaut blocks used in Morgan's approach to recursion [45]. Much of the material in this section also appears in [11].

In Section 3.5, we define the scope rules of ZRC-L. The conversion laws are presented in Section 3.6, where we also exemplify their application. Section 3.7 discusses the refinement laws; for the sake of conciseness, we focus on those related to the development of procedures and data refinement. Appendix D lists all conversion and refinement laws and their derivations. Finally, Section 3.7 summarises the results obtained and discusses a few related works.

3.1 ZRC-L

As with the languages of several other refinement techniques, ZRC-L is an extension of Dijkstra's language of guarded commands. It includes additional statements so that specifications as well as programs can be written and more sophisticated program design mechanisms can be used. Specifications, in particular, can be expressed in ZRC-L with the use of the Z notation. However, a number of statements which express specifications in a form better suited for refinement are also available. One of them is the specification statement, which has been presented in Section 1.1. The other ones are assumption and coercion, which can be regarded as special specification statements.

The state components, or rather, the before-state variables, and those that are introduced by variable blocks, are collectively named program variables. This is in contrast with Morgan's calculus where the variables that represent the before-state are 0-subscripted and called initial variables. In ZRC, as in Z, the after-state variables are those that are decorated.

The frame of a specification statement can contain only program and output variables. Since preconditions do not characterise state changes, they cannot contain free occurrences of after-state variables. There is here a significant difference from the use of 0-subscripted variables in [45, 47]. It is not only the case that we decorate a different set of variables, but also we use the initial (in our terminology, the program) variables to write the preconditions.

The program **skip** can be considered as an abbreviation for $: [\mathrm{true}, \mathrm{true}]$. It does not change any variable, as it has an empty frame, and always terminates. Assumptions and coercions, which are called annotations, can also be viewed as specification statements with empty frames. An assumption $\{pre\}$ corresponds to the specification statement $: [pre, \mathrm{true}]$, which acts as **skip** if executed from a state that satisfies pre, and aborts otherwise. A coercion $[post]$ corresponds to $: [\mathrm{true}, post]$. If executed from a state that satisfies $post$, it acts as **skip** as well, but otherwise it is a miracle, as it establishes $post$ without modifying any variable. Programs of the form $\{pre\}\ ;\ p$ and $[post]\ ;\ p$ can be written as $\{pre\}\ p$ and $[post]\ p$, respectively.

A miracle is a program that is miraculous at some initial states so that, as already explained in Chapter 2, it can achieve any postcondition when executed from these states. These programs violate the law of excluded miracle; they cannot be refined by any executable program. Miracles may arise by mistake during the refinement process, but they may be useful, as shown in [40, 42].

A variable block has the form $[\![\,\mathbf{var}\ dvl \bullet p\,]\!]$, where dvl declares variables with no decoration that may be referred to in the program p along with their dashed counterparts. We assume that program variables, their dashed counterparts, and input and output variables are not free in declarations.

The design of programs may require the use of logical constants. These can be introduced by a constant block of the form $[\![\,\mathbf{con}\ dcl \bullet p\,]\!]$, where dcl declares logical constants and p is a program. As opposed to variable blocks, constant blocks are not executable and have to be eliminated during refinement.

Procedures, possibly recursive, are declared in blocks as well. In order to illustrate the notation we employ to write procedure blocks, we consider the example below.

$$[\![\,\mathbf{proc}\ Inc \mathrel{\widehat=} x := x + 1 \bullet Inc\ ;\ Inc\,]\!]$$

This very simple program uses the procedure Inc to increase the value of x by 2. The program $x := x + 1$ is the body of Inc, and $Inc\ ;\ Inc$ is the main program (the scope of the procedure).

The general form of a procedure block is $[\![$ **proc** $pn \mathrel{\widehat=} body \bullet mp$ $]\!]$, where pn is a procedure name; $body$ is indeed a procedure body: a program or a parametrised statement (a construct we introduce below), and finally mp is a main program.

Parametrised procedures can be defined with the use of parametrised statements. These can have the form (**val** $dvl \bullet p$), (**res** $dvl \bullet p$), or (**val-res** $dvl \bullet p$), which correspond to the traditional conventions of parameter passing known as call-by-copy: call-by-value, call-by-result, and call-by-value-result, respectively. In each case, dvl declares the formal parameters, and p is a program.

As opposed to assignments, for instance, parametrised statements are not programs by themselves. Nevertheless, a parametrised statement (or the name of a procedure whose body is a parametrised statement) can be applied to a list of actual parameters to yield a program which acts as that obtained by passing the actual parameters to the program in the body of the parametrised statement. The number of actual parameters must be the same as the number of formal parameters; the correspondence between them is positional. As an example, we present the procedure block below.

$$[\![\ \textbf{proc}\ Inc \mathrel{\widehat=} (\textbf{val-res}\ n : \mathbf{N} \bullet n := n + 1) \bullet Inc(x)\ ;\ \ Inc(y)\,]\!]$$

This program increments the variables x and y using a parametrised procedure Inc.

Parametrised statements whose parameters use different mechanisms of transmission can be defined as well. For instance, (**val** $x : \mathbf{N}$; **val-res** $y : \mathbf{Z} \bullet y := y + x$) has a value parameter x of type $\mathbf{N}$ and a value-result parameter of type $\mathbf{Z}$.

In the case of a call-by-result or a call-by-value-result, the list of actual parameters must be duplicate-free. In [41, 3, 45] this list is also supposed to contain only variables. In ZRC-L, however, we allow for function applications as well. The idea is that, if the function is implemented by an array, then the function application corresponds to an array indexing, which in most programming languages is acceptable as an actual parameter irrespective of the mechanism of parameter transmission used. This generalisation is needed in the treatment of promotion (see Section 3.6).

The development of recursive procedures requires the use of variants. Recursion may be used if the refinement of a program (parametrised statement) p leads to another program (parametrised statement) that contains p itself as a component. Due to termination concerns, however, the introduction of recursion requires the definition of a variant: an integer expression whose value must be decreased by each recursive call, but cannot assume negative values (cf. iteration variant in Section 1.1). From a theoretical point of view, the type of a variant can be any well-founded set, but in practice it is enough to consider that the variant is an integer bounded below by 0.

As suggested in [45], a variant is declared in a new kind of procedure block called a variant block. Its form is $[\![$ **proc** $pn \mathrel{\widehat=} body$ **variant** vrt **is** $e \bullet mp$ $]\!]$, where vrt is a name for the variant expression e. As constant blocks, variant blocks are not executable and have to be refined away. By way of illustration, we consider the program that assigns to the variable x the factorial of y, $x : [\text{true}, x' = y!\,]$. If, when refining this program, we decide that we want to develop a recursive procedure that implements the factorial function, we have to introduce a variant block like that presented below, which declares a procedure $Fact$.

$$\begin{array}{l}[\![\ \textbf{proc}\ Fact \mathrel{\widehat=} (\textbf{val}\ n : \mathbf{N} \bullet \{N = n\}\ x : [\text{true}, x' = n!\,])\ \textbf{variant}\ N\ \textbf{is}\ n \bullet \\ \quad x : [\text{true}, x' = y!\,] \\ ]\!]\end{array}$$

At this point, we can refine the body of $Fact$ to obtain a recursive implementation for this procedure, and refine the main program to introduce the appropriate procedure call. The variant N plays the role of a logical constant in the body of *Fact*. The assumption $\{N = n\}$ in the body of this parametrised statement fixes the initial value of N as being n. Recursive calls may be introduced only at points where this value has been strictly decreased. In Section 3.7.2 we return to this example and show in detail how *Fact* can he refined to a recursive procedure.

In Sections 3.2 and 3.4 we consider the weakest precondition semantics of the statements we have informally introduced in this section, and of a few others. The entire set of definitions that compose the *wp* semantics of ZRC-L is presented in Appendix C.

3.2 Primitive Statements, Composition, Variables, and Constants

In this section we discuss the *wp* semantics of the primitive statements (specification statement, **skip**, assignment, etc.) of ZRC-L, of sequential composition, and of the variable and constant blocks. Our definition of the alternation semantics is the same as that in [47]. Procedures and recursion are considered in Section 3.4. The semantics of iteration is defined in terms of recursion in the usual way.

In the formalisation of Morgan's calculus [42, 45], the weakest precondition of a specification statement is defined as shown below, where vl is the list of all variables, and vl_0 the list of corresponding 0-subscripted (initial) variables.

$$wp.w : [pre, post].\psi \equiv pre \wedge (\forall\, w \bullet post \Rightarrow \psi)[vl/vl_0] \tag{3.1}$$

In comparison, onr definition is as follows.

Definition 3.1 *For every postcondition ψ with no free program variables,*

$$wp.w : [pre, post].\psi \equiv pre \wedge (\forall\, dw' \bullet post \Rightarrow \psi)[\_/']$$

where dw declares the variables of w.

This definition considers the type of the variables in the frame when quantifying over them. Since the frame lists the program variables that can he modified, instead of their dashed counterparts, we have to consider the declaration dw' instead of dw. What we are using is a decoration operation that applies to declarations. In general, the declaration d_s differs from d just in the names of the variables that it declares: the alphabet of d_s can be obtained by appending the symbol "s" to the names of the non-decorated variables in the alphabet of d.

The purpose of the substitution in Definition 3.1 is the same as that in (3.1): to eliminate the variable decorations. The predicate $p[\_/']$ is that obtained by removing the dashes of the free variables of p. More precisely, $p[\_/']$ is an abbreviation for $p[vl/vl']$, where vl is the list of all program variables and vl' is the list of variables obtained by dashing the variahles in vl. In more general terms, for every list of variahles l, l' can be obtained by dashing the non-decorated variables in l. The list vl of program variables, in particular, does not contain any decorated variables and so, vl' is the result of dashing all variables in vl.

Definition 3.1 and other weakest precondition definitions presented in the sequel contemplate only postconditions that do not contain free occurrences of program variables. Nevertheless, as already remarked in Chapter 2, *wp* semantics relies on the principle that the meaning of a program is precisely specified only if, for every postcondition ψ, the preconditions that ensure termination in a state that satisfies ψ can be characterised. Therefore, we must be able to calculate weakest preconditions with respect to postconditions that are expressed in terms of program variables as well. For this reason we introduce another definition.

Definition 3.2 *For every program p and postcondition ψ, including those that contain free occurrences of program variables,*

$$wp.p.\psi \equiv (wp.p.\psi[cl/vl])[vl/cl]$$

where vl is the list of all program variables and cl is a list of fresh constants, none of which is free in p or ψ.

The weakest precondition of $x : [x \geq 0, x' = \sqrt{x}\,]$ with respect to $x' = x$, for instance, is $x = 1$, as expected. On the other hand, according to [42, p. 11], $wp.x : [x \geq 0, x = \sqrt{x_0}\,].x = x_0$ is $x \geq 0 \wedge \sqrt{x} = x_0$, with x_0 as an ordinary constant or variable which might as well have been called y. Definitions 3.1 and 3.2 together formalise the use of decorations both in the postconditions of specification statements and in postconditions of wp.

The free names (variables, logical constants, etc.) of a program are precisely identified in Section 3.5. Informally, these are the names that are not bound by a declaration. We must note, however, that not only the variables explicitly introduced by a variable block, but also their dashed counterparts, are bound in this program. Also, we regard both the components and the global variables of a schema as its free variables. Nonetheless, if p is a schema, and nv and ov lists of variables, $p[nv/ov]$ is not a substitution, but a renaming, which affects the components of p only.

Theorem 3.1 below shows that Definitions 2.6 and 3.2 are not in contradiction.

Theorem 3.1 *For every schema $\langle d;\ d';\ di?;\ do! \mid p\rangle$ and postcondition ψ,*

$$wp.\langle d;\ d';\ di?;\ do! \mid p\rangle.\psi \equiv (wp.\langle d;\ d';\ di?;\ do! \mid p\rangle.\psi[cl/\alpha d])[\alpha d/cl]$$

where cl is a list of fresh constants, which are not free in $\langle d;\ d';\ di?;\ do! \mid p\rangle$ and ψ.

Proof

$$\begin{array}{ll}
& wp.\langle d;\ d';\ di?;\ do! \mid p\rangle.\psi \\
\equiv & (\exists\, d';\ do! \bullet p) \wedge (\forall\, d';\ do! \bullet p \Rightarrow \psi) \qquad \text{[by Definition 2.6]} \\
\equiv & (\exists\, d';\ do! \bullet p) \wedge (\forall\, d';\ do! \bullet p \Rightarrow \psi[cl/\alpha d][\alpha d/cl]) \qquad \text{[by } cl \text{ are not free in } \psi] \\
\equiv & ((\exists\, d';\ do! \bullet p) \wedge (\forall\, d';\ do! \bullet p \Rightarrow \psi[cl/\alpha d]))[\alpha d/cl] \\
& \qquad \text{[by } cl \text{ are not free in } d', do!, \text{ and } p, \text{ and } \alpha d \text{ are not in } \alpha d' \text{ and } \alpha do!] \\
\equiv & (wp.\langle d;\ d';\ di?;\ do! \mid p\rangle.\psi[cl/\alpha d])[\alpha d/cl] \qquad \text{[by Definition 2.6]}
\end{array}$$

□

As defined in Chapter 2, αd is the set of variables declared by d, or rather, its alphabet. In the above theorem αd is used in substitutions as a list of variables. For the sake of simplicity, we employ this notation whenever the order in which the variables of the alphabet of a declaration is listed is not relevant. In Theorem 3.1, for example, it is not necessary to determine the particular fresh constant of cl that replaces each of the variables in the alphabet of d. Nonetheless, we assume that the list of variables denoted by αd is always the same and, also, that αd_s lists the alphabet of the decorated declaration d_s in the same order as αd lists the corresponding variables in the alphabet of d. In this way, the predicate $p[\alpha d'/\alpha d]$, for instance, is that obtained by dashing the free occurrences of the variables of the alphabet of d in p.

In [43, 45], types are treated as special forms of invariants. However, the notion of invariant in these works and that of Z are different. While in [43] invariants compose a context for programs, in Z the operations themselves contain the invariant as part of their specification. As the variable blocks with invariant and the invariant blocks of [43, 45] are not considered here, types are treated directly in ZRC: the type declarations are regarded as axioms, as in [2]. Moreover, as we have already seen, the weakest precondition of schemas and specification statements take the types of the variables into account, and so do the other weakest precondition definitions that follow.

Annotations and **skip** can be explained in terms of specification statements, as noted in the previous section. These interpretations explain their weakest precondition semantics, which we present below.

Definition 3.3 *For every postcondition ψ' with no free program variables,*

$$wp.\mathbf{skip}.\psi' \equiv \psi$$
$$wp.\{pre\}.\psi' \equiv pre \land \psi$$
$$wp.[post].\psi' \equiv post[\_/'] \Rightarrow \psi$$

The predicate ψ', or more generally, a predicate p' is that obtained by dashing the free program variables of p; p' is an abbreviation for $p[vl'/vl]$, where vl is the list of program variables.

The weakest precondition of assignments is defined below.

Definition 3.4 *For every postcondition ψ' with no free program variables,*

$$wp.vl := el.\psi' \equiv \psi[el/vl]$$

If $vl := el$ is to establish ψ', then this predicate must hold when the variables of vl' assume the values denoted by the corresponding expressions of el and all other variables assume the values of their undashed counterparts. This is exactly the property characterised by $\psi[el/vl]$.

Sequential compositions are considered in the definition that follows.

Definition 3.5 *For every postcondition ψ with no free program variables,*

$$wp.(p_1\ ;\ p_2).\psi \equiv wp.p_1.(wp.p_2.\psi)'$$

Usually, sequential composition is defined by weakest precondition composition. In our case, an intermediary substitution is necessary, because postconditions are expressed in terms of dashed variables, and weakest preconditions, in terms of program variables.

The weakest precondition of a variable block is usually defined just for postconditions in which the variables that it declares are not free. As far as the calculation of weakest preconditions is concerned, this restriction introduces no loss of generality since these variables are bound in the variable block and, therefore, can be renamed in case of clash. Nonetheless, proofs are usually carried out under the assumption that they are not free in the postconditions involved and it might not be entirely clear why this assumption is legitimate. We clarify this point by proposing a more general definition for $wp.\ [\![\ \mathbf{var}\ dvl \bullet p\ ]\!]\ .\psi$ and introducing theorems that back up the usual assumption later in Section 3.3.

Definition 3.6 *For every postcondition ψ with no free program variables,*

$$wp.\,[\![\,\mathbf{var}\ dvl \bullet p\,]\!]\,.\psi \equiv \forall\, dl \bullet wp.p[l, l'/vl, vl'].\psi$$

provided dvl and dl declare the variables of vl and l, respectively, and differ just in the names of the variables that they declare; and the names of l and l' are not free in p and ψ.

By way of illustration, we calculate the weakest precondition of $[\![\,\mathbf{var}\ x : \mathbb{N} \bullet x := 1\,]\!]$ with respect to $x' = 1$. In this example, we are working with different variables of name x. According to Definition 3.6, we must, as shown below, rename the occurrences of x inside the variable block.

$$
\begin{aligned}
& wp.\,[\![\,\mathbf{var}\ x : \mathbb{N} \bullet x := 1\,]\!]\,.x' = 1 \\
\equiv\ & \forall\, y : \mathbb{N} \bullet wp.(x := 1)[y, y'/x, x'].x' = 1 && \text{[by Definition 3.6]} \\
\equiv\ & \forall\, y : \mathbb{N} \bullet wp.y := 1.x' = 1 && \text{[by a property of substitution]} \\
\equiv\ & \forall\, y : \mathbb{N} \bullet (x = 1)[1/y] && \text{[by definition of } wp\text{]} \\
\equiv\ & x = 1 && \text{[by predicate calculus]}
\end{aligned}
$$

Surprising as it might be, this result is in accordance with the fact that $[\![\,\mathbf{var}\ x : \mathbb{N} \bullet x := 1\,]\!]$ is equivalent to **skip**, since it does not change any external variable and always terminates.

Theorem 3.2 shows that when only postconditions that do not contain free occurrences of the names in vl and vl' are considered, Definition 3.6 coincides the with the usual definition of the weakest precondition of a variable block. The proof of this theorem relies on Lemma 3.1. This lemma establishes that, if we systematically change the names of the free variables of a program and of a postcondition without causing any clashes, then the result of applying wp to them is not altered, except only for the names of its free variables.

Lemma 3.1 *For all lists of variables l and vl, and for every program p,*

$$wp.p.\psi \equiv (wp.p[l, l'/vl, vl'].\psi[l, l'/vl, vl'])[vl/l]$$

provided the names of l and l' are not free in p and ψ.

Proof Structural induction over p.

□

As an example we take the programs $x : [\text{true}, x' = x + 1]$ and $y : [\text{true}, y' = y + 1]$, observing that the latter can be obtained by substituting in the former y and y' for x and x', respectively. Similarly, we consider the postconditions $x' > 0$ and $y' > 0$. It is not difficult to see that $wp.x : [\text{true}, x' = x + 1].x' > 0 \equiv x > -1$ and that $wp.y : [\text{true}, y' = y + 1].y' > 0 \equiv y > -1$, and so $wp.y : [\text{true}, y' = y + 1].y' > 0 \equiv (wp.x : [\text{true}, x' = x + 1].x' > 0)[y/x]$.

Theorem 3.2 *For every postcondition ψ in which neither program variables nor names of vl and vl' are free,*

$$wp.\,[\![\,\mathbf{var}\ dvl \bullet p\,]\!]\,.\psi \equiv \forall\, dvl \bullet wp.p.\psi$$

provided dvl declares the variables of vl.

Proof

$wp.\,[\![\,\mathbf{var}\ dvl \bullet p\,]\!]\,.\psi$

$\equiv \forall\, dl \bullet wp.p[l, l'/vl, vl'].\psi$ [by Definition 3.6]

$\equiv \forall\, dl \bullet wp.p[l, l'/vl, vl'].\psi[l, l'/vl, vl']$ [by vl and vl' are not free in ψ]

$\equiv \forall\, dvl \bullet (wp.p[l, l'/vl, vl'].\psi[l, l'/vl, vl'])[vl/l]$

[by vl are not free in $wp.p[l, l'/vl, vl'].\psi[l, l'/vl, vl']$ (by Theorem 3.9)]

$\equiv \forall\, dvl \bullet wp.p.\psi$ [by Lemma 3.1]

□

For example, we can use Theorem 3.2 to calculate the weakest precondition of the variable block $[\![\,\mathbf{var}\ x : \mathbf{N} \bullet x := 1\,]\!]$ with respect to $y' = 1$ in a much simpler way than that dictated by Definition 3.6.

$wp.\,[\![\,\mathbf{var}\ x : \mathbf{N} \bullet x := 1\,]\!]\,.y' = 1$

$\equiv \forall\, x : \mathbf{N} \bullet wp.x := 1.y' = 1$ [by Theorem 3.2]

$\equiv \forall\, x : \mathbf{N} \bullet (y = 1)[1/x]$ [by definition of wp]

$\equiv y = 1$ [by predicate calculus]

In this case, since neither x nor x' are free in the postcondition, the variable x declared by $[\![\,\mathbf{var}\ x : \mathbf{N} \bullet x := 1\,]\!]$ does not have to be renamed.

Our definition of the constant block semantics also generalises its usual definition.

Definition 3.7 *For every postcondition ψ with no free program variables,*

$wp.[\![\,\mathbf{con}\ dcl \bullet p\,]\!]\,.\psi \equiv \exists\, dl \bullet wp.p[l/cl].\psi$

provided dcl and dl declare the constants of cl and l, respectively, and differ just in the names of the constants that they declare; and the names of l and l' are not free in p and ψ.

The generalisation follows the same lines used above in the case of variable blocks. If just postconditions not containing free occurrences of the names of cl and cl' are taken into account, Definition 3.7 is equivalent to the usual weakest precondition definition of constant blocks. This result is established by Theorem 3.3. Before introducing this theorem, we present a lemma that is used in its proof.

Lemma 3.2 *For all lists of constants l and cl, and for every program p,*

$wp.p.\psi \equiv (wp.p[l/cl].\psi[l/cl])[cl/l]$

provided the names of l and l' are not free in p and ψ.

Proof Structural induction over p.

□

This lemma is similar to Lemma 3.1. In this case, the systematic change of names of constants, rather than variables, is considered.

Theorem 3.3 *For every postcondition ψ in which neither program variables nor names of cl and cl' are not free,*

$$wp.[\![\,\mathbf{con}\ dcl \bullet p\,]\!].\psi \equiv \exists\, dcl \bullet wp.p.\psi$$

provided dcl declares the constants of cl.

Proof

$$\begin{aligned}
& wp.[\![\,\mathbf{con}\ dcl \bullet p\,]\!].\psi \\
\equiv\ & \exists\, dl \bullet wp.p[l/cl].\psi && \text{[by Definition 3.7]} \\
\equiv\ & \exists\, dl \bullet wp.p[l/cl].\psi[l/cl] && \text{[by } cl \text{ are not free in } \psi] \\
\equiv\ & \exists\, dcl \bullet (wp.p[l/cl].\psi[l/cl])[cl/l] && \text{[by } cl \text{ are not free in } wp.p[l/cl].\psi[l/cl] \text{ (by Theorem 3.9)]} \\
\equiv\ & \exists\, dcl \bullet wp.p.\psi && \text{[by Lemma 3.2]}
\end{aligned}$$

□

If we take the constant block $[\![\,\mathbf{con}\ c : \mathbb{N} \bullet x : [x = c, x' = c + 1]\ ]\!]$, then we can use Theorem 3.3 to calculate its weakest precondition with respect to $x' = 1$ as follows.

$$\begin{aligned}
& wp.[\![\,\mathbf{con}\ c : \mathbb{N} \bullet x : [x = c, x' = c + 1]\ ]\!].x' = 1 \\
\equiv\ & \exists\, c : \mathbb{N} \bullet wp.x : [x = c, x' = c + 1].x' = 1 && \text{[by Theorem 3.3]} \\
\equiv\ & \exists\, c : \mathbb{N} \bullet x = c \land c = 0 && \text{[by definition of } wp] \\
\equiv\ & x = 0 && \text{[by predicate calculus]}
\end{aligned}$$

As $x' = 1$ contains no free occurrence of c or c', no renaming is necessary.

In Chapter 2, we have shown that, from the healthiness conditions pointed out in [14], just continuity is not satisfied when weakest preconditions of schemas are considered. On the other hand, when programs are taken into account as well, wp satisfies only monotonicity. The law of excluded miracle is not satisfied by specification statements, and $\land$-distributivity is not satisfied by constant blocks.

As already mentioned, the semantics of procedures and recursion is the subject of Section 3.4. In the next section we formalise the notion of refinement adopted in ZRC.

3.3 Refinement

The definition that we adopt for $\sqsubseteq$, the refinement relation, is that in [47], which embodies the concept of total correctness.

Definition 3.8 *For all programs p_1 and p_2, $p_1 \sqsubseteq p_2$ if and only if, for all postconditions ψ,*

$$wp.p_1.\psi \Rightarrow wp.p_2.\psi$$

Intuitively $p_1 \sqsubseteq p_2$ exactly when p_2 terminates whenever p_1 does, and produces only results that are acceptable to p_1. Therefore, if $p_1 \sqsubseteq p_2$, then p_2 is always satisfactory as a substitute for p_1.

A more formal justification of Definition 3.8 can be found in [42, 4].

The derivations of the conversion and refinement laws of ZRC rely on Definition 3.8 and, consequently, consist of establishing implications between weakest preconditions. The theorems that follow allow us to assume that the postconditions involved in these proofs satisfy certain restrictions concerning their sets of free variables. These assumptions simplify the proofs and are also exploited in the formalisation of Morgan's calculus.

The first theorem allows us to consider only postconditions that do not contain free program variables.

Theorem 3.4 *If, for every postcondition ψ that does not contain free program variables, we have that $wp.p_1.\psi \Rightarrow wp.p_2.\psi$, then $wp.p_1.\delta \Rightarrow wp.p_2.\delta$ for every postcondition δ.*

Proof

$$
\begin{array}{ll}
wp.p_1.\delta & \\
\equiv (wp.p_1.\delta[cl/vl])[vl/cl] & \text{[by definition of } wp\text{]} \\
\Rightarrow (wp.p_2.\delta[cl/vl])[vl/cl] & \text{[by assumption]} \\
\equiv wp.p_2.\delta & \text{[by definition of } wp\text{]}
\end{array}
$$

□

The corollary below is useful if we want to prove that p_1 is equal to p_2 (their weakest preconditions are equivalent), and not only refined by it.

Corollary 3.1 *If, for every postcondition ψ that does not contain free program variables, we have that $wp.p_1.\psi \equiv wp.p_2.\psi$, then $wp.p_1.\delta \equiv wp.p_2.\delta$, for every postcondition δ.*

The proof of this corollary is a straightforward application of Theorem 3.4.

In the case where p_1 is a variable block, Theorem 3.5 is of use as well. Provided neither the variables introduced by p_1 nor their dashed counterparts are free in p_2 (which is often the case), only postconditions that do not contain free occurrences of these variables have to be examined. For these postconditions, the simpler definition of the weakest precondition of a variable block introduced by Theorem 3.2 applies.

Theorem 3.5 *Suppose that, for every postcondition ψ in which program variables and the names of vl and vl' are not free, $wp.\ |[\,\mathbf{var}\ dvl \bullet p\,]|\ .\psi \Rightarrow wp.p_2.\psi$. Then $wp.\ |[\,\mathbf{var}\ dvl \bullet p\,]|\ .\delta \Rightarrow wp.p_2.\delta$, for every postcondition δ with no free program variables, provided dvl declares the variables of vl and the names of vl and vl' are not free in p_2.*

Proof If the names of vl (and vl') are in scope as variables, the proof is as follows.

$$
\begin{array}{ll}
wp.\ |[\,\mathbf{var}\ dvl \bullet p\,]|\ .\delta & \\
\equiv (wp.\ |[\,\mathbf{var}\ dvl \bullet p\,]|\ [l, l'/vl, vl'].\delta[l, l'/vl, vl'])[vl/l] & \text{[by Lemma 3.1]} \\
\equiv (wp.\ |[\,\mathbf{var}\ dvl \bullet p\,]|\ .\delta[l, l'/vl, vl'])[vl/l] & \text{[by a property of substitution]} \\
\Rightarrow (wp.p_2.\delta[l, l'/vl, vl'])[vl/l] & \text{[by assumption]}
\end{array}
$$

$$\equiv (wp.p_2[l, l'/vl, vl'].\delta[l, l'/vl, vl'])[vl/l] \qquad \text{[by } vl \text{ and } vl' \text{ are not free in } p_2]$$
$$\equiv wp.p_2.\delta \qquad \text{[by Lemma 3.1]}$$

If the names of vl are in scope as constants (in which case, the variables of vl' are not in scope), the proof is similar, but uses Lemma 3.2, instead of Lemma 3.1.

□

A similar result holds when p_2 is a variable block.

In the event that p_1 is a constant block, Theorem 3.6 allows us to confine our attention to postconditions in which the constants that it introduces and the dashed variables named after them are not free. In this way we can make use of Theorem 3.3 which gives a simple but restricted definition of constant blocks.

Theorem 3.6 *Suppose that, for every postcondition ψ in which program variables and the names of cl and cl' are not free, $wp.|[\,\mathbf{con}\ dcl \bullet p\,]|.\psi \Rightarrow wp.p_2.\psi$. Then $wp.|[\,\mathbf{con}\ dcl \bullet p\,]|.\delta \Rightarrow wp.p_2.\delta$, for every postcondition δ with no free program variables, provided dcl declares the constants of cl, and the names of cl and cl' are not free in p_2.*

Proof Similar to that of Theorem 3.5.

□

A similar theorem covering the case in which p_2 is a constant block can be proved. Appendix D presents many law derivations which make use of the theorems listed above.

An important result about $\sqsubseteq$, which can be easily proved by structural induction, is that it distributes through the program structure: the program constructors are monotonic with respect to it. This means that the different components of a program can be refined separately.

As shown later on in Section 3.7.3, a possible way of refining a variable block is by data refinement. This consists of replacing the variables that it declares and modifying its body accordingly. The objective is to rewrite the program using data structures which, for instance, can be more efficiently implemented. The variables declared in the original variable block are called abstract, and those declared in the new variable block, concrete. A data refinement relation characterises the programs that can replace the body of the original variable block.

Our definition for this relation has been suggested by the work in [33], where a proof-obligation expressed in terms of weakest preconditions characterises data refinement between schemas that specify operations. We adopt the notation of [46] and write $p_1 \preccurlyeq_{avl,cvl,ci} p_2$ to mean that p_1 is data-refined by p_2. The lists of variables avl and cvl enumerate the abstract and concrete variables, respectively; ci is the coupling invariant, a predicate that specifies how the abstract and concrete variables are related. The concrete variables must be fresh.

Definition 3.9 *For all programs p_1 and p_2, lists of variables avl and cvl, and predicate ci, $p_1 \preccurlyeq_{avl,cvl,ci} p_2$ if and only if*

$$ci \wedge wp.p_1.\psi \Rightarrow wp.p_2.\exists\, davl' \bullet ci' \wedge \psi$$

for all postconditions ψ in which the variables of cvl and cvl' are not free. The declaration davl introduces the variables of avl. The variables of cvl and cvl' must not be free in p_1. Moreover, avl and cvl must be disjoint.

When avl, cvl, and ci can be deduced from the context, we write the data refinement relation simply as $\preccurlyeq$.

For reassurance, we observe that if both p_1 and p_2 are schemas, $p_1 \preccurlyeq_{avl,cvl,ci} p_2$ holds exactly when the corresponding Z proof-obligations for data refinement can be discharged. This relationship is precisely formulated in Theorem 3.7.

Theorem 3.7 *For all schemas A and C that define, respectively, an abstract and a concrete state; schema R that defines a retrieve relation between A and C; and schemas Op_1 and Op_2,*

$$R \land wp.Op_1.\psi \Rightarrow wp.Op_2.(\exists A' \bullet R' \land \psi)$$

if and only if

$$\forall A;\ C \bullet \text{pre } Op_1 \land R \Rightarrow \text{pre } Op_2 \text{ and } \forall A;\ C;\ C' \bullet \text{pre } Op_1 \land R \land Op_2 \Rightarrow \exists A' \bullet Op_1 \land R'$$

A proof for this theorem is not provided here, since this result is not used further ahead.

The next theorem establishes that our definition coincides with that in [46, 50]. This result guarantees that our definition does not incur extra complexity to the derivation of the data refinement laws of ZRC.

Theorem 3.8 *For all programs p_1 and p_2, lists of abstract and concrete variables avl and cvl, and coupling invariant ci, $p_1 \preccurlyeq p_2$ if and only if*

$$(\exists\, davl \bullet ci \land wp.p_1.\psi) \Rightarrow wp.p_2.\exists\, davl' \bullet ci' \land \psi$$

for every postcondition ψ in which neither program variables nor variables of cvl′ are free. The variables of avl and avl′ must not be free in p_2. The declaration davl introduces the variables of avl. The variables of cvl and cvl′ must not be free in p_1. Moreover, avl and cvl must be disjoint.

Proof At first, suppose that $p_1 \preccurlyeq p_2$.

$\exists\, davl \bullet ci \land wp.p_1.\psi$

$\Rightarrow \exists\, davl \bullet wp.p_2.\exists\, davl' \bullet ci' \land \psi$ [by Definition 3.9]

$\equiv wp.p_2.\exists\, davl' \bullet ci' \land \psi$ [by avl are not free in $wp.p_2.\exists\, davl' \bullet ci' \land \psi$ (by Theorem 3.9)]

Conversely, suppose that $(\exists\, davl \bullet ci \land wp.p_1.\psi) \Rightarrow wp.p_2.\exists\, davl' \bullet ci' \land \psi$ holds for every postcondition ψ in which neither program variables nor variables of cvl' are free. For a postcondition δ in which the variables of cvl and cvl' are not free,

$ci \land wp.p_1.\delta$

$\equiv ci \land (wp.p_1.\delta[cl/vl])[vl/cl]$ [by definition of wp]

$\equiv (ci \land wp.p_1.\delta[cl/vl])[vl/cl]$ [by cl are not free in ci]

$\Rightarrow (\exists\, davl \bullet ci \land wp.p_1.\delta[cl/vl])[vl/cl]$ [by predicate calculus]

$\Rightarrow (wp.p_2.\exists\, davl' \bullet ci' \land \delta[cl/vl])[vl/cl]$ [by assumption]

$\equiv (wp.p_2.(\exists\, davl' \bullet ci' \land \delta)[cl/vl])[vl/cl]$

[by vl are not free in $davl$ and ci', and cl are not in avl']

$\equiv wp.p_2.\exists\, davl' \bullet ci' \land \delta$ [by definition of wp]

□

The characterisation of data refinement in this theorem considers just postconditions that do not contain free occurrences of program variables. The existence of this simpler specification of the data refinement relation is not surprising, since Corollary 3.1 and Theorem 3.1 establish that the semantics of a program is completely defined by the restriction of *wp* to postconditions with no free occurrences of program variables. The derivations of the data refinement laws of ZRC rely on Theorem 3.8, rather than on Definition 3.9 directly. These laws are discussed in Section 3.7.3 and listed in Appendix D.

3.4 Procedures, Parameters, and Recursion

Back [3] and Morgan [41] have both formalised the use of procedures and parameters in the context of refinement techniques. These works are connected in a perplexing and unanticipated way to the substitution operator that renames the free variables of a program. In this section we examine this relationship and give our reasons for adopting Back's approach in the treatment of procedures in ZRC; also in this section we define a semantics for procedures and recursion.

3.4.1 Exploring the Effect of Substitution

Both Back and Morgan adopt the copy rule of Algol 60 when defining a semantics for non-recursive procedures. According to this rule, a program that contains a procedure call is equivalent to that obtained by substituting the procedure body for the procedure name. Variable capture must be avoided, in order to ensure that the scope of variables is static. As an example, we take the procedure block below, which has been presented in Section 3.1.

$\llbracket\, \mathbf{proc}\ Inc \mathrel{\widehat=} x := x + 1 \bullet Inc \ ;\ Inc \,\rrbracket$

As expected, this program is equivalent to $x := x + 1 \ ;\ x := x + 1$.

In order to illustrate the concerns related to the capture of variables, we consider the program below which assigns 1 to a global variable x.

$$\llbracket\, \mathbf{proc}\ P \mathrel{\widehat=} x := 1 \bullet \llbracket\, \mathbf{var}\ x : \mathbb{N} \bullet P \,\rrbracket \,\rrbracket \tag{3.2}$$

In the main program of this procedure block, the variable block $\llbracket\, \mathbf{var}\ x : \mathbb{N} \bullet P \,\rrbracket$ declares a variable x local to its body, P. Since there is a variable x free in the body of P, we cannot, as in the previous example, apply the copy rule and substitute $x := 1$ for P. This substitution would lead to the capture of the global variable x mentioned in the body of P by the local declaration of x in the main program and, therefore, would violate the rules of static scope. Before applying the copy rule to (3.2), we have to rename the local variable x.

The refinement law *vrbR* (variable renaming), which is presented and derived in Appendix D, can be used to rename the variables introduced by a variable block. By applying this law, we conclude that, since z is not free in P, (3.2) is equivalent to $\llbracket\, \mathbf{proc}\ P \mathrel{\widehat=} x := 1 \bullet \llbracket\, \mathbf{var}\ z \bullet P[z/x] \,\rrbracket \,\rrbracket$. At this point, our main concern is the result of $P[z/x]$.

There seems to be two acceptable possibilities: P and $z := 1$. In the first case, the substitution operator acts on the name P and, since x is clearly not free in this program, P is itself the result; substitution is a syntactic operator, and is referred to as syntactic substitution. In the second case, the substitution operator acts on the body of P and yields the result of substituting z for x

in that program. The behaviour of substitution is dependent on the context in which it is applied, and this operator is referred to as context dependent substitution. This somewhat unusual form of substitution is adopted in [55] and, as we explain later, is part of a possible solution to the problems we uncover here.

Both forms of substitution can be defined by recursion in the usual way. The interesting part of their definitions is that concerned with the application of substitution to a procedure name. In the case of syntactic substitution we have that, for a procedure name pn and lists of variables vl_1 and vl_2, $pn[vl_2/vl_1] = pn$. For context dependent substitution, if p is the body of the procedure pn, then $pn[vl_2/vl_1] = p[vl_2/vl_1]$. The main purpose of this section is to show that either definition of substitution leads to inconsistency in Morgan's formalisation of procedures and parameters. Moreover, we show that Back's formalisation presents no problems, provided we adopt syntactic substitution.

If we assume that $P[z/x]$ is $z := 1$ (context dependent substitution), we can deduce that $|[\,\mathbf{var}\ z : \mathbf{N} \bullet P[z/x]\ ]|$ is equivalent to $|[\,\mathbf{var}\ z : \mathbf{N} \bullet z := 1\,]|$. This program, however, can be shown to be equivalent to **skip**: it does not change any variable other than the local variable that it introduces, and always terminates. Overall, we can prove that (3.2), which is supposed to assign 1 to x, is equivalent to **skip**.

In [55], Sampaio avoids this problem by restricting the application of the law that accounts for the renaming of the variables declared by a variable block. He defines the notion of contiguous scope and establishes that the renaming is possible only if the variables have a contiguous scope in the body of the variable block. A variable is said to have a contiguous scope in a program if either this program does not contain procedure calls or the variable is not free in the bodies of the procedures that are called. In the above example, since x does not have a contiguous scope in P, because x is free in the body of this procedure, we cannot deduce, according to [55], that $|[\,\mathbf{var}\ x : \mathbf{N} \bullet P\,]| = |[\,\mathbf{var}\ z : \mathbf{N} \bullet P[z/x]\ ]|$. Consequently, the undesired deduction that we have presented cannot be carried out.

Although it might be considered a solution to the problem, we cannot adopt this approach if we accept the usual *wp* semantics of variable blocks presented in [3, 47, 45] or that adopted in ZRC, which is similar. In all these models, the equality $|[\,\mathbf{var}\ x : \mathbf{N} \bullet P\,]| = |[\,\mathbf{var}\ z : \mathbf{N} \bullet P[z/x]\ ]|$ can be deduced, provided z is a fresh variable. As a consequence, if we assume context dependent substitution, the undesired deduction can be carried out in these models.

Sampaio gives an algebraic semantics for the language introduced in [55]. In this semantics, the restricted renaming law that he proposes (based on the notion of contiguous scope) is regarded as an axiom and no model is presented for the language.

In summary, if we discard the possibility of changing the semantics of variable blocks, we have to assume that $P[z/x]$ is equal to P, or, in more general terms, that, substitution is a syntactic operator. This is the form of substitution adopted by both Back and Morgan, and adopted in ZRC-L as well.

While this decision avoids the problem discussed above, we show below that it leads to another problem in Morgan's formalisation of parameters [41, 45]. In his calculus, the use of parametrised procedures is made possible by substitutions which define both the formal and actual parameters of a procedure at the point(s) of call rather than definition. The forms of substitution available correspond to call-by-value, call-by-result, and call-by-value-result. For example, a substitution by result has the form $p[\mathbf{result}\ vl_2/vl_1]$, where p is the program to which it applies, vl_1, the list of formal parameters, and vl_2, the list of actual parameters. The main program of the procedure block below, for instance, is composed by two substitutions by result.

$$|[\,\mathbf{proc}\ Zero \mathrel{\widehat{=}} n := 0 \bullet Zero[\mathbf{result}\ x/n]\ ;\ Zero[\mathbf{result}\ y/n]\ ]|$$

This program assigns 0 to the variables x and y using a procedure *Zero*.

In [41] Morgan provides a *wp* semantics for substitutions, but they can also be defined in terms of variable blocks. For instance, from the weakest precondition of a substitution by result, we can derive the equality below, where l is a list of fresh variables.

$$p[\mathbf{res}\ vl_2/vl_1] = |[\,\mathbf{var}\ l \bullet p[l/vl_1]\ ;\ vl_2 := l\,]| \tag{3.3}$$

The variable block above implements a call-by-result using the well-known technique of assignment from a local variable. This is the definition actually adopted in [45].

In order to explain the problem with this approach to parametrised procedures, we consider the program that assigns 1 to a variable z using the procedure P of (3.2).

$$|[\,\mathbf{proc}\ P \mathrel{\widehat{=}} x := 1 \bullet P[\mathbf{result}\ z/x]\ ]| \tag{3.4}$$

As we show below, in view of our comments about procedures and result substitutions, and assuming that syntactic substitution is adopted, we can get to an unexpected conclusion. Namely, this procedure block is equivalent to $|[\,\mathbf{var}\ l \bullet x := 1\ ;\ z := l\,]|$, a program that assigns 1 to x and an arbitrary value (that assigned to l upon declaration) to z.

$$\begin{aligned}
&|[\,\mathbf{proc}\ P \mathrel{\widehat{=}} x := 1 \bullet P[\mathbf{result}\ z/x]\ ]| \\
&= |[\,\mathbf{proc}\ P \mathrel{\widehat{=}} x := 1 \bullet |[\,\mathbf{var}\ l \bullet P[l/x]\ ;\ z := l\,]|\ ]| && [\text{by } (3.3)] \\
&= |[\,\mathbf{proc}\ P \mathrel{\widehat{=}} x := 1 \bullet |[\,\mathbf{var}\ l \bullet P\ ;\ z := l\,]|\ ]| && [\text{by } P[l/x] = P] \\
&= |[\,\mathbf{var}\ l \bullet x := 1\ ;\ z := l\,]| && [\text{by the copy rule}]
\end{aligned}$$

It might appear that an immediate solution to this problem is to adopt context dependent rather than syntactic substitution. In this case, $P[l/x]$, in the second line of the above derivation, would be replaced by $l := 1$ (rather than by P), and the overall result of the derivation would be $z := 1$, as expected. Nevertheless, we have just concluded that substitution must be regarded as a syntactic operator, since otherwise we run into the problem raised earlier in this section.

If we apply the copy rule at an earlier stage, before replacing the result substitution by the variable block defined by (3.3), the resulting program assigns 1 to z as well. The development is shown below.

$$\begin{aligned}
&|[\,\mathrm{proc}\ P \mathrel{\widehat{=}} x := 1 \bullet P[\mathbf{result} z/x]\ ]| \\
&= x := 1[\mathbf{result}\ z/x] && [\text{by the copy rule}] \\
&= |[\,\mathbf{var}\ l \bullet (x := 1)[l/x]\ ;\ z := l\,]| && [\text{by } (3.3)] \\
&= |[\,\mathbf{var}\ l \bullet l := 1\ ;\ z := l\,]| && [\text{by a property of substitution}] \\
&= z := 1 && [\text{by properties of assignments and variable blocks}]
\end{aligned}$$

In conclusion, the order in which the laws are applied influences the result.

In [45], Morgan uses the strategy illustrated by our second development above. However, the laws that can be derived from the model of procedures and substitutions provided in [41, 45] do not enforce the application of this strategy: the order of application used in the first development is also supported by this model.

Altogether, whatever definition we adopt for the substitution operator, we run into problems if we consider Morgan's formalisation of procedures. If we assume that $P[z/x]$ is $z := 1$ or, in other words, context dependent substitution, we run into the problem illustrated by our first example. Alternatively, if we assume that $P[z/x]$ is P (syntactic substitution), the problem posed by our second example comes about. This problem is not specific to result substitutions: similar difficulties would arise if we had used value or value-result substitutions.

In Back's approach, which we adopt in ZRC, this problem does not occur if we define that the substitution operator is syntactic. In his work, as in ZRC, parametrised procedures are defined with the use of parametrised statements. Their application to actual parameters is defined in terms of variable blocks. For example, call-by-result is defined by the equation below, where l is a list of variables that are not free in p, and are not in vl_2.

$$(\mathbf{res}\ vl_1 \bullet p)(vl_2) = |[\,\mathbf{var}\ l \bullet p[l/vl_1]\ ;\ vl_2 := l\,]|$$

The right-hand side of this equation is identical to that of (3.3).

Using Back's parametrised statements, we can write the procedure block (3.4) in the following way: $|[\,\mathbf{proc}\ P \mathrel{\widehat=} (\mathbf{res}\ x \bullet x := 1) \bullet P(z)\,]|$. Since the result of applying a procedure name to an actual parameter cannot be established without investigating the body of the procedure, when reasoning about $P(z)$, the only way to reduce it to a variable block is by first applying the copy rule. Consequently, within Back's framework, the unwanted deduction that could be carried out using Morgan's approach cannot arise.

As already explained, Sampaio avoids the undesired deductions we have presented by adopting context dependent substitution and introducing a notion of contiguous scope which is used to restrict the application of the law that renames local variables. If we assume that variables cannot be redeclared, or in other words, if we rule out the possibility of using nested scope, the variables always have a contiguous scope. In this case, the usual law that renames local variables can be applied without further constraints. This restriction over variable declarations, however, is generally too severe. Moreover, as Sampaio's formalisation of procedures and parameters is essentially the same as that of Back, he could have defined substitution as a syntactic operator, and then avoided the restriction imposed on the renaming law.

A negative aspect of Back's work is the introduction of an additional construct: the parametrised statement. Before laws can be derived, the notion of refinement has to be extended to these statements, and properties of the new refinement relation have to be proved. This is accomplished in the next section, where we also define the semantics of variant blocks.

3.4.2 Semantics

As already remarked in Section 3.1, the forms of parametrised statement that we consider in ZRC correspond to call-by-value, call-by-result, and call-by-value-result, instead of call-by-reference as in [3]. The semantics of these statements, or more precisely, of the programs obtained by applying them to lists of actual parameters is not surprising. The definitions are as follows.

Call-by-value $(\mathbf{val}\ dvl \bullet p)(el) = [\![\ \mathbf{var}\ dl \bullet l := el\ ;\ p[l, l'/vl, vl']\ ]\!]$ provided dvl and dl declare the variables of vl and l, respectively, and differ just in the names of the variables that they declare; and the names of l and l' are not free in p and el.

Call-by-result $(\mathbf{res}\ dvl_1 \bullet p)(vl_2) = [\![\ \mathbf{var}\ dl \bullet p[l, l'/vl_1, vl_1']\ ;\ vl_2 := l\ ]\!]$ provided dvl_1 and dl declare the variables of vl_1 and l, respectively, and differ just in the names of the variables that they declare; and the names of l and l' are not free in p, and are not in vl_2.

Call-by-value-result $(\mathbf{val\text{-}res}\ dvl_1 \bullet p)(vl_2) = [\![\ \mathbf{var}\ dl \bullet l := vl_2\ ;\ p[l, l'/vl_1, vl_1']\ ;\ vl_2 := l\ ]\!]$ provided dvl_1 and dl declare the variables of vl_1 and l, respectively, and differ just in the names of the variables that they declare; and the names of l and l' are not free in p, and are not in vl_2.

As mentioned in Section 3.1, in ZRC-L a function application can be the actual parameter of a call-by-result. The semantics of programs of this form is defined below in terms of ordinary calls.

Call-by-result (with a function application as actual parameter)

$$(\mathbf{res}\ v : t \bullet p)(f\ x) = [\![\ \mathbf{var}\ u : t \bullet (\mathbf{res}\ v : t \bullet p)(u)\ ;\ f := f \oplus \{x \mapsto u\}\ ]\!]$$

provided u and u' are not free in p.

The variable block that corresponds to a call-by-result whose actual parameter is a function application introduces a fresh auxiliary variable u. In this program, u is used as actual parameter, instead of the function application, and subsequently the function is updated accordingly. A call-by value-result can also take a function application as actual parameter. The definition of its semantics is similar to that of a call-by-result and is presented in Appendix C.

Parametrised statements that combine different forms of parameter transmission are defined by composition. For a mechanism of parameter passing par, and a formal parameter declaration fpd (either an ordinary declaration or a declaration that combines different forms of parameter transmission itself), we have the definition below.

Multiple parametrisation mechanisms

$$(\mathbf{par}\ dvl_1; fpd \bullet p)(el_1, el_2) = (\mathbf{par}\ dvl_1 \bullet (fpd \bullet p)(el_2))(el_1)$$

provided the variables declared by dvl_1 are not free in el_2.

This definition expresses in a general way the definitions of [3].

The definition of refinement that we adopt for parametrised statements is that of [3].

Definition 3.10 *For all parametrised statements $(fpd \bullet p_1)$ and $(fpd \bullet p_2)$, with the same formal parameter declaration, $(fpd \bullet p_1) \sqsubseteq (fpd \bullet p_2)$ if and only if, for all lists al of actual parameters, $(fpd \bullet p_1)(al) \sqsubseteq (fpd \bullet p_2)(al)$.*

Surprisingly, maybe, $p_1 \sqsubseteq p_2$ is not equivalent to $(fpd \bullet p_1) \sqsubseteq (fpd \bullet p_2)$. As shown in [3], parametrised statements are monotonic with respect to the refinement relation, so that $p_1 \sqsubseteq p_2$ implies $(fpd \bullet p_1) \sqsubseteq (fpd \bullet p_2)$. Nevertheless, there are cases in which $(fpd \bullet p_1) \sqsubseteq (fpd \bullet p_2)$, but $p_1 \sqsubseteq p_2$ does not hold. For instance, by applying the definition of call-by-value, it is not difficult to prove that both $(\mathbf{val}\ n : \mathbb{N} \bullet n := n + 1)(m)$ and $(\mathbf{val}\ n : \mathbb{N} \bullet n := n + 2)(m)$ are eqnivalent to skip. Therefore, we deduce that ($\mathbf{val}\ n : \mathbf{N} \bullet n := n+1$) $\sqsubseteq$ ($\mathbf{val}\ n : \mathbf{N} \bullet n := n+2$). Nevertheless, $n := n+1$ is not refined by $n := n+2$.

In order to simplify the presentation and derivation of the refinement laws, we establish that a parametrised statement may declare no formal parameters. These special parametrised statements are actually programs themselves and we define that, for every program p, $(\ \bullet\ p) = p$.

The main program of a procedure block and the body of a recursive procedure may refer to the procedure name, so they are not conventional programs. From the semantic point of view, they are contexts: functions from programs (parametrised statements) to programs (parametrised statements). If c is the main program of a procedure block or the body of a recursive procedure, it corresponds to the function that, when applied to a program (parametrised statement) p, yields the result of substituting p for the free occurrences of the procedure name in c.

The semantics of a procedure block, which we define below, is based on the copy rule of Algol 60 and on least fixed points, as adopted, for example, in [3, 47, 45].

Procedure block $|[\ \mathbf{proc}\ pn \mathrel{\widehat{=}} (fpd \bullet p_1)(pn) \bullet p_2(pn)\ ]| = p_2(\mu(fpd \bullet p_1))$

Since, as we have already said, programs can be seen as special parametrised statements, this definition (and others that follow) contemplates procedure blocks that declare either parametrised or non-parametrised procedures. It states that a procedure block is equivalent to the program obtained by applying its main program to the least fixed point of the procedure body. A more usual notation for the least fixed point of a function $(fpd \bullet p_1)$ is $\mu\, pn \bullet (fpd \bullet p_1)(pn)$, where $(fpd \bullet p1)(pn)$ is the application of the context $(fpd \bullet p_1)$ to the program pn (and not the application of the parametrised statement $(fpd \bullet p_1)$ to the actual parameter pn). Nonetheless, for the sake of brevity, we will adopt the more concise notation $\mu(fpd \bullet p_1)$.

The existence of $\mu(fpd \bullet p_1)$ has to be justified. According to Knaster-Tarski [60], we can establish that $\mu(fpd \bullet p_1)$ exists by showing that the set of programs and the sets of parametrised statements with the same formal parameter declaration are complete lattices, and that $(fpd \bullet p_1)$ is monotonic. As expected, the partial order of interest is refinement. Programs are modelled as monotonic predicate transformers, which, as shown in [7], form a complete lattice. The bottom element of the set of parametrised statements with formal parameter declaration fpd is $(fpd \bullet \mathbf{abort})$. Moreover, the least upper bound of a set of parametrised statements $\{i \bullet (fpd \bullet p_i)\}$ can be defined as $(\sqcup\{i \bullet (fpd \bullet p_i)\})(al) = \sqcup\{i \bullet (fpd \bullet p_i)(al)\}$, for all lists al of actual parameters. Finally, the program constructors are monotonic with respect to $\sqsubseteq$. Consequently, $(fpd \bullet p_1)$ is a monotonic function, as required.

In the case where $(fpd \bullet p_1)$ does not contain free occurrences of pn or, in other words, pn is not a recursive procedure, $\mu(fpd \bullet p_1)$ is $(fpd \bullet p_1)$ itself. Therefore, $|[\ \mathbf{proc}\ pn \mathrel{\widehat{=}} (fpd \bullet p_1) \bullet p_2(pn)\ ]|$ is the program obtained by substituting $(fpd \bullet p_1)$ for the calls to pn in p_2, $p_2(fpd \bullet p_1)$, as expected.

The variant name introduced by a variant block is a logical constant whose scope is restricted to the procedure body. This logical constant is supposed to assume ever decreasing values in the successive recursive procedure calls. Accordingly, the semantics of a variant block is as follows.

Variant block

$$|[\ \mathbf{proc}\ pn \mathrel{\widehat{=}} (fpd \bullet p_1)(pn)\ \mathbf{variant}\ n\ \mathbf{is}\ e \bullet p_2(pn)\ ]| = p_2(\mu(fpd \bullet |[\ \mathbf{con}\ n : \mathbf{Z} \bullet p_1\ ]|))$$

The fact that variant names are logical constants implies that, as mentioned before, variant blocks are not executable.

In order to consider data refinement of procedure and variant blocks, we have to extend the data refinement relation to contexts.

Definition 3.11 *For all contexts from programs to programs pc_1 and pc_2, lists of abstract and concrete variables avl and cvl, and predicate ci, $pc_1 \preccurlyeq pc_2$ if and only if, for all programs p_1 and p_2 such that $p_1 \preccurlyeq p_2$, $pc_2(p_1) \preccurlyeq pc_2(p_2)$.*

Actually, we have adopted the definition proposed in [18].

The definitions that we have introduced in this and in previous sections are used in Appendix D to derive conversion and refinement laws. In the next section we consider the scope rules of ZRC.

3.5 Scope Rules

The scope rules of ZRC-L are those that a programmer with experience in a langnage like Pascal, for instance, might expect. They are defined below by a fnnction (fn_p) that associates a program or a parametrised statement with the set of its free names.

The function fn_p is defined recursively. The function fn_{pd}, which is used in its definition, gives the free names of a predicate; fn_e defines the free names of an expression; finally, fn_d specifies the free names of a declaration: those that are free in the expressions that define the types of the variables in its alphabet. These functions are defined (under other names) in [8].

$fn_p.(d \mid p) = \alpha d \cup fn_d.d \;\cup\; fn_{pd}.p$

$fn_p.pn = \{pn\}$, where pn is a procedure name.

$fn_p.w : [pre, post] = elem.w \;\cup\; fn_{pd}.pre \;\cup\; fn_{pd}.post$

$fn_p.\mathbf{skip} = \varnothing$

$fn_p.\{pre\} = fn_{pd}.pre$

$fn_p.[post] = fn_{pd}.post$

$fn_p.vl := el = elem.vl \;\cup\; fn_e.el$

$fn_p.(p_1 \,;\, p_2) = fn_p.p_1 \;\cup\; fn_p.p_2$

$fn_p.\mathbf{if} \,[]\, i \bullet g_i \rightarrow p_i \;\mathbf{fi} = \bigcup\{\, i \bullet fn_{pd}.g_i \;\cup\; fn_p.p_i \,\}$

$fn_p.\mathbf{do} \,[]\, i \bullet g_i \rightarrow p_i \;\mathbf{od} = \bigcup\{\, i \bullet fn_{pd}.g_i \;\cup\; fn_p.p_i \,\}$

$fn_p.\, |[\, \mathrm{var}\; dvl \bullet p \,]| = fn_d.dvl \;\cup\; fn_p.p \backslash (\alpha dvl \;\cup \alpha dvl')$

$fn_p.|[\, \mathrm{con}\; dcl \bullet p \,]| = fn_d.dcl \;\cup\; fn_p.p \backslash \alpha dcl$

$fn_p.(fpd \bullet p) = fn_d.fpd \;\cup\; fn_p.p \backslash (\alpha fpd \cup \alpha fpd')$

$fn_p.|[\, \mathrm{proc}\; pn \mathrel{\widehat=} (fpd \bullet p_1) \bullet p_2 \,]| = (fn_p.(fpd \bullet p_1) \;\cup fn_p.p_2) \backslash \{pn\}$

$fn_p.|[\, \mathbf{proc}\; pn \mathrel{\widehat=} (fpd \bullet p_1) \;\mathbf{variant}\; n \;\mathbf{is}\; e \bullet p_2 \,]| =$
$\quad fn_d.fpd \cup fn_p.p_1 \backslash \{pn, n\} \cup fn_e.e \backslash \alpha fpd \cup fn_p.p_2 \backslash \{pn\}$

The function *elem* gives the set of elements of a list. The α function, which is usually applied to ordinary declarations to determine the set of variables that they declare, is applied above to formal parameter declarations as well.

A program or parametrised statement is well-scoped if its free variables are before or after-state, input, or output variables, or still global variables of the Z specification in which it is inserted. A well-scoped constant block, in particular, satisfies yet another restriction: the dashed variables named after the constants it introduces are not free in its body. Our definitions, theorems, lemmas, corollaries, and laws contemplate only well-scoped programs and parametrised statements.

The following theorem is extensively used throughout this work.

Theorem 3.9 *If a variable (constant) v and its dashed counterpart v' are not free in the program p and in the postcondition ψ, then v is not free in $wp.p.\psi$.*

Proof Structural induction over p.

□

As a matter of fact, this theorem has already been mentioned in proofs presented in the previous section. It is used again later in this chapter and in Appendix D.

The lemma below is also used in subsequent proofs.

Lemma 3.3 *For every program p, postcondition ψ, and predicate δ such that neither its free variables nor their undashed counterparts are free in p,*

$$(wp.p.\psi) \land \delta \Rightarrow wp.p.(\psi \land \delta')$$

Proof Structural induction over p.

□

Intuition might suggest that equality and not only implication holds in the above theorem. However, we should note that in the case where p is a miracle, it will establish δ' under any circumstances.

The next sections discuss the conversion and refinement laws of ZRC. They all take the scope rules of ZRC-L into account.

3.6 Conversion Laws

The majority of the ZRC conversion laws are based on those in [34, 64]. The most general of these laws, which can convert any schema that specifies an operation, is named bC (basic conversion). Its two formulations have already been used in Section 1.1. The first one is shown below: it can be applied to every schema of the form $(\Delta S;\ di?;\ do! \mid p)$ in order to translate it to an equivalent specification statement. Its derivation and those of all other ZRC conversion laws are presented in Appendix D.

Law bC Basic conversion

$$\begin{array}{ll} & (\Delta S;\ di?;\ do! \mid p) \\ = & bC \\ & \alpha d, \alpha do! : [inv \land \exists\, d';\ do! \bullet inv' \land p, inv' \land p] \end{array}$$

where $S \mathrel{\widehat=} (d \mid inv)$

The predicate $inv \land \exists\, d';\ do! \bullet inv' \land p$ is a conjunct of pre $(\Delta S;\ di?;\ do! \mid p)$, which also includes the restrictions over the state and input variables that are introduced by d and $di?$. Nevertheless, the methods for calculating pre $\langle \Delta S;\ di?;\ do! \mid p \rangle$ which are often employed in practice [66, 16, 65, 52] leave both inv and the restrictions of d and $di?$ implicit, and point out just $\exists\, d';\ do! \bullet inv' \land p$ as the precondition of $\langle \Delta S;\ di?;\ do! \mid p \rangle$. Consequently, when determining the result of applying bC to a particular schema that specifies an operation, we can rely on the calculations that establish its precondition.

By way of illustration, we consider the specification of a simple bank account presented in [16]. In this example, the state is formed by two components: bal and odl. They are both integers which represent, respectively, the balance and the overdraft limit of the acconnt.

$$\begin{array}{|l}
\textit{Account} \\
\hline
bal, odl : \mathbb{Z} \\
\hline
odl \geq 0 \\
odl + bal \geq 0 \\
\hline
\end{array}$$

An account overdrawn by an amount of money b has a negative balance: $-b$. The state invariant establishes that the overdraft limit of an account cannot be negative and cannot be overstepped.

We examine the operation *Withdraw* which withdraws money from the account. It has an input, $with?$, which determines the amount of money to be withdrawn.

$$\begin{array}{|l}
\textit{Withdraw} \\
\hline
\Delta Account \\
with? : \mathbb{Z} \\
\hline
0 < with? \leq odl + bal \\
bal' = bal - with? \\
odl' = odl \\
\hline
\end{array}$$

According to [66, 16, 65, 52], the precondition of *Withdraw* is $0 < with? \leq odl + bal$. The amount of money to be withdrawn must be positive and must be available in the account. The result of applying bC to *Withdraw* is the following specification statement.

$$bal, odl : \left[\left(\begin{array}{l} odl \geq 0 \\ odl + bal \geq 0 \\ 0 < with? \leq odl + bal \end{array} \right), \left(\begin{array}{l} odl' \geq 0 \\ odl' + bal' \geq 0 \\ 0 < with? \leq odl + bal \\ bal' = bal - with? \\ odl' = odl \end{array} \right) \right]$$

When bC is applied to a schema that specifies an operation, its precondition should preferably be already available in a simplified form. Nonetheless, this should not be seen as an extra burden of the refinement process. On the contrary, calculating the precondition of the operations is a recommended part of the specification phase [66, 65, 52] and, therefore, bC actually encourages the reuse of an existing result.

In [34] King writes specification statements using the 0-subscript convention for initial variables adopted in [47, 45]. We have followed, however, the lines of [64, 65] and kept the dashing convention of Z to maintain the compliance with this notation. Moreover, in [47, 45] the names of the variables are usually very short probably to make their copying easier. Following this guideline, King has suggested the shortening of the variable names of the Z specifications when translating them to the notation of the refinement calculus. Nevertheless, we believe that a tool can make all the necessary copying, understanding the designs is important, and the collected code gives guidance to the developer. Therefore, we have suppressed this shortening phase.

Schemas which specify operations that do not change the state or, more precisely, schemas of the form $\langle \Xi S;\ di?;\ do! \mid p \rangle$, can be written as $\langle \Delta S;\ di?;\ do! \mid p \wedge c_1' = c_1 \wedge \ldots \wedge c_n' = c_n \rangle$, where $c_1, \ldots, c_n$ are the components of S: the state components. As a consequence, we can use bC to transform $\langle \Xi S;\ di?;\ do! \mid p \rangle$ into a specification statement. Nonetheless, since schemas of this form occur very frequently in Z specifications, we propose an additional conversion law, or rather, an additional formulation of bC that considers their particular features.

Law bC Basic conversion (operations that do not modify the state)

$$\begin{array}{ll} & \langle \Xi S;\ di?;\ do! \mid p \rangle \\ = & bC \\ & \alpha do! : [inv \wedge \exists\, do! \bullet p[\alpha d/\alpha d'], p] \end{array}$$

where $S \mathrel{\widehat{=}} \langle d \mid inv \rangle$

The specification statements generated by this formulation of bC do not include the state components in their frames. Moreover, their postconditions do not enforce the maintenance of the invariant because, since the state components are not modified, the state invariant is necessarily maintained. The predicate $\exists\, do! \bullet p[\alpha d/\alpha d']$ is what is commonly regarded as the precondition of $\langle \Xi S;\ di?;\ do! \mid p \rangle$.

As an example, we consider the operation *Balance*, which retrieves the balance of the bank account. This operation does not change the state and has an output: *bal!*.

Balance

$\Xi Account$

$bal! : \mathbb{Z}$

$bal! = bal$

This is a total operation: its precondition is true. By applying the above formulation of bC to *Balance*, we get the specification statement below.

$$bal! : [odl \geq 0 \wedge odl + bal \geq 0, bal! = bal]$$

This is a much more concise result than that we would obtain if we used the first formulation of bC.

Every schema that specifies an operation can be written in a form appropriate to the application of bC and, therefore, this law can be used to transform any such schema into a specification statement. Nonetheless, before bC can be applied to a schema defined with the use of schema operators like conjunction, disjunction, and others, the schema expressions have to be expanded. As pointed out by King, however, some schema expressions can be translated directly into more structured programs.

Schema disjunctions can be transformed into alternations if the operations involved act over the same state and have the same inputs and outputs. In ZRC, this can be effected by an application of the conversion law *sdisjC* (schema disjunction conversion), which has three formulations. Its first formulation is shown below.

Law *sdisjC* Schema disjunction conversion

$$\begin{array}{ll} & Op_1 \lor Op_2 \\ \sqsubseteq & sdisjC \\ & \textbf{if } pre_1 \rightarrow Op_1 \, [\!] \; pre_2 \rightarrow Op_2 \textbf{ fi} \end{array}$$

where

- $\text{pre}\, Op_1 \equiv pre_1 \land inv \land t$;
- $\text{pre}\, Op_2 \equiv pre_2 \land inv \land t$;
- inv is the state invariant;
- t is the restriction that is introduced by the declarations of the state components and input variables.

Syntactic Restriction Op_1 and Op_2 act over the same state and have the same input and output variables.

The guards pre_1 and pre_2 are supposed to be the preconditions of Op_1 and Op_2 as calculated in [66, 16, 65, 52]. Their characterisation in terms of $\text{pre}\, Op_1$ and $\text{pre}\, Op_2$, however, does not uniquely identify them, since conjunction is idempotent. Nonetheless, any of the predicates pre_1 and pre_2 that satisfy this characterisation can be used as guards. The same observation holds for the other formulations of *sdisjC*, which also refer to pre_1 and pre_2.

As an example, we consider once again the bank account operation *Withdraw*, which is partial. If it takes as input an integer that is not positive or that represents an amount of money not available in the account, then its behaviour cannot be predicted. In the Oxford style of writing Z specifications, treatment of this error case consists of defining a robust operation using a disjunction like $(Withdraw \land Success) \lor WithdrawError$, where *WithdrawError* is a schema that specifies the effect of the operation in an error situation. Typically, *Success* is a schema as that we present below.

```
┌─ Success ──────────────────────────────
│ result! : MESSAGES
├──────────────
│ result! = ok
└────────────────────────────────────────
```

The set *MESSAGES* contains the possible error messages and *ok*, the message that signals success. The operation $Withdraw \land Success$ behaves as *Withdraw*, except that it has an output, *result!*, which indicates that the withdrawal has been successfully accomplished. Its precondition is that of *Withdraw* [65]. The schema *WithdrawError* that we present below defines that, in case of error, the state is not changed and the message *error* is output.

```
── WithdrawError ──────────────────────────
│ $\Xi Account$
│ $with? : \mathbb{Z}$
│ $result! : MESSAGES$
├──────────────
│ $with? \leq 0 \lor with? > odl + bal$
│ $result! = error$
└──────────────────────────────────────────
```

In the absence of error, $(Withdraw \land Success) \lor WithdrawError$ acts as $Withdraw \land Success$; otherwise, it behaves as *WithdrawError*. By applying *sdisjC* to this operation, we can get the alternation below.

$$\begin{array}{l} \textsf{if } 0 < with? \leq odl + bal \rightarrow Withdraw \land Success \\ [\,]\ with? \leq 0 \lor with? > odl + bal \rightarrow WithdrawError \\ \textbf{fi} \end{array}$$

The schemas $Withdraw \land Success$ and *WithdrawError* that compose the branches of this alternation can be converted to specification statements by the *bC* law. Later on we present another law, *sconjC* (schema conjunction conversion), which can also be applied to $Withdraw \land Success$.

The second formulation of *sdisjC* introduces a fresh boolean variable used to record the precondition of the first disjunct.

Law *sdisjC* Schema disjunction conversion with boolean variable introduction

$$\begin{array}{ll} & Op_1 \lor Op_2 \\ \sqsubseteq & sdisjC \\ & [\![\, \textbf{var}\ b : Boolean \bullet b : [\text{true}, b' \Leftrightarrow pre_1]\ ;\ \textsf{if}\ b \rightarrow Op_1\ [\,]\ pre_2 \rightarrow Op_2\ \textbf{fi} \,]\!] \end{array}$$

where

- pre $Op_1 \equiv pre_1 \land inv \land t$;
- pre $Op_2 \equiv pre_2 \land inv \land t$;
- inv is the state invariant;
- t is the restriction that is introduced by the declarations of the state components and input variables.

Syntactic Restrictions

- Op_1 and Op_2 act over the same state and have the same input and output variables;
- b and b' are not free in Op_1 and Op_2.

For example, $(Withdraw \land Success) \lor WithdrawError$ can be transformed into the variable block below using this formulation of $sdisjC$.

```
|[ var $succs : Boolean$ •
    $succs : [\text{true}, succs' \Leftrightarrow 0 < with? \leq odl + bal]$;
    if $succs \rightarrow Withdraw \land Success$
    [] $with? \leq 0 \lor with? > odl + bal \rightarrow WithdrawError$
    fi
]|
```

The specification statement registers the precondition of $Withdraw \land Succcss$ in the local variable $succs$. The $Boolean$ type is not one the Z primitive types and is not defined in the Z mathematical toolkit, but can be specified using the Z notation without difficulties. As a matter of fact, the formulation of $sdisjC$ above is a specialisation of that we present below, which introduces a fresh variable v of an arbitrary type t. Examples of the application of this formulation of $sdisjC$ can be found in the next chapter.

Law $sdisjC$ Schema disjunction conversion with variable introduction

$$Op_1 \lor Op_2$$
$$\sqsubseteq \quad sdisjC$$
$$|[\,\textbf{var}\ v : t \bullet v : [\text{true}, \phi[v'/v]\,]\ ;\ \textbf{if}\ \psi_1 \rightarrow \{\phi \land \psi_1\}\ Op_1 \,[]\ \psi_2 \rightarrow \{\phi \land \psi_2\}\ Op_2\ \textbf{fi}\,]|$$

provided

- $\phi \land (pre_1 \lor pre_2) \Rightarrow \psi_1 \lor \psi_2$
- $\phi \land (pre_1 \lor pre_2) \Rightarrow (\psi_i \Rightarrow pre_i)$ for $i = 1, 2$

where

- $\text{pre}\, Op_1 \equiv pre_1 \land inv \land t$;
- $\text{pre}\, Op_2 \equiv pre_2 \land inv \land t$;
- inv is the state invariant;
- t is the restriction that is introduced by the declarations of the state components and input variables.

Syntactic Restrictions

- ϕ, ψ_1, and ψ_2 are well-scoped and well-typed predicates;
- ϕ, ψ_1, and ψ_2 have no free dashed variables;
- Op_1 and Op_2 act over the same state and have the same input and output variables;
- v and v' are not free in Op_1 and Op_2.

This formulation of $sdisjC$ generalises the corresponding translation rule of [34]. The latter converts Op_1 and Op_2 to specification statements.

A schema conjunction may be translated into a sequential composition if the conjuncts act on different state components. This can be achieved by applying the conversion law *sconjC*.

Law *sconjC* Schema conjunction

$$Op_1 \wedge Op_2$$
$$\sqsubseteq \quad sconjC$$
$$Op_1 \;;\; Op_2$$

Syntactic Restriction Op_1 and Op_2 have no common free variables.

As an example, we take *Withdraw* $\wedge$ *Success*; by applying *sconjC* to this operation, we can get the sequential composition *Withdraw* ; *Success*. This method of translating schema conjuuctions was proposed in [34]. There, although the conversion procedure is clearly explained, the formulation of the law is mistaken. It was due to our effort to formalise ZRC that we uncovered this problem.

As a matter of fact, the relationship between the Z relational and weakest precondition semantics presented in Chapter 2 does not acconnt for schemas which, like *Success*, have no state components and initial variables. It is to be expected, however, that their weakest precondition satisfies the characterisatiou presented in Theorem 2.6 for schemas that specify operations in general and that, consequently, the laws of ZRC can be applied to them as well.

Due to the form of its predicate, *Success* can be translated to an assignment; the law *assC* (assignment conversion) can perform this task. ft is suggested but not actually formnlated in [34].

Law *assC* Assignment conversion

$$(\Delta S;\; di?;\; do! \mid c_1' = e_1 \wedge \ldots \wedge c_n' = e_n \wedge o_1! = e_{n+1} \ldots \wedge o_m! = e_{n+m})$$
$$\sqsubseteq \quad assC$$
$$c_1, \ldots, c_n, o_1!, \ldots, o_m! := e_1, \ldots, e_{n+m}$$

provided $inv[e_1, \ldots, e_n / c_1, \ldots, c_n]$

where

- $S \mathrel{\widehat=} (d \mid inv)$
- $c_1, \ldots, c_n$ are state components (elements of αd);
- $o_1!, \ldots, o_m!$ are ontput variables (elements of $\alpha do!$).

Syntactic Restriction $\alpha d'$ and $\alpha do!$ are not free in $e_1, \ldots, e_{n+m}$.

The syntactic restriction guarantees that no after-state or output variable is free in any of the expressions $e_1, \ldots, e_{n+m}$, and so, the equalities $c_1' = e_1, \ldots, c_n' = e_n, o_1! = e_{n+1} \ldots, o_m! = e_{n+m}$ can be established by $c_1, \ldots, c_n, o_1!, \ldots, o_m! := e_1, \ldots, e_{n+m}$. The proviso guarantees that this assignment maintains the state invariant. By applying *assC* to *Success*, we get *result*! := *ok*.

Tbe formulation of the next law, *scompC* (schema composition conversion), has been motivated by comments in [58, 52]. This law applies to schema compositions $Op_1 \mathbin{⨾} Op_2$, where Op_1 and Op_2 are operations that act over the same state and have no common output variables. Using *scompC*, we can translate a schema composition like this into a sequential (program) composition, as long as the preconditiou of Op_2 is guaranteed to be established by Op_1, provided the precondition of $Op_1 \mathbin{⨾} Op_2$ bolds. This restriction is enforced by the proviso. The restriction on the output variables is necessary because, if Op_1 and Op_2 have a common output variable, then $Op_1 \mathbin{⨾} Op_2$ produces an output that takes the specification of both Op_1 and Op_2 into account, and $Op_1\ ;\ Op_2$, an output that considers just the specification of Op_2.

Law *scompC* Schema composition conversion

$$\begin{array}{ll} & Op_1 \fatsemi Op_2 \\ \sqsubseteq & scompC \\ & Op_1\ ;\ Op_2 \end{array}$$

provided $(pre_C \land Op_1) \Rightarrow pre_2'$

where

- $pre_C \equiv \mathrm{pre}(Op_1 \fatsemi Op_2) \land inv \land t \land t_1 \land t_2$;
- $\mathrm{pre}\ Op_2 \equiv pre_2 \land inv \land t \land t_2$;
- inv is the state invariant;
- t, t_1, and t_2 are the restrictions that are introduced by the declarations of the state components, of the input variables of Op_1, and of the input variables of Op_2, respectively.

Syntactic Restrictions

- Op_1 and Op_2 act over the same state;
- Op_1 and Op_2 have no common output variables.

As in the case of *sdisjC*, pre_C and pre_2 are supposed to be what is commonly regarded as the precondition of $Op_1 \fatsemi Op_2$ and Op_2, respectively.

To illustrate the application of *scompC* we adapt the example of [58], a counter with a limit. Its state is formed by the components *value*, which records the current value of the counter, and *limit*, which records the positive natural number that limits the value of the counter.

Counter

$value, limit : \mathbb{N}$

$value \leq limit$
$limit > 0$

This counter has a reset (*Reset*) and an increment (*Inc*) operatiou.

Reset

$\Delta Counter$

$value' = 0$
$limit' = limit$

Inc

$\Delta Counter$

$value' = value + 1$
$limit' = limit$

The application of *scompC* to $Reset \fatsemi Inc$ yields the program $Reset\ ;\ Inc$. As $Reset \fatsemi Inc$ is a total operation, the proof-obligation generated is $Reset \Rightarrow value' < limit'$. Since $Reset$ specifies that $value' = 0$ and the state invariant guarantees that $limit' > 0$, we can conclude that $value' < limit'$ and the implication above is valid. This reflects the fact that, after resetting the counter, it is always possible to increment it.

In [64] Woodcock presents and derives a law that implements a promoted operation using a call-by-value-result. This work has motivated the formulation of *promC* (promotion conversion), a law that applies to a promoted operation $\exists \Delta L \bullet \Phi \wedge Op$. The operation Op acts over a (local) state L and has no inputs or outputs. The schema Φ defines a mixed operation: it acts on the local state L and on the global state G. The latter contains just one component: a function f from an arbitrary type X to L. The purpose of Φ is to specify how an operation on the local state can be used to update the global state. Its input, $x?$, identifies an element in the range of f.

Law *promC* Promotion conversion

$$\begin{array}{ll} & \exists \Delta L \bullet \Phi \wedge Op \\ \sqsubseteq & promC \\ & [\![\textbf{proc } pn \mathrel{\widehat=} (\textbf{val-res } r : L \bullet \langle r, r' : L \mid (inv \wedge inv' \wedge p)[r.x_i, r'.x_i / x_i, x_i'] \rangle) \bullet pn(f\ x?)\ ]\!] \end{array}$$

where

$$L \mathrel{\widehat=} \langle x_1 : t_1;\ \ldots;\ x_n : t_n \mid inv \rangle$$
$$Op \mathrel{\widehat=} \langle \Delta L \mid p \rangle$$
$$G \mathrel{\widehat=} \langle f : X \rightarrowtail\!\!\!\!\rightarrow L \rangle$$

$$\begin{array}{|l} \hline \Phi \\ \Delta G \\ \Delta L \\ x? : X \\ \hline x? \in \operatorname{dom} f \\ \theta L = f\ x? \\ \{x?\} \mathbin{\lhd\!\!\!-} f' = \{x?\} \mathbin{\lhd\!\!\!-} f \\ f'\ x? = \theta L' \\ \hline \end{array}$$

Syntactic Restriction pn, r, and f are not free in Op.

The procedure block introduced by *promC* declares a procedure pn that implements the local operation Op. This procedure has a value-result parameter r whose type is L: that of the elements in the range of f. The body of pn is not Op itself, but a corresponding operation that acts on the state formed by the single component r, instead of on L. The substitution $(inv \wedge inv' \wedge p)[r.x_i, r'.x_i / x_i, x_i']$ replaces all references to the components of the original state in the predicate of Op with references to the corresponding components of r or r'. The main program consists only of a procedure call with the actual parameter $f\ x?$.

By way of illustration, we consider the specification of a registration and booking system of a holiday playscheme for children presented in [52]. In this example, a child representation is defined by a schema named *Child* which records its age and other details that may be relevant to the system: a value of the given set $CHILDINFO$.

$$\begin{array}{|l}
\textit{Child} \\
\hline
age : 5 \mathinner{\ldotp\ldotp} 16 \\
details : CHILDINFO \\
\hline
\end{array}$$

We define an operation *Birthday*, which increases the age of the child by 1.

$$\begin{array}{|l}
\textit{Birthday} \\
\hline
\Delta Child \\
\hline
age' = age + 1 \\
details' = details \\
\hline
\end{array}$$

The registration system associates identifiers to children. Its specification uses another given set, $CHILDID$, which contains all child identifiers.

$$\begin{array}{|l}
\textit{ChildReg} \\
\hline
creg : CHILDID \pfun Child \\
\hline
\end{array}$$

The following mixed operation is used to promote operations on *Child* to act on a particular child registered in the system.

$$\begin{array}{|l}
\Phi 1 C \\
\hline
\Delta ChildReg \\
\Delta Child \\
c? : CHILDID \\
\hline
c? \in \operatorname{dom} creg \\
\theta Child = creg\ c? \\
\{c?\} \ndres creg' = \{c?\} \ndres creg \\
creg'\ c? = \theta Child' \\
\hline
\end{array}$$

The operation that updates the age of the child $creg\ c?$ can be specified by the schema expression $\exists \Delta Child \bullet \Phi 1 C \land Birthday$. By applying the conversion law *promC* to this promoted operation, we can obtain the procedure block below.

$$\begin{array}{l}
[\![\, \mathbf{proc}\ bd \mathrel{\widehat=} (\mathbf{val\text{-}res}\ c : Child \bullet \langle c, c' : Child \mid c'.age = c.age + 1 \land c'.details = c.details \rangle) \bullet \\
\quad bd(creg\ c?) \\
]\!]
\end{array}$$

Two additional conversion laws are presented in Chapter 4. In the next section, we discuss the refinement laws of ZRC.

3.7 Refinement Laws

The refinement laws of ZRC are presented and derived in Appendix D. They are similar to the corresponding laws of Morgan's calculus that deal with initial variables, but some modifications have been necessary to take the Z decorations into account. As a consequence, in many cases the ZRC refinement laws correspond more closely to those in [65]. The application of several of these laws has already been illustrated in Section 1.1. Many more examples can be found in Chapter 4. Here, we discuss the refinement laws concerned with the development of procedures and data refinement.

In the next section, we present laws that can be used to introduce procedure and variant blocks, and procedure calls. These laws have no counterpart in [45] and support Morgan's technique of procedure development. In Section 3.7.2, we present refinement laws that can be used to introduce parametrised statements. These correspond to the laws of [44] that introduce substitutions. Finally, Section 3.7.3 contemplates the data refinement laws, which are based on [46], instead of [45].

3.7.1 Procedures and Recursion

The refinement law that can be employed to introduce a procedure block is presented below.

Law *prcI* Procedure introduction

$$\begin{aligned} & p_2 \\ = \;& prcI \\ & |[\,\textbf{proc}\ pn \mathrel{\widehat=} (fpd \bullet p_1) \bullet p_2\,]| \end{aligned}$$

Syntactic Restrictions

- pn is not free in p_2;
- $(fpd \bullet p_1)$ is well-scoped and well-typed.

This law allows any program p_2 to be transformed into a block that declares a procedure pn not called in p_2, and whose main program is p_2 itself. Calls to pn can be introduced subsequently in p_2 using the *pcallI* (procedure call introduction) law presented later on.

As mentioned in Section 3.4, a program can be regarded as a parametrised statement with an empty formal parameter declaration. Therefore, even though the body of pn is presented above as being a parametrised statement, *prcI* can also be used to introduce a block that declares a non-parametrised procedure. Similar comments apply to the other laws that we present in this section.

The introduction of a variant block can be achieved with the use of the refinement law below.

Law *vrtI* Variant introduction

$$\begin{aligned} & p_2 \\ = \;& vrtI \\ & |[\,\textbf{proc}\ pn \mathrel{\widehat=} (fpd \bullet \{n = e\}\ p_1)\ \textbf{variant}\ n\ \textbf{is}\ e \bullet p_2\,]| \end{aligned}$$

Syntactic Restrictions

- pn and n are not free in e and p_2;
- $(fpd \bullet p_1)$ and e are well-scoped and well-typed.

Recursion can be introduced by refining $(fpd \bullet \{n = e\}\ p_1)$ and subsequently replacing occurrences of $(fpd \bullet \{0 \leq n < e\}\ p_1)$ in the resulting program by procedure calls (using *pcallI*).

The *pcallI* law has three formulations. The first of them is aimed at the introduction of procedure calls in the main program of procedure blocks that declare non-recursive procedures.

Law *pcallI* Call to a non-recursive procedure introduction

$$\begin{array}{ll} & [\![\, \mathbf{proc}\ pn \mathrel{\widehat=} (fpd \bullet p_1) \bullet p_2[(fpd \bullet p_1)]\,]\!] \\ = & pcallI \\ & [\![\, \mathbf{proc}\ pn \mathrel{\widehat=} (fpd \bullet p_1) \bullet p_2[pn]\,]\!] \end{array}$$

Syntactic Restriction pn is not recursive

We identify an occurrence of a program (parametrised statement) p_1 in a context c by writing $c[p_1]$. Subsequent references to $c[p_2]$ denote the context obtained by substituting p_2 for that particular occurrence of p_1 in c. The program $c[p_2]$ should not be confused with $c(p_2)$. As already explained, the latter, is the result of substituting p_2 for the free occurrences of the procedure name, as opposed to a particular occurrence of p_1, in c.

The introduction of a procedure call in the main program of a variant block can be accomplished by the following formulation of *pcallI*.

Law *pcallI* Procedure call introduction in the main program of a variant block

$$\begin{array}{ll} & [\![\, \mathbf{proc}\ pn \mathrel{\widehat=} (fpd \bullet p_1)\ \mathbf{variant}\ n\ \mathbf{is}\ e \bullet p_2[(fpd \bullet p_3)]\,]\!] \\ \sqsubseteq & pcallI \\ & [\![\, \mathbf{proc}\ pn \mathrel{\widehat=} (fpd \bullet p_1)\ \mathbf{variant}\ n\ \mathbf{is}\ e \bullet p_2[pn]\,]\!] \end{array}$$

provided $\{n = e\}\ p_3 \sqsubseteq p_1$

Syntactic Restrictions

- pn is not free in p_1;
- n is not free in e and p_3.

Recursive calls can be introduced with the use of the last formulation of *pcallI*, which is presented below.

Law *pcallI* Recursive call introduction

$$\begin{array}{ll} & [\![\, \mathbf{proc}\ pn \mathrel{\widehat=} (fpd \bullet p_1[(fpd \bullet \{0 \leq e < n\}\ p_3)])\ \mathbf{variant}\ n\ \mathbf{is}\ e \bullet p_2\,]\!] \\ \sqsubseteq & pcallI \\ & [\![\, \mathbf{proc}\ pn \mathrel{\widehat=} (fpd \bullet p_1[pn]) \bullet p_2\,]\!] \end{array}$$

provided $\{n = e\}\ p_3 \sqsubseteq p_1[(fpd \bullet \{0 \leq e < n\}\ p_3)]$.

Syntactic Restriction n is not free in p_3 and $p_1[pn]$

In the next section, we present an example that illustrates the application of this and several other refinement laws we have presented here.

3.7.2 Parametrised Statements

The *pcallI* law allows the substitution of parametrised procedure names for parametrised statements. We still need. however, laws that introduce parametrised statements. In this section, we present three such laws, which account for value, result, and value-result parametrised statements. They correspond to the laws of [44] that introduce substitutions that apply to specification statements. The laws of [45] combine an application of these simpler laws with an application of a law that introduces procedure calls.

The law that introduces a call-by-value is as follows.

Law vS Value specification

$$\begin{array}{ll} & w : [pre[el/vl], post[el, el'/vl, vl']\,] \\ = & vS \\ & (\mathbf{val}\ dvl \bullet w : [pre, post])(el) \end{array}$$

where dvl declares the variables of vl.

Syntactic Restrictions

- The variables of vl are not in w;
- The variables of w are not free in el.

The introduction of a call-by-result can he achieved with the refinemeut law that is presented below.

Law rS Result specification

$$\begin{array}{ll} & w, vl_2 : [pre, post] \\ = & rS \\ & (\mathbf{res}\ dvl_1 \bullet w, vl_1 : [pre, post[vl_1/vl_2]\,])(vl_2) \end{array}$$

where dvl_1 declares the variables of vl_1.

Syntactic Restrictions

- vl_1 and vl_2 have the same length and contain no duplicated variables:
- The variables of vl_1 are not in w and are not free in pre;
- The variables of vl_1 and vl_1' and are not free in $post$.

Another formulation of rS allows the introduction of a call-by-result whose actual parameter is a function application.

Law rS Result specification (function application as actual parameter)

$$\begin{array}{ll} & w, f : [pre, \{x?\} \lhd f' = \{x?\} \lhd f \land post[f'\ x?/fp']\ ] \\ = & rS \\ & (\textbf{res}\ fp : t \bullet w, fp : [pre, post])(f\ x?) \end{array}$$

where t is the type that contains the range of f.

Syntactic Restrictions

- f is of a function type;
- f and fp are not in w;
- f and f' and are not free in $post$.

The program $w, f : [pre, \{x?\} \lhd f' = \{x?\} \lhd f \land post[f'\ x?/fp']\ ]$ modifies f only by changing $f\ x?$ as specified in $post[f'\ x?/fp']$. The program $w, fp : [pre, post]$ changes fp in the same way. So, by passing the parameter $f\ x?$ to $(\textbf{res}\ fp \bullet w, fp : [pre, post])$, we obtain the desired effect on f.

The law vrS (value-result specification) presented below introduces a call-by-value-result.

Law vrS Value-result specification

$$\begin{array}{ll} & w, vl_2 : [pre[vl_2/vl_1], post[vl_2/vl_1]\ ] \\ = & vrS \\ & (\textbf{val-res}\ dvl_1 \bullet w, vl_1 : [pre, post[vl_1'/vl_2']\ ])(vl_2) \end{array}$$

where dvl_1 declares the variables of vl_1.

Syntactic Restrictions

- The variables of vl_1 are not in w;
- The variables of vl_1' are not free in $post$;
- The variables of vl_2 and vl_2' are not free in $w : [pre, post]$.

As rS, this law also has an extra formulation which contemplates calls that have function applications as actual parameters. This additional formulation of vrS is similar to the second formulation of rS presented above, and can be found in Appendix D. As a matter of fact, yet a third formulation of vrS is presented (and derived) there. It is more general than the above formulation of vrS, as it can be applied to any form of program and not only specification statements.

The mpS (merge parametrised statements) law, which can also be found in Appeudix D, combines parametrised statements. The three laws mentioned above introduce parametrised statements that use a single mechanism of parameter passing and mpS allows us to merge them, if they are compatible (in a sense precisely defined by the formulation of this law). The mpS law corresponds to the law of [44] that merges substitutions.

As an example, we refine the program $x : [\text{true}, x' = y!\ ]$ mentioned in Section 3.1. As suggested there we implement by recursion a procedure *Fact* that assigns to x the factorial of its value

$$
\begin{array}{l}
|[\ \textbf{proc}\ \mathit{Fact} \mathrel{\widehat{=}} (\textbf{val}\ n : \mathbf{N} \bullet \textbf{if}\ n = 0 \rightarrow x := 1 \mathrel{[]} n > 0 \rightarrow \mathit{Fact}(n-1)\ ;\ x := x \times n\ \textbf{fi}) \\
\qquad \mathit{Fact}(y) \\
]|
\end{array}
$$

Figure 3.1: Collected code for the factorial program

parameter. The first step in our development is the introduction of the variant block that declares *Fact*.

$$
\begin{array}{l}
\quad x : [\mathrm{true}, x' = y!\,] \\
= vrtI \\
\quad |[\ \textbf{proc}\ \mathit{Fact} \mathrel{\widehat{=}} (\textbf{val}\ n : \mathbf{N} \bullet \{N = n\}\ x : [\mathrm{true}, x' = n!\,])\ \textbf{variant}\ N\ \textbf{is}\ n \bullet \\
\qquad\quad x : [\mathrm{trne}, x' = y!\,] \hfill \lhd \\
\quad]|
\end{array}
$$

Obviously, we want to implement the main program of this variant block by a call to *Fact*. With this purpose we introduce an appropriate parametrised statement.

$$
\begin{array}{l}
= vS \\
\quad (\textbf{val}\ n : \mathbf{N} \bullet x : [\mathrm{true}, x' = n!\,])(y)
\end{array}
$$

At this point we can introduce the procedure call.

$$
\begin{array}{l}
\quad |[\ \textbf{proc}\ \mathit{Fact} \mathrel{\widehat{=}} (\textbf{val}\ n : \mathbf{N} \bullet \{N = n\}\ x : [\mathrm{true}, x' = n!\,])\ \textbf{variant}\ N\ \textbf{is}\ n \bullet \\
\qquad\quad (\textbf{val}\ n : \mathbf{N} \bullet x : [\mathrm{true}, x' = n!\,])(y) \\
\quad]| \\
\sqsubseteq pcallI \\
\quad |[\ \textbf{proc}\ \mathit{Fact} \mathrel{\widehat{=}} (\textbf{val}\ n : \mathbf{N} \bullet \{N = n\}\ x : [\mathrm{true}, x' = n!\,])\ \textbf{variant}\ N\ \textbf{is}\ n \bullet \mathit{Fact}(y)\ ]|
\end{array}
$$

The proof-obligation that is generated by this application of *pcallI* amounts to showing that $\{N = n\}\ x : [\mathrm{true}, x = n!\,]$, is refined by itself, which is trivial since refinement is reflexive.

It is not difficult to verify that the program in the procednre body can be refined to the following alternation.

$$
\begin{array}{l}
\textbf{if}\ n = 0 \rightarrow x := 1 \\
[]\ n > 0 \rightarrow x : [n > 0 \land N = n, x' = (n-1)!\,]\ ; \hfill \lhd \\
\qquad\qquad x := x \times n \\
\textbf{fi}
\end{array}
$$

The remaining specification statement assigns to x the factorial of $n - 1$. It can he implemented by a recursive call. First we introduce a parametrised statement using the *vS* law again.

$$
\begin{array}{l}
= vS \\
\quad (\textbf{val}\ n : \mathbf{N} \bullet x : [n + 1 > 0 \land N = n + 1, x' = n!\,])(n-1)
\end{array}
$$

The body of this parametrised statement can be refined to $\{0 \leq n < N\}\ x : [\mathrm{true}, x' = n!\,]$. Now we can apply *pcallI* to introduce the recursive call to *Fact*. The resulting procedure block is presented in Figure 3.1. The proof-obligation that is generated is shown below.

$$
\begin{array}{ll}
 & \{N = n\}\ x : [\text{true}, x' = n!] \\
\sqsubseteq & \textbf{if } n = 0 \rightarrow x := 1 \\
 & [\!]\ n > 0 \rightarrow (\textbf{val } n : \mathbb{N} \bullet \{0 \le n < N\}\ x : [\text{true}, x' = n!])(n-1)\ ;\ x := x \times n \\
 & \textbf{fi}
\end{array}
$$

This is exactly the result that we have obtained when developing the body of *Fact*. Therefore, we do not need to provide any additional justification to discharge this proof-obligation.

In general, if we apply *pcallI* in the way illustrated above, with p_3 as the program in the original specification of the procedure, the discharge of the proof-obligations generated is trivial. This strategy of refinement produces developments that follow Morgan's approach to recursion.

3.7.3 Data Refinement

As already mentioned in Section 3.3, a variable block can be refined using data refinement. This can be accomplished by the refinement law *dR* (data refinement) which we present below. The list of abstract variables is *avl*, the concrete variables are those of *cvl*, and *ci* is the coupling invariant. When applied to a variable block that declares the abstract variables, *dR* generates another variable block that declares the concrete variables instead and whose body data-refines the body of the original variable block.

There are three formulations of *dR*. The first one is as follows.

Law *dr* Data refinement (restricted)

$$
\begin{array}{ll}
 & |[\ \textbf{var}\ dvl;\ davl \bullet p_1\ ]| \\
\sqsubseteq & dR \\
 & |[\ \textbf{var}\ dvl;\ dcvl \bullet p_2\ ]|
\end{array}
$$

provided

- $p_1 \preccurlyeq p_2$;
- $\forall\, dcvl \bullet \exists\, davl \bullet ci$.

where *davl* and *dcvl* declare the variables of *avl* (the abstract variables) and *cvl* (the concrete variables); and *ci* is the coupling invariant.

Syntactic Restrictions

- The variables of *cvl* and *cvl'* are not free in p_1, and are not in *avl*;
- The variables of *avl* and *avl'* are not free in p_2;
- *ci* is a well-scoped and well-typed predicate.

The first proviso of this formulation of *dR* obliges p_1 and p_2 to be related by data refinement. The syntactic restrictions enforce the freeness conditions imposed by the definition of data refinement and guarantee that *ci* is well-scoped and well-typed.

The second proviso requires that any combination of values that can be assumed by the concrete variables correspond to some combination of values that can be assumed by the abstract variables. This proviso is very restrictive, but it is necessary since the initial value of a variable after its declaration is arbitrary. By way of illustration, we take the mean calculator that has been presented in [46]. In its abstract specification, a bag b is used to store a collection of numbers with the objective of calculating its mean: $\sum b / \# b$. The operators $\sum$ and $\#$ are not part of the Z (ZRC) notation, but, for simplicity, we adopt the notation of [46] at this point and assume that $\sum b$ is the sum of the elements of b, and $\# b$, its size. In [46] Morgan and Gardiner suggest a data refinement that replaces b with the variables s and n which record, respectively, the sum and the size of the bag. The coupling invariant is $s = \sum b \land n = \# b$.

Although it can be easily proved that $x, y := \sum b, \# b$ is data-refined by $x, y := s, n$, the refinement below does not hold.

$$|[\,\mathbf{var}\ b : \operatorname{bag} \mathbb{N} \bullet x, y := \textstyle\sum b, \# b\,]| \sqsubseteq |[\,\mathbf{var}\ s, n : \mathbb{N} \bullet x, y := s, n\,]|$$

We observe that the weakest precondition of $|[\,\mathbf{var}\ b : \operatorname{bag} \mathbb{N} \bullet x, y := \sum b, \# b\,]|$ with respect to $x > 0 \Rightarrow y > 0$, for instance, is true, since any bag that has a sum greater than 0 has some element. On the other hand, the weakest precondition of $|[\,\mathbf{var}\ s, n : \mathbb{N} \bullet x, y := s, n\,]|$ with respect to the same postcondition is false. Upon declaration s and n may get any value and, in particular, they may get values such as 3 and 0, which do not correspond to the sum and size of any bag.

The second formulation of *dR* considers variable blocks whose bodies start with an initialisation of the abstract variables and so do not depend on their initial arbitrary values. In this case, the restrictive proviso of the first formulation may be dropped.

Law *dR* Data refinement (variable blocks with initialisation)

$$|[\,\mathbf{var}\ dvl;\ davl \bullet avl : [true, init'] \ ;\ p_1\,]|$$

$$\sqsubseteq \quad dR$$

$$|[\,\mathbf{var}\ dvl;\ dcvl \bullet cvl : [\text{true}, (\exists\, davl \bullet ci \land init)'] \ ;\ p_2\,]|$$

provided $p_1 \preccurlyeq p_2$

where *davl* and *dcvl* declare the variables of *avl* (the abstract variables) and *cvl* (the concrete variables); and *ci* is the coupling invariant.

Syntactic Restrictions

- The variables of *cvl* and *cvl'* are not free in *init* and p_1, and are not in *avl*;
- The variables of *avl* and *avl'* are not free in p_2;
- *ci* is a well-scoped and well-typed predicate.

In [46] just variable blocks with invariants are considered. In these blocks the initialisation is implicit.

The third and last formulation of *dR* applies to every variable block that declares the abstract variables, irrespective of any particular property of the abstract or concrete variables, or of the coupling invariant.

Law *dR* Data refinement

$$\begin{array}{ll} & |[\,\mathbf{var}\ dvl;\ davl \bullet p_1\,]| \\ \sqsubseteq & dR \\ & |[\,\mathbf{var}\ dvl;\ dcvl \bullet cvl : [\mathrm{true}, (\exists\, davl \bullet ci)'] \ ;\ p_2\,]| \end{array}$$

provided $p_1 \preccurlyeq p_2$

where *davl* and *dcvl* declare the variables of *avl* (the abstract variables) and *cvl* (the concrete variables); and *ci* is the coupling invariant.

Syntactic Restrictions

- The variables of cvl and cvl' are not free in p_1, and are not in avl;
- The variables of avl and avl' are not free in p_2;
- ci is a well-scoped and well-typed predicate.

The variable block that is generated by this formulation of *dR* contains an initialisation of the concrete variables, and so, before they are used, these variables are assigned values which correspond to some combination of values of *avl*.

The program p_2 mentioned in all formulations of *dR* can most of the times be calculated from p_1, *avl*, *cvl*, and *ci* using data refinement laws. These are enumerated and derived in Appendix D. They are based on those of [46], but as those of [45], which support the auxiliary variable technique, they can be applied to programs with free initial (or in the terminology adopted here, program) variables. More specifically, the data refinement law that deals with specification statements contemplates the possibility of program variables occurring free in them. This law is shown below.

Data Refinement Law Specification statement

$$\begin{array}{ll} & vl, w : [pre, post] \\ \preccurlyeq & \\ & |[\,\mathbf{con}\ davl \bullet cvl, w : [ci \land pre, \exists\, davl' \bullet ci' \land ul' = ul \land post]\ ]| \end{array}$$

where

- *davl* declares the variables of *avl*;
- $avl = vl, ul$, and vl and ul are disjoint.

Syntactic Restriction The variables of *avl* are not in *w*.

The list of variables *ul* contains the abstract variables that are not in the frame of the specification statement.

The program *p* obtained by data-refining a specification statement *s* with the use of this law is always the most general data refinement of *s*. This result is established by Theorem 3.10, which shows that there is no program that data-refines *s* that cannot be obtained by refining *p*: nothing is lost by taking *p* as the data refinement of *s*.

Theorem 3.10 *For all programs $vl, w : [pre, post]$ and p, lists of abstract and concrete variables avl and cvl, and coupling invariant ci, if $vl, w : [pre, post] \preccurlyeq p$ then*

$$|[\ \mathbf{con}\ davl \bullet cvl, w : [ci \land pre, \exists\, davl' \bullet ci' \land ul' = ul \land post]\ ]| \sqsubseteq p$$

The lists of variables vl and ul partition avl. The declaration $davl$ introduces the variables of avl. The variables of cvl and cvl' must not be free in $vl, w : [pre, post]$, and avl and cvl must be disjoint.

Proof

$$
\begin{array}{l}
wp.|[\ \mathbf{con}\ davl \bullet cvl, w : [ci \land pre, \exists\, davl' \bullet ci' \land ul' = ul \land post]\ ]|.\psi \\
\equiv\ \exists\, davl \bullet ci \land pre \land (\forall\, dcvl';\ dw' \bullet (\exists\, davl' \bullet ci' \land ul' = ul \land post) \Rightarrow \psi)[\_/']\\
\hfill [\text{by definition of } wp]\\
\equiv\ \exists\, davl \bullet ci \land pre \land (\forall\, dcvl';\ dw';\ davl' \bullet ci' \land ul' = ul \land post \Rightarrow \psi)[\_/']\\
\hfill [\text{by } avl' \text{ are not free in } \psi]\\
\equiv\ \exists\, davl \bullet ci \land pre \land (\forall\, davl';\ dw' \bullet ul' = ul \land post \Rightarrow \forall\, dcvl' \bullet ci' \Rightarrow \psi)[\_/']\\
\hfill [\text{by } cvl' \text{ are not in } ul' \text{ and are not free in } davl',\ dw', \text{ and } post]\\
\equiv\ \exists\, davl \bullet ci \land pre \land (\forall\, dvl';\ dw' \bullet post \Rightarrow \forall\, dcvl' \bullet ci' \Rightarrow \psi)[ul/ul'][\_/']\\
\hfill [\text{by predicate calculus}]\\
\equiv\ \exists\, davl \bullet ci \land pre \land (\forall\, dvl';\ dw' \bullet post \Rightarrow \forall\, dcvl' \bullet ci' \Rightarrow \psi)[\_/']\\
\hfill [\text{by a property of substitution}]\\
\equiv\ \exists\, davl \bullet ci \land wp.vl, w : [pre, post].\forall\, dcvl' \bullet ci' \Rightarrow \psi \hfill [\text{by definition of } wp]\\
\Rightarrow wp.p.\exists\, davl' \bullet ci' \land \forall\, dcvl' \bullet ci' \Rightarrow \psi \hfill [\text{by assumption}]\\
\Rightarrow wp.p.\exists\, davl' \bullet \psi \hfill [\text{by monotonicity of } wp]\\
\equiv\ wp.p.\psi \hfill [\text{by } avl' \text{ are not free in } \psi]
\end{array}
$$

□

The data refinement laws that apply to annotations are special cases of the law that considers specification statements. Therefore, it is a direct consequence of the theorem above that, when applied to a program p, all these laws produce the most general program that data-refines p.

Schemas and assignments can be data-refined by first transforming them into equivalent specification statements and then using the appropriate data refinement law. In the case of schemas, the conversion law bC (basic conversion) can be used to perform the transformation; in the case of assignments the refinement law sS (simple specification) is suitable.

In general, data refinement distributes through the program structure. This means that the structure of the program is preserved when it is data-refined. Procedure and variant blocks, however, may be lost. Procedure blocks that declare non-parametrised procedures, whether recursive or not, can be data-refined by merely data-refining their bodies and main programs. The same cannot be said about procedure blocks that introduce parametrised procedures and about variant blocks.

Parametrised procedures have to be removed before the application of the data refinement laws of ZRC can proceed. In the case where the procedure is not recursive, the first formulation of *pcallI* (procedure call introduction) can be used to remove the procedure calls and then *prcI* (procedure introduction) can be used to remove the procedure block. Unfortunately, if the procedure is recursive, it does not seem to exist a simple way to do that. Applications of parametrised statements to actual parameters can be data-refined by transforming them into equivalent variable blocks. These transformations can be guided by the definition of the parametrised statements semantics itself.

As far as variant blocks are concerned, even though in theory we can data-refine those that do not declare parametrised procedures by data-refining their procedure bodies and main programs, this is not really worthwhile. If we are not able to redefine the variants in a way that guarantees that their relationship to the procedure bodies is maintained, then they will be of no use to the process of refining the procedure.

In face of these restrictions, and also observing that assignments are transformed into specification statements during data refinement, we conclude that any necessary data refinement should be carried out as early in the refinement process as possible. The later the data refinement occurs, the greater is the chance that part of the effort to refine the program is wasted.

The data refinement relation is not reflexive in general. However, if the abstract variables and the global variables that are free in the coupling invariant, and their dashed counterparts are not free in a program, then it is data-refined by itself. This is established by the theorem we present in the sequel.

Theorem 3.11 *For every program p, all lists of abstract and concrete variables avl and cvl, and every coupling invariant ci, if the variables of avl, the free variables of ci, and their dashed counterparts are not free in p, then $p \preccurlyeq p$. The variables of cvl and cvl' must not be free in p either, and avl and cvl must be disjoint.*

Proof

$$
\begin{array}{ll}
\exists\, davl \bullet ci \wedge wp.p.\psi & \\
\Rightarrow \exists\, davl \bullet wp.p.(\psi \wedge ci') & \text{[by Lemma 3.3]} \\
\Rightarrow \exists\, davl \bullet wp.p.\exists\, davl' \bullet ci' \wedge \psi & \text{[by monotonicity of } wp\text{]} \\
\equiv wp.p.\exists\, davl' \bullet ci' \wedge \psi & \text{[by } avl \text{ are not free in } wp.p.\exists\, davl' \bullet ci' \wedge \psi \text{ (by Theorem 3.9)]}
\end{array}
$$

□

Applications of this theorem may save effort during the data refinement of a variable block.

3.8 Conclusions

Our main objective has been the proposal and formalisation of a refinement calculus for Z whose design builds upon results already available in this area and which employs a notation that is compatible with the Z style. Indeed ZRC includes conversion laws that correspond to those that have been initially presented or suggested in [34, 64, 58], but uses the decoration conventions of the Oxford style of writing Z specifications. Furthermore, its refinement laws are based mainly on those of [45] with adaptations and extensions that contemplate the Z style as in [65].

In Appendix D we prove the validity of all conversion and refinement laws of ZRC. To make this possible, we have provided a weakest precondition semantics for ZRC-L and defined a refinement relation. The scope rules of ZRC-L have also been specified and the syntactic restrictions of the laws take these rules into account. As a consequence of this effort of formalisation, we have clarified many details of the original presentation and formalisation of the conversion and refinement laws.

Another method for refining Z specifications is presented in [32]. This work, which is discussed in more detail in Chapter 5, defines *wp* as a schema operator and proposes a refinement-*wp* calculus for Z. Both in [32] and here, the postconditions of *wp* define state transitions instead of states as in [47, 45]. We have chosen this alternative approach because it seems to be difficult to justify differences in the treatment of dashed and undashed variables in the context of Z. We have managed, however, to deal separately with the additional complication that has been introduced due to the treatment of these more complex postconditions. As a consequence, the ZRC laws are derived in Appendix D in much the same way as the corresponding laws of Morgan's calculus can be derived in the framework of [47, 45].

In [1, 2, 4], where another formalisation of the stepwise refinement technique is proposed, specifications employ the dashing convention of Z (and ZRC). Nevertheless, dashed variables are regarded as local to specifications and cannot occur free in the postconditions of *wp*. In other words, as in [47, 45], these predicates are assumed to specify states.

Our definition for the data refinement relation also considers postconditions that define state transitions. This definition is equivalent to the Z characterisation of data refinement and to the definition presented in [46], which takes into account only postconditions that define states. Therefore, once again no extra complexity has been introduced in the derivation of the ZRC laws.

The data refinement laws of ZRC are based on those of [46], but apply directly to programs with free occurrences of program variables. In particular, the data refinement law of ZRC that applies to specification statements does not seem to have been proposed before.

The technique of data refinement that is presented in [45] makes use of auxiliary variables. As shown in [39, 46], this technique is equivalent to the application of two data refinements: a first data refinement introduces the concrete variables, the abstract variables are then made auxiliary (in a sense precisely defined in [39]), and finally they are removed by a second data refinement. These data refinements are special in that they take an empty list of variables as argument. There are no abstract variables in the case of the first data refinement, and no concrete variables in the case of the second data refinement.

Functional data refinement [46, 45] is yet another specialised technique also described in [58, 65]. In this case, the coupling invariant is a conjunction between a concrete invariant (a predicate over the concrete and global variables) and a number of equalities. For each abstract variable there is an equality that defines its value as a function of the values of the concrete variables (and, possibly, of the global variables) when these satisfy the concrete invariant.

Our treatment of procedures follows the approach of [3], because we have found the formalism in [41] to be inconsistent. As we have explained, there is a subtle interaction between substitution, procedures and parameters. Of particular importance is the definition of the substitution operator when applied to a procedure name. Two alternatives have been analysed: one of them establishes that the substitution operates on the procedure body (context dependent substitution); the other one specifies that the procedure name itself is taken into account (syntactic substitution). Unfortunately, whichever option is chosen, Morgan's approach to procedures and parameters runs into difficulties.

To our knowledge, the interaction between procedures, parameters, and substitution that we have discussed was originally pointed out in [55]. Sampaio's idea of restricting the application of the renaming law can be considered as a solution to the problems found in Morgan's approach, but this restriction turns out to be too severe in practice. Also, Sampaio has presented no mathematical model to justify the restricted version of the renaming law.

The problem with Morgan's work seems to be a consequence of an unfortunate design decision: formal parameters are not regarded as local variables in the procedure body. This decision was perhaps an attempt to avoid parametrised statements, as suggested by Back. They do indeed increase the complexity of Back's formalism, which involves a greater number of definitions and theorems that Morgan's. However Back's approach does not present any of the complications we have uncovered in Morgan's work and does not impose restrictions as the solution proposed by Sampaio. Therefore, it seems to be the right direction to follow. Regarding formal parameters as ordinary (global) variables causes problems, as revealed by our study.

As far as the development of procedures is concerned, however, we follow Morgan's style. In [3] Back presents rules to prove the correctness of (recursive) procedures. In contrast, the development of procedures in ZRC is supported by refinement laws. Some of them have a counterpart in Morgan's calculus and others are additional laws that support the use of variants he proposes in [45].

Another analysis of the usage of procedures in the refinement calculus is presented in [20]. This study, however, concentrates on the methodological aspects of the development of procedures. In [20], the suitability of the refinement laws presented in [45] is discussed and an alternative strategy of program refinement, where (non-recursive) procedures are introduced in the final phase of development, is suggested. In [49], Morris presents another formalisation of procedures and parameters. We have not considered this work in our study because it is similar to Back's.

Completeness has not been tackled here. We have proved that the ZRC laws are sound, but have not considered whether or not they are enough to derive any possible program. Actually, since we cover only downward simulation [5], our set of refinement laws is not going to be complete. Apparently, a complete data refinement method for a language with recursion and unbounded nondeterminism is yet to be found [19].

In the next chapter we apply ZRC to refine (some of) the operations of three different system specifications. There we discuss a few issues concerning the use of ZRC.

Chapter 4

Case Studies

A few short program developments that illustrate the application of ZRC have already been presented in Chapters 1 and 3. In this chapter we present three more substantial case studies. In the next section we develop an implementation for a class manager; in Section 4.2 we present a development of (part of) a text editor; finally, in Section 4.3 we refine some operations of an Airbus cabin-illumination system. In Section 4.4 we finish this chapter by presenting a few conclusions that we have drawn from these case studies.

The examples considered have not been tailored to ZRC. The class manager specification has been originally written by Jones [30] using VDM and in [34] King uses this example as a case study for his technique of refining Z specifications. The text editor has been specified by Neilson in [51], where a technique based mainly on rules of verification is used to develop a C implementation for this system. The illumination system has been specified in [23].

4.1 The Class Manager

In [34] King presents initially a concise Z specification of the class manager, which is then data-refined with the use of the Z rules. In what follows, we reproduce the more concrete specification.

The class manager records the students that are enrolled on a class, distinguishing those that have done the midweek exercises. The set of student identifications is called *Student* and is introduced as a given set.

$[Student]$

A global constant max establishes the maximum size of a class: a positive natural number.

$$\begin{array}{|l} max : \mathbb{N} \\ \hline max > 0 \end{array}$$

The state components are cl, which records the identification of the students that are registered in the class; ex, which singles out the students that have done the exercises; and num, the number of students which are enrolled.

$$\begin{array}{|l}
\textit{Class\_1} \\
\hline
cl : 1 \mathbin{..} max \rightarrow Student \\
ex : 1 \mathbin{..} max \rightarrow Boolean \\
num : 0 \mathbin{..} max \\
\hline
((1 \mathbin{..} num) \lhd cl) \in (\mathbb{N} \rightarrowtail Student) \\
\hline
\end{array}$$

The components *cl* and *ex* are both arrays (total functious), with index set $1 \mathbin{..} max$, and *num* is a natural number in the interval from 0 to *max*. The information about the *num* students that are enrolled in the class is held in the positions from 1 to *num* of *cl* and *ex*. If *i* is an index that identifies one of these positions, then the student $cl\ i$ has done the midweek exercises exactly when $ex\ i$. The state invariant establishes that, when restricted to $1 \mathbin{..} num$, *cl* is injective, so that the record of enrolled students does not contain repetitions.

The class manager operations are specified by *Enrol_ok_1*, *Compl_ok_1*, and *Leave_ok_1*. The first of these schemas defines the operation that enrolls a student on the class. Its input, $s?$, is the student identification.

$$\begin{array}{|l}
\textit{Enrol\_ok\_1} \\
\hline
\Delta Class\_1 \\
s? : Student \\
\hline
s? \notin \{\, i : 1 \mathbin{..} num \bullet cl\ i \,\} \\
num < max \\
cl' = cl \oplus \{num' \mapsto s?\} \\
ex' = ex \oplus \{num' \mapsto \mathrm{false}\} \\
num' = num + 1 \\
\hline
\end{array}$$

The success of this operation depends on $s?$ not being already registered and the class not being full. If registered, $s?$ is supposed not to have done the exercises. The schema *Compl_ok_1* defines the operation that records that a student has completed the exercises. It also takes as input an identification represented by $s?$, which must be that of a registered student who has not yet completed the exercises.

$$\begin{array}{|l}
\textit{Compl\_ok\_1} \\
\hline
\Delta Class\_1 \\
s? : Student \\
\hline
\exists i : 1 \mathbin{..} num \bullet cl\ i = s? \land ex\ i = \mathrm{false} \land \\
\qquad cl' = cl \land ex' = ex \oplus \{i \mapsto \mathrm{true}\} \land num' = num \\
\hline
\end{array}$$

The operation *Leave_ok_1*, which records that a student has left the class, is not actually considered in [34], but we propose a definition for this operation and refine it below. As *Enrol_ok_1* and *Compl_ok_1*, it takes as input an identification $s?$, which in this case must be among those recorded

Operation	Precondition
Enrol_ok_1	$s? \notin \{\, i : 1 \mathbin{..} num \bullet cl\ i \,\} \land num < max$
Compl_ok_1	$\exists\, i : 1 \mathbin{..} num \bullet cl\ i = s? \land ex\ i = \text{false}$
Leave_ok_1	$\exists\, i : 1 \mathbin{..} num \bullet cl\ i = s?$

Table 4.1: Precondition of the Operations

in the first *num* positions of *cl*.

Leave_ok_1

$$\begin{array}{|l}
\Delta Class\_1 \\
s? : Student \\
\hline
\exists\, i : 1 \mathbin{..} num \bullet cl\ i = s? \land \\
\qquad (1 \mathbin{..} i-1) \lhd cl' = (1 \mathbin{..} i-1) \lhd cl \land \\
\qquad (1 \mathbin{..} i-1) \lhd ex' = (1 \mathbin{..} i-1) \lhd ex \land \\
\qquad (i \mathbin{..} num-1) \lhd cl' = (i+1 \mathbin{..} num) \lhd cl \land \\
\qquad (i \mathbin{..} num-1) \lhd ex' = (i+1 \mathbin{..} num) \lhd ex \land \\
\qquad num' = num - 1
\end{array}$$

Table 4.1 shows the preconditions of *Enrol_ok_1*, *Compl_ok_1*, and *Leave_ok_1*.

As part of the error treatment, we extend the set of messages *Response* that is defined in [34].

$Response ::= ok \mid full \mid found \mid missing \mid not\_found$

The specification of the case of success is as indicated in the Oxford style of writing Z specifications.

Success

$$\begin{array}{|l}
resp! : Response \\
\hline
resp! = ok
\end{array}$$

The error cases of the operations are defined by the schemas that follow. The first schema specifies the error case of *Enrol_ok_1* in which the class is full.

Full_1

$$\begin{array}{|l}
\Xi Class\_1 \\
resp! : Response \\
\hline
num = max \\
resp! = full
\end{array}$$

The error case in which the student is already recorded is contemplated by *Found_1*.

Found_1

$$\begin{array}{|l}
\Xi Class\_1 \\
s? : Student \\
resp! : Response \\
\hline
\exists\, i : 1 \mathbin{..} num \bullet cl\ i = s? \\
resp! = found
\end{array}$$

The schema *Missing_1* defines the error case of *Compl_ok_1*: the student is not enrolled or has

Operation	Precondition
$Full\_1$	$num = max$
$Found\_1$	$\exists i : 1 \,..\, num \bullet cl\ i = s?$
$Missing\_1$	$\forall i : 1 \,..\, num \bullet cl\ i \neq s? \lor ex\ i = \text{true}$
$NotFound\_1$	$\forall i : 1 \,..\, num \bullet cl\ i \neq s?$

Table 4.2: Precondition of the Error Conditions

already completed the exercises.

$$\begin{array}{|l}
\multicolumn{1}{l}{Missing\_1} \\
\Xi Class\_1 \\
s? : Student \\
resp! : Response \\
\hline
\forall i : 1 \,..\, num \bullet cl\ i \neq s? \lor ex\ i = \text{true} \\
resp! = missing \\
\hline
\end{array}$$

The final error case is that of *Leave_ok_1*: the student is not registered iu the class.

$$\begin{array}{|l}
\multicolumn{1}{l}{NotFound\_1} \\
\Xi Class\_1 \\
s? : Student \\
resp! : Response \\
\hline
\forall i : 1 \,..\, num \bullet cl\ i \neq s? \\
resp! = not\_found \\
\hline
\end{array}$$

The preconditions of *Full_1*, *Found_1*, *Missing_1*, and *NotFound_1* are presented in Table 4.2.

The schemas *Enrol_1*, *Complete_1*, and *Leave_1* give a robust definition for the operations of the class manager.

$$\begin{array}{l}
Enrol\_1 \mathrel{\widehat=} (Enrol\_ok\_1 \land Success) \lor Full\_1 \lor Found\_1 \\
Complete\_1 \mathrel{\widehat=} (Compl\_ok\_1 \land Success) \lor Missing\_1 \\
Leave\_1 \mathrel{\widehat=} (Leave\_ok\_1 \land Success) \lor NotFound\_1
\end{array}$$

These definitions complete the specification.

In order to develop an implementation for the class manager, we consider each of the schemas *Enrol_1*, *Complete_1*, and *Leave_1* in succession. We convert all of them to alternations that implement the successful and error cases separately. In the case of *Complete_1* and *Leave_1* we apply the third formulation of *sdisjC* (schema disjunction conversion). In the case of *Enrol_1*, since it is specified as a disjunction of three schemas, we use a more general formulation of this conversion law. The actual formulation of *sdisjC* that we apply is a straightforward extension of its third formulation and as such is not presented here. In all cases, the variable that is introduced by *sdisjC* is named w. If $s?$ is already in cl (the student is already enrolled on the class), then w is initialised with its position. Otherwise, w takes the value $num + 1$. Its type is $1..max + 1$.

By applying *sdisjC* to *Enrol_1*, we can obtain the variable block shown below. The guard $w = num + 1 \land num < max$ identifies the successful case of *Enrol_1*, $num = max$ identifies the case in which the class is full, and finally $w \in 1\,..\,num$ identifies the case in which $s?$ is already enrolled on the class.

```
$Enrol\_1$
$\sqsubseteq sdisjC$
  $|[\,\mathbf{var}\ w : 1\,..\,max + 1 \bullet$
    $w : [\mathrm{true}, (w' \in 1..num \land cl\ w' = s?) \lor (w' = num + 1 \land s? \notin \{\, i : 1..num \bullet cl\ i \,\})]$ ;
    $\mathbf{if}\ w = num + 1 \land num < max \rightarrow$
        $\{w = num + 1 \land s? \notin \{\, i : 1\,..\,num \bullet cl\ i \,\} \land num < max\}\,(Enrol\_ok\_1 \land Success)$
    $[]\ num = max \rightarrow$
        $\left\{\begin{pmatrix} (w \in 1\,..\,num \land cl\ w = s?) \lor (w = num + 1 \land s? \notin \{\, i : 1\,..\,num \bullet cl\ i \,\}) \\ num = max \end{pmatrix}\right\}$
        $Full\_1$
    $[]\ w \in 1\,..\,num \rightarrow \{w \in 1\,..\,num \land cl\ w = s?\}\ Found\_1$
    $\mathbf{fi}$
  $]|$
```

The four proof-obligations generated by this application of $sdisjC$ are implications whose antecedent we simplify below.

$$\begin{aligned}
&((w \in 1\,..\,num \land cl\ w = s?) \lor (w = num + 1 \land s? \notin \{\, i : 1\,..\,num \bullet cl\ i \,\})) \land \\
&(\mathrm{pre}\ (Enrol\_ok\_1 \land Success) \lor \mathrm{pre}\, Full\_1 \lor \mathrm{pre}\, Found\_1) \\
&\equiv ((w \in 1\,..\,num \land cl\ w = s?) \lor (w = num + 1 \land s? \notin \{\, i : 1\,..\,num \bullet cl\ i \,\})) \land \\
&\quad (\mathrm{pre}\, Enrol\_ok\_1 \lor \mathrm{pre}\, Full\_1 \lor \mathrm{pre}\, Found\_1) \qquad \text{[by a property pointed out in [65, p.211]]} \\
&\equiv ((w \in 1\,..\,num \land cl\ w = s?) \lor (w = num + 1 \land s? \notin \{\, i : 1\,..\,num \bullet cl\ i \,\})) \land \\
&\quad ((s? \notin \{\, i : 1\,..\,num \bullet cl\ i \,\} \land num < max) \lor num = max \lor \exists\, i : 1\,..\,num \bullet cl\ i = s?) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{[see Tables 4.1 and 4.2]} \\
&\equiv (w \in 1\,..\,num \land cl\ w = s?) \lor (w = num + 1 \land s? \notin \{\, i : 1\,..\,num \bullet cl\ i \,\}) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{[by } num \le max \text{ since } num : 0\,..\,max\text{]}
\end{aligned}$$

Since pre distributes over disjunctions [65], pre $(Enrol\_ok\_1 \land Success) \lor \mathrm{pre}\, Full\_1 \lor \mathrm{pre}\, Found\_1$ is the precondition of $Enrol\_1$. As this is supposed to be a robust operation, it should not come as a surprise the fact that, as shown above, this predicate can be reduced to true. The consequents of the proof-obligations are enumerated below.

(a) $(w = num + 1 \land num < max) \lor num = max \lor w \in 1\,..\,num$;

(b) $w = num + 1 \land num < max \Rightarrow s? \notin \{\, i : 1\,..\,num \bullet cl\ i \,\} \land num < max$;

(c) $num = max \Rightarrow num = max$; and

(d) $w \in 1\,..\,num \Rightarrow \exists\, i : 1\,..\,num \bullet cl\ i = s?$.

We consider the cases $w \in 1\,..\,num \land cl\ w = s?$ and $w = num + 1 \land s? \notin \{\, i : 1\,..\,num \bullet cl\ i \,\}$

separately. If $w \in 1 \mathbin{..} num$ and $cl\ w = s?$, then, obviously, $w \in 1 \mathbin{..} num$, so (a) holds; if we suppose additionally that $w = num + 1$, we have a contradiction, so (b) holds; and finally, w is a witness for $\exists i : 1 \mathbin{..} num \bullet cl\ i = s?$, and so (d) holds. Since (c) is trivial, it does not have to be considered. If $w = num + 1$ and $s? \notin \{\ i : 1 \mathbin{..} num \bullet cl\ i\ \}$, then, since (a) can be written more simply as $w = num + 1 \lor num = max \lor w \in 1 \mathbin{..} num$ (by $num \leq max$), it follows from $w = num + 1$; (b) holds trivially; and if we assume that $w \in 1 \mathbin{..} num$, then there is a contradiction, and conseqnently (d) holds.

The operation $Enrol\_ok\_1 \land Success$ can be converted to a sequential composition by an application of *sconjC* (schema conjunction conversion).

$$
\begin{array}{l}
\quad Enrol\_ok\_1 \land Success \\
\sqsubseteq sconjC \\
\quad Enrol\_ok\_1 \ ; \ Success
\end{array}
$$

An application of the *assC* (assignment conversion) law justifies the conversion of *Success* to an assignment.

$$
\begin{array}{l}
\quad Success \\
\sqsubseteq assC \\
\quad resp! := ok
\end{array}
$$

In [34], the fact that *Enrol_ok_1* and *Success* act on different states is not exploited and this conjunction is expanded before being translated. This is perhaps because the translation rule that applies to schema conjunctions presented in [34] is not properly formulated and has proved to be misleading. Conjunctions like $Enrol\_ok\_1 \land Success$ are used in the definition of most robust operations specified in accordance with the Oxford style.

By applying the *bC* (basic conversion) law to *Enrol_ok_1*, we obtain the following specification statement.

$$
\begin{array}{l} cl, \\ ex, \\ num \end{array} : \left[\begin{pmatrix} ((1 \mathbin{..} num) \lhd cl) \in (\mathbb{N} \pinj Student) \\ s? \notin \{\ i : 1 \mathbin{..} num \bullet cl\ i\ \} \\ num < max \end{pmatrix}, \begin{pmatrix} ((1 \mathbin{..} num') \lhd cl') \in (\mathbb{N} \pinj Student) \\ s? \notin \{\ i : 1 \mathbin{..} num \bullet cl\ i\ \} \\ num < max \\ cl' = cl \oplus \{num' \mapsto s?\} \\ ex' = ex \oplus \{num' \mapsto \text{false}\} \\ num' = num + 1 \end{pmatrix} \right]
$$

This program can be implemented by a multiple assignment that inserts $s?$ in cl and adjusts ex and num.

$$
\begin{array}{l}
\sqsubseteq assigI \\
\quad cl, ex, num := cl \oplus \{num + 1 \mapsto s?\}, ex \oplus \{num + 1 \mapsto \text{false}\}, num + 1
\end{array}
$$

The proof-obligation associated with this application of *assigI* (assignment introduction) is an implication. Its antecedent is the conjunction of the predicates (a_1) to (a_3) listed below.

(a_1) $((1 \mathrel{..} num) \lhd cl) \in (\mathbb{N} \rightarrowtail\!\!\!\!\!\!\!\;\mapsto Student)$;

(a_2) $s? \notin \{\, i : 1 \mathrel{..} num \bullet cl\ i \,\}$; and

(a_3) $num < max$.

The consequent of this implication is the conjunction of the following predicates.

(c_1) $((1 \mathrel{..} num + 1) \lhd (cl \oplus \{num + 1 \mapsto s?\})) \in (\mathbb{N} \rightarrowtail\!\!\!\!\!\!\!\;\mapsto Student)$;

(c_2) $s? \notin \{\, i : 1 \mathrel{..} num \bullet cl\ i \,\}$;

(c_3) $num < max$;

(c_4) $cl \oplus \{num + 1 \mapsto s?\} = cl \oplus \{num + 1 \mapsto s?\}$;

(c_5) $ex \oplus \{num + 1 \mapsto \mathrm{false}\} = ex \oplus \{num + 1 \mapsto \mathrm{false}\}$; and

(c_6) $num + 1 = num + 1$.

The proof of (c_1) amounts to establishing that the invariant is maintained by the assignment. This follows from (a_1) and (a_2).

Since the specification of *Enrol_ok_1* explicitly states its precondition, it appears in both the pre and the postcondition of the specification statement that is generated by applying bC to *Enrol_ok_1*. As a consequence, the precondition of *Enrol_ok_1* appears above as (c_2) and (c_3), and we have to prove that it holds. Fortunately, this kind of proof does not really add up to the complexity of the proof-obligation: in this case, (c_2) and (c_3) also appear as (a_2) and (a_3). The conjunctions (c_4) to (c_6) hold trivially.

The application of bC to *Full_1* generates the following specification statement.

$$resp! : [((1 \mathrel{..} num) \lhd cl) \in (\mathbb{N} \rightarrowtail\!\!\!\!\!\!\!\;\mapsto Student) \land num = max, num = max \land resp! = full]$$

It can also be implemented by an assignment.

$$\begin{array}{l} \sqsubseteq assigI \\ \quad resp! := full \end{array}$$

In this case the proof-obligation generated is the much simpler implication below.

$$((1 \mathrel{..} num) \lhd cl) \in (\mathbb{N} \rightarrowtail\!\!\!\!\!\!\!\;\mapsto Student) \land num = max \Rightarrow num = max \land full = full$$

It is a trivial task to discharge this proof-obligation.

In much the same way, *Found_1* can be refined to the assignment $resp! := found$. The assumptions that remain in the branches of the alternation can be refined to skip by applying the *assumpR* (assumption removal) law, and then eliminated by the *slC* (skip left composition) law.

The only program that still needs to be refined is the specification statement that initialises w. We implement it with an iteration whose development can be carried out in a standard way and, for the sake of conciseness, is not presented here. This iteration can be found in Figure 4.1, where we

```
|[ var $w : 1 \mathinner{\ldotp\ldotp} max + 1$ •
    $w := 1$ ;
    do $w \neq num + 1 \wedge cl\ w \neq s? \rightarrow w := w + 1$ od ;
    if $w = num + 1 \wedge num < max \rightarrow$
        $cl, ex, num := cl \oplus \{num + 1 \mapsto s?\}, ex \oplus \{num + 1 \mapsto \text{false}\}, num + 1$
    [] $num = max \rightarrow resp! := full$
    [] $w \in 1 \mathinner{\ldotp\ldotp} num \rightarrow resp! := found$
    fi
]|
```

Figure 4.1: Implementation of *Enrol_1*

presented the collected code of *Enrol_1*. Its invariant is $w \leq num + 1 \wedge s? \notin \operatorname{ran}((1 \mathinner{\ldotp\ldotp} w - 1) \lhd cl)$ and its variant is $num + 1 - w$.

The development of *Complete_1* is similar to that of *Enrol_1* and is not presented in as many details. The application of *sdisjC* to *Complete_1* can introduce the variable block below. The guards of the alternation are $w \in 1 \mathinner{\ldotp\ldotp} num \wedge ex\ w = \text{false}$ and $w = num + 1 \vee ex\ w = \text{true}$. They identify whether or not the student is enrolled and has not yet completed the exercises.

```
    $Complete\_1$
  $\sqsubseteq$ $sdisjC$
    |[ var $w : 1 \mathinner{\ldotp\ldotp} max + 1$ •
      $w : [\text{true}, (w' \in 1 \mathinner{\ldotp\ldotp} num \wedge cl\ w' = s?) \vee (w' = num + 1 \wedge s? \notin \{ i : 1 \mathinner{\ldotp\ldotp} num \bullet cl\ i \})]$ ;
      if $w \in 1 \mathinner{\ldotp\ldotp} num \wedge ex\ w = \text{false} \rightarrow$
          $\{w \in 1 \mathinner{\ldotp\ldotp} num \wedge cl\ w = s? \wedge ex\ w = \text{false}\}\ (Compl\_ok\_1 \wedge Success)$
      [] $w = num + 1 \vee ex\ w = \text{true} \rightarrow$
          $\left\{\begin{pmatrix} (w \in 1 \mathinner{\ldotp\ldotp} num \wedge cl\ w = s?) \vee (w = num + 1 \wedge s? \notin \{ i : 1 \mathinner{\ldotp\ldotp} num \bullet cl\ i \}) \\ w = num + 1 \vee ex\ w = \text{true} \end{pmatrix}\right\}$
          $Missing\_1$
      fi
    ]|
```

Three proof-obligations are generated by this application of *sdisjC*. They are all implications with antecedent $(w \in 1 \mathinner{\ldotp\ldotp} num \wedge cl\ w = s?) \vee (w = num + 1 \wedge s? \notin \{ i : 1 \mathinner{\ldotp\ldotp} num \bullet cl\ i \})$. The consequents are the predicates we show below.

(c_1) $(w \in 1 \mathinner{\ldotp\ldotp} num \wedge ex\ w = \text{false}) \vee w = num + 1 \vee ex\ w = \text{true}$;

(c_2) $w \in 1 \mathinner{\ldotp\ldotp} num \wedge ex\ w = \text{false} \Rightarrow \exists i : 1 \mathinner{\ldotp\ldotp} num \bullet cl\ i = s? \wedge ex\ i = \text{false}$; and

(c_3) $w = num + 1 \vee ex\ w = \text{true} \Rightarrow \forall i : 1 \mathinner{\ldotp\ldotp} num \bullet cl\ i \neq s? \vee ex\ i = \text{true}$.

It is not difficult to discharge these proof-obligations if the cases $w \in 1 \mathinner{\ldotp\ldotp} num \wedge cl\ w = s?$ and

```
|[ var $w : 1 .. max + 1 \bullet$
     $w := 1$ ;
     do  $w \neq num + 1 \land cl\ w \neq s? \rightarrow w := w + 1$ od ;
     if $w \in 1 .. num \land ex\ w = \mathsf{false} \rightarrow ex := ex \oplus \{w \mapsto \mathsf{true}\}$
     [] $w = num + 1 \lor ex\ w = \mathsf{true} \rightarrow resp! := missing$
     fi
]|
```

Figure 4.2: Implementation of *Complete_1*

$w = num + 1 \land s? \notin \{\ i : 1 .. num \bullet cl\ i\ \}$ are considered separately.

As *Enrol_ok_1* $\land$ *Success*, *Compl_ok_1* $\land$ *Success* can be converted to the sequential composition *Compl_ok_1* ; *Success* by an application of *sconjC*. In order to refine *Compl_ok_1* to an assignment, first we apply *bC* to this schema, and then use *abA* (absorb assumption) to obtain the specification statement below, whose precondition incorporates the assumption preceding *Compl_ok_1* $\land$ *Success* in the program generated by the application of *sdisjC* to *Complete_1*.

$$\begin{matrix} cl, \\ ex, \\ num \end{matrix} : \left[\begin{array}{l} w \in 1 .. num \land cl\ w = s? \land ex\ w = \mathsf{false} \land ((1 .. num) \lhd cl) \in (\mathbb{N} \rightarrowtail Student), \\ \left(\begin{array}{l} ((1 .. num') \lhd cl') \in (\mathbb{N} \rightarrowtail Student) \\ \exists i : 1 .. num \bullet \\ \quad cl\ i = s? \land ex\ i = \mathsf{false} \land cl' = cl \land ex' = ex \oplus \{i \mapsto \mathsf{true}\} \land num' = num \end{array} \right) \end{array} \right]$$

This program can be refined to the assignment that follows.

$\sqsubseteq$ *assigI*
  $ex := ex \oplus \{w \mapsto \mathsf{true}\}$

The interesting part of the proof-obligation generated by this application of *assigI* consists of showing that the existential quantification below holds under the assumption that the precondition of the above specification statement is satisfied.

$$\exists i : 1 .. num \bullet cl\ i = s? \land ex\ i = \mathsf{false} \land ex \oplus \{w \mapsto \mathsf{true}\} = ex \oplus \{i \mapsto \mathsf{true}\}$$

Since $w \in 1 .. num$, $cl\ w = s?$, and $ex\ w = \mathsf{false}$, it is clear that w satisfies the requirements imposed by this existential quantification.

The schema *Missing_1* can be refined to $resp! := missing$ in much the same way as *Full_1* has been refined to $resp! := full$ (and *Found_1* to $resp! := found$) in the development of *Enrol_1*. The code for *Complete_1* is presented in Figure 4.2.

The development of *Leave_1* is again similar. For brevity, we do not present the variable block that we introduce with an application of *sdisjC*. It also introduces w and the guards of the alternation in its body are $w \in 1 .. num$ and $w = num + 1$. In what follows, we present only the refinement of *Leave_ok_1*.

Initially, we apply *bC* to this schema. Afterwards, we incorporate in the precondition of the resulting specification statement the assumption that reflects the characterisation of w and the guard of the branch in which *Leave_ok_1* occurs. This is accomplished by an application of *abA*. The program that we obtain is presented below.

$$cl, ex, num : \left[\begin{array}{l} w \in 1 \mathrel{..} num \land cl\ w = s? \land ((1 \mathrel{..} num) \lhd cl) \in (\mathbb{N} \rightarrowtail\!\!\!\!\rightarrow Student), \\ \left(\begin{array}{l} ((1 \mathrel{..} num') \lhd cl') \in (\mathbb{N} \rightarrowtail\!\!\!\!\rightarrow Student) \\ \exists\, i : 1 \mathrel{..} num \bullet cl\ i = s? \land \\ \quad (1 \mathrel{..} i-1) \lhd cl' = (1 \mathrel{..} i-1) \lhd cl \land \\ \quad (1 \mathrel{..} i-1) \lhd ex' = (1 \mathrel{..} i-1) \lhd ex \land \\ \quad (i \mathrel{..} num-1) \lhd cl' = (i+1 \mathrel{..} num) \lhd cl \land \\ \quad (i \mathrel{..} num-1) \lhd ex' = (i+1 \mathrel{..} num) \lhd ex \land \\ \quad num' = num - 1 \end{array} \right) \end{array} \right]$$

The precondition of this specification statement establishes that $cl\ w = s?$. Moreover, the characterisation of cl' and num' and the fact that the invariant holds for cl and num imply that it holds for cl' and num' as well. Consequently, we can use *sP* (strengthen postcondition) to simplify the postcondition of this specification statement. If afterwards we apply *wP* (weakening precondition) to eliminate the invariant from the precondition of the resulting specification statement, we obtain the program below.

$$cl, ex, num : \left[w \in 1 \mathrel{..} num \land cl\ w = s?, \left(\begin{array}{l} (1 \mathrel{..} w-1) \lhd cl' = (1 \mathrel{..} w-1) \lhd cl \\ (1 \mathrel{..} w-1) \lhd ex' = (1 \mathrel{..} w-1) \lhd ex \\ (w \mathrel{..} num-1) \lhd cl' = (w+1 \mathrel{..} num) \lhd cl \\ (w \mathrel{..} num-1) \lhd ex' = (w+1 \mathrel{..} num) \lhd ex \\ num' = num - 1 \end{array} \right) \right]$$

This program can be implemented by an iteration that shifts the elements of *cl* and *ex*, and an assignment that decrements *num*. The assignment can be introduced by an application of the *fassigI* (following assignment introduction) law.

$\sqsubseteq$ *fassigI*

$$\begin{array}{l} \begin{array}{l} cl, \\ ex, \\ num \end{array} : \left[w \in 1 \mathrel{..} num \land cl\ w = s?, \left(\begin{array}{l} (1 \mathrel{..} w-1) \lhd cl' = (1 \mathrel{..} w-1) \lhd cl \\ (1 \mathrel{..} w-1) \lhd ex' = (1 \mathrel{..} w-1) \lhd ex \\ (w \mathrel{..} num-1) \lhd cl' = (w+1 \mathrel{..} num) \lhd cl \\ (w \mathrel{..} num-1) \lhd ex' = (w+1 \mathrel{..} num) \lhd ex \\ num' = num \end{array} \right) \right]; \quad \lhd \\ num := num - 1 \end{array}$$

Since *num* is not modified by the specification statement, we can simplify it by applying the *cfR* (contract frame) law to remove *num* from its frame.

$\sqsubseteq$ *cfR*

$$cl, ex : \left[w \in 1 \mathrel{..} num \land cl\ w = s?, \left(\begin{array}{l} (1 \mathrel{..} w-1) \lhd cl' = (1 \mathrel{..} w-1) \lhd cl \\ (1 \mathrel{..} w-1) \lhd ex' = (1 \mathrel{..} w-1) \lhd ex \\ (w \mathrel{..} num-1) \lhd cl' = (w+1 \mathrel{..} num) \lhd cl \\ (w \mathrel{..} num-1) \lhd ex' = (w+1 \mathrel{..} num) \lhd ex \end{array} \right) \right]$$

In order to introduce the iteration, we need a variable to range over the indexes of *cl* and *ex*. We introduce the variable i by applying the *vrbI* (variable introduction) law.

$\sqsubseteq$ *vrbI*

$$
\begin{array}{l}
[\![\, \mathbf{var}\ i : 1 \mathrel{..} max \bullet \\
\qquad \begin{array}{l} i, \\ cl, \\ ex \end{array} : \left[w \in 1 \mathrel{..} num \land cl\ w = s?, \left(\begin{array}{l}
(1 \mathrel{..} w-1) \lhd cl' = (1 \mathrel{..} w-1) \lhd cl \\
(1 \mathrel{..} w-1) \lhd ex' = (1 \mathrel{..} w-1) \lhd ex \\
(w \mathrel{..} num-1) \lhd cl' = (w+1 \mathrel{..} num) \lhd cl \\
(w \mathrel{..} num-1) \lhd ex' = (w+1 \mathrel{..} num) \lhd ex
\end{array} \right) \right] \\
]\!]
\end{array}
$$

The *seqcI* (sequential composition introduction) law is used to introduce the iteration invariant. The restriction over i that is imposed by it ($w \leq i \leq num$) is normally introduced separately by means of a variable block with invariant in [45]. Since this block is not available in ZRC, we have to deal with a slightly longer iteration invariant.

$\sqsubseteq$ *seqcI*

$$
\begin{array}{l}
[\![\, \mathbf{con}\ CL : 1 \mathrel{..} max \rightarrow Student;\ EX : 1 \mathrel{..} max \rightarrow Boolean \bullet \\
\qquad i, cl, ex : \left[\begin{array}{l}
w \in 1 \mathrel{..} num \land cl\ w = s?, \\
\left(\begin{array}{l}
\forall j : 1 \mathrel{..} w-1 \bullet cl'\ j = cl\ j \land ex'\ j = ex\ j \\
\forall j : w \mathrel{..} i'-1 \bullet cl'\ j = cl\ (j+1) \land ex'\ j = ex\ (j+1) \\
\forall j : i' \mathrel{..} num \bullet cl'\ j = cl\ j \land ex'\ j = ex\ j \\
w \leq i' \leq num
\end{array} \right)
\end{array} \right] ; \qquad \lhd \\
\qquad i, cl, ex : \left[\begin{array}{l}
\left(\begin{array}{l}
\forall j : 1 \mathrel{..} w-1 \bullet cl\ j = CL\ j \land ex\ j = EX\ j \\
\forall j : w \mathrel{..} i-1 \bullet cl\ j = CL\ (j+1) \land ex\ j = EX\ (j+1) \\
\forall j : i \mathrel{..} num \bullet cl\ j = CL\ j \land ex\ j = EX\ j \\
w \leq i \leq num
\end{array} \right), \\
\left(\begin{array}{l}
(1 \mathrel{..} w-1) \lhd cl' = (1 \mathrel{..} w-1) \lhd CL \\
(1 \mathrel{..} w-1) \lhd ex' = (1 \mathrel{..} w-1) \lhd EX \\
(w \mathrel{..} num-1) \lhd cl' = (w+1 \mathrel{..} num) \lhd CL \\
(w \mathrel{..} num-1) \lhd ex' = (w+1 \mathrel{..} num) \lhd EX
\end{array} \right)
\end{array} \right] \qquad (i) \\
]\!]
\end{array}
$$

We can implement the first specification statement by initialising i with w.

$\sqsubseteq$ *assigI*

$i := w$

By not changing cl and ex, we establish the first and third conjunct of the postcondition of this specification statement in a trivial way; by assigning w to i, we also establish the second conjunct, which, when i' is w, becomes a qnantification over the empty set: $w \mathrel{..} w-1$; finally $w \leq num$ is a consequence of $w \in 1 \mathrel{..} num$, which is a conjunct of the precondition of the specification statement. These comments account for the proof-obligation associated with the above application of *assigI*.

Before applying *itI* (iteration introduction), we have to use *sP* to rewrite the postcondition of $(\imath)$.

$$
\begin{array}{l}
(\imath) \sqsubseteq sP \\
\quad i, cl, ex : \left[\begin{pmatrix} \forall j : 1 \mathrel{..} w-1 \bullet cl\ j = CL\ j \wedge ex\ j = EX\ j \\ \forall j : w \mathrel{..} i-1 \bullet cl\ j = CL\ (j+1) \wedge ex\ j = EX\ (j+1) \\ \forall j : i \mathrel{..} num \bullet cl\ j = CL\ j \wedge ex\ j = EX\ j \\ w \leq i \leq num \end{pmatrix}, \begin{pmatrix} \forall j : 1 \mathrel{..} w-1 \bullet cl'\ j = CL\ j \wedge ex'\ j = EX\ j \\ \forall j : w \mathrel{..} i'-1 \bullet cl'\ j = CL\ (j+1) \wedge ex'\ j = EX\ (j+1) \\ \forall j : i' \mathrel{..} num \bullet cl'\ j = CL\ j \wedge ex'\ j = EX\ j \\ w \leq i' \leq num \wedge i' = num \end{pmatrix} \right]
\end{array}
$$

It is not difficult to see that when the iteration invariant holds and $i = num$, the requirements imposed by the postcondition of $(\imath)$ are satisfied.

$$
\begin{array}{l}
\sqsubseteq itI \\
\quad \mathbf{do}\ i \neq num \rightarrow \\
\qquad \begin{matrix} i, \\ cl, \\ ex \end{matrix} : \left[\begin{pmatrix} \forall j : 1 \mathrel{..} w-1 \bullet cl\ j = CL\ j \wedge ex\ j = EX\ j \\ \forall j : w \mathrel{..} i-1 \bullet cl\ j = CL\ (j+1) \wedge ex\ j = EX\ (j+1) \\ \forall j : i \mathrel{..} num \bullet cl\ j = CL\ j \wedge ex\ j = EX\ j \\ w \leq i \leq num \wedge i \neq num \end{pmatrix}, \begin{pmatrix} \forall j : 1 \mathrel{..} w-1 \bullet cl'\ j = CL\ j \wedge ex'\ j = EX\ j \\ \forall j : w \mathrel{..} i'-1 \bullet cl'\ j = CL\ (j+1) \wedge ex'\ j = EX\ (j+1) \\ \forall j : i' \mathrel{..} num \bullet cl'\ j = CL\ j \wedge ex'\ j = EX\ j \\ w \leq i' \leq num \wedge num - i' < num - i \end{pmatrix} \right] \\
\quad \mathbf{od}
\end{array}
$$

The body of the iteration can be implemented by the assignment below.

$$
\begin{array}{l}
\sqsubseteq assigI \\
\quad cl, ex, i := cl \oplus \{i \mapsto cl\ (i+1)\}, ex \oplus \{i \mapsto ex\ (i+1)\}, i+1
\end{array}
$$

The proof-obligation generated by this application of *assigI* consists of showing that the assignment above preserves the iteration invariant when the guard $i \neq num$ holds. Namely, we have to prove an implication whose antecedent is the conjunction of the predicates (a_1) to (a_5) below.

(a_1) $\forall j : 1 \mathrel{..} w-1 \bullet cl\ j = CL\ j \wedge ex\ j = EX\ j$;

(a_2) $\forall j : w \mathrel{..} i-1 \bullet cl\ j = CL\ (j+1) \wedge ex\ j = EX\ (j+1)$;

(a_3) $\forall j : i \mathrel{..} num \bullet cl\ j = CL\ j \wedge ex\ j = EX\ j$;

(a_4) $w \leq i \leq num$; and

(a_5) $i \neq num$.

The consequent of the implication is the conjunction of (c_1) to (c_5).

```
|[ var $w : 1 \,..\, max + 1$ •
     $w := 1$ ;
     do  $w \neq num + 1 \wedge cl\ w \neq s?$ → $w := w + 1$ od ;
     if $w \in 1 \,..\, num$ →
          |[ var $i : 1 \,..\, max$ •
             $i := w$ ;
             do $i \neq num$ → $cl, ex, i := cl \oplus \{i \mapsto cl\ (i+1)\}, ex \oplus \{i \mapsto ex\ (i+1)\}, i + 1$ od
          ]| ;
          $num := num - 1$
       [] $w = num + 1$ → $resp! := not\_found$
     fi
]|
```

Figure 4.3: Implementation of *Leave_1*

(c$_1$) $\forall j : 1 \,..\, w - 1 \bullet (cl \oplus \{i \mapsto cl\ (i+1)\})\ j = CL\ j \wedge (ex \oplus \{i \mapsto ex\ (i+1)\})\ j = EX\ j$;

(c$_2$) $\forall j : w \,..\, i \bullet (cl \oplus \{i \mapsto cl\ (i+1)\})\ j = CL\ (j+1) \wedge (ex \oplus \{i \mapsto ex\ (i+1)\})\ j = EX\ (j+1)$;

(c$_3$) $\forall j : i + 1 \,..\, num \bullet (cl \oplus \{i \mapsto cl\ (i+1)\})\ j = CL\ j \wedge (ex \oplus \{i \mapsto ex\ (i+1)\})\ j = EX\ j$;

(c$_4$) $w \leq i + 1 \leq num$; and

(c$_5$) $num - (i + 1) < num - i$.

From (a$_1$) and (a$_4$), we can deduce that (c$_1$) holds, since $(cl \oplus \{i \mapsto cl\ (i+1)\})\ j = cl\ j$ and, similarly, $(ex \oplus \{i \mapsto ex\ (i+1)\})\ j = ex\ j$, for any j in the interval from 1 to $i - 1$. Likewise, from (a$_3$), we deduce (c$_3$). For (c$_2$), we have the result below.

$$
\begin{aligned}
& (cl \oplus \{i \mapsto cl\ (i+1)\})\ i \\
& = cl\ (i+1) && \text{[by a property of functions]} \\
& = CL\ (i+1) && \text{[by (a}_3\text{), (a}_4\text{), and (a}_5\text{)]}
\end{aligned}
$$

Similarly, $(ex \oplus \{i \mapsto ex\ (i+1)\})\ i = EX\ (i+1)$. From this and (a$_2$), we infer that (c$_2$) holds. As a consequence of (a$_4$) and (a$_5$), we have that $w \leq i < num$: (c$_4$) follows from this; (c$_5$) is trivial.

As we observed earlier on, we had to define an iteration invariant which is longer than that we could define if we had variable blocks with invariants in ZRC. As a consequence, the proof-obligations that were generated during the refinement were longer as well. Namely, we had to add (c$_4$) in the previous proof-obligation and had to observe that $w \leq num$ when assigning w to i. Had we introduced the constraint $w \leq i \leq num$ in a variable block with invariant, this could be avoided. However, in a later stage, when removing the invariants, we would have to prove that both (c$_4$) and $w \leq num$ hold anyway. These proofs would require further manipulation of the program already obtained and therefore would lead to a longer development. Of course, they could be omitted if regarded as trivial; this is the strategy employed in [45].

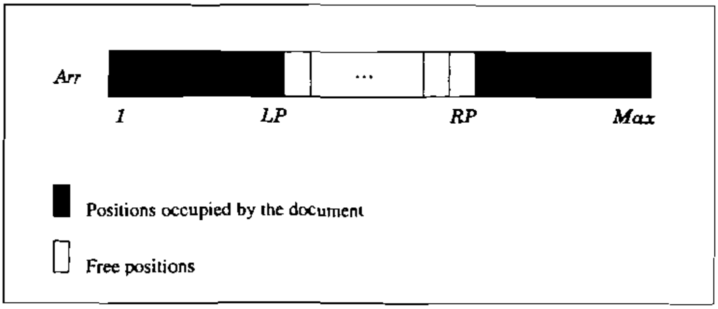

Figure 4.4: State of the Text Editor

The constants CL and EX are not in use anymore and so we can use *conR* (constant removal) to eliminate them. The resulting collected code is shown in Figure 4.3.

4.2 The Text Editor

Our second case stndy is a screen-oriented text editor which, as said before, has heen specified by Neilson in [51]. There, Neilson uses what is called a hierarchical approach to specification: he defines the state and operations of the text editor incrementally, and groups the definitions in levels. At each level a new model (state and operations) is defined: the new state includes that of the previous level, and some of the new operations are promotions of previous levels' operations. There are nine levels and each of them contemplates a different aspect of the text editor; the ninth level defines it as a whole. In this section we consider the states aud some operations of the first two levels.

As already remarked, in [51] a C implementation is developed for the text editor. The levels of the specification are considered separately and, in each case, the first development step is data refinement. Here, we actually consider the resulting concrete specifications.

The documents manipulated by the text editor are sequences of characters or, more precisely, elements of a given set *Char*. A global constant *Max* determines the maximum size of these documents.

The (concrete) state at level 1 can be specified as we show below.

ConcDoc1

$Arr : 1 \mathrel{..} Max ⇸ Char$
$LP, RP, CP : 0 \mathrel{..} Max$

$LP \leq RP$
$CP \leq Max + LP - RP$

The component *Arr* is an array that holds the document (sequence of characters) being edited; LP, RP, and CP are pointers. The document is in the positions from 1 to LP and from $RP + 1$ to *Max* of *Arr* (see Figure 4.4), and CP is the cursor position: an index of this subsequence of *Arr*.

The operation that moves the cursor to the left by a character can be specified as follows.

$$\begin{array}{|l}
\hline
\multicolumn{1}{l}{\mathit{LeftMvChar}_{Doc1}\,C} \\
\Delta \mathit{ConcDoc1} \\
\hline
CP \neq 0 \\
Arr' = Arr \\
LP' = LP \\
RP' = RP \\
CP' = CP - 1 \\
\hline
\end{array}$$

This operation is partial: the cursor can be moved to the left if it is not at the top of the document ($CP \neq 0$). In this case, moving the cursor to the left corresponds to decrementing CP.

In [51] the error cases of all abstract operations are treated using the Oxford style of writing Z specifications. The concrete operations presented there, on the other hand, correspond to the robust abstract operations, but are not specified in a structured way. Here, for the sake of conciseness, we consider concrete operations which correspond to successful cases of abstracts operations. The longer concrete operations of [51] can be refined by, for instance, writing them using the Oxford style of error treatment and applying the strategy exemplified in the preceding section. Alternatively, they can be transformed into specification statements using the bC (basic conversion) law and refined to alternations. In both approaches, operations that contemplate successful cases eventually emerge in the development, and can be refined as we show here.

By applying bC to $\mathit{LeftMvChar}_{Doc1}\,C$, we get the following specification statement.

$\sqsubseteq bC$

$$\begin{array}{l} Arr, \\ LP, \\ RP, \\ CP \end{array} : \left[\begin{array}{l} LP \leq RP \wedge CP \leq Max + LP - RP \wedge CP \neq 0, \\ \left(\begin{array}{l} LP' \leq RP' \wedge CP' \leq Max + LP' - RP' \\ CP \neq 0 \wedge Arr' = Arr \wedge LP' = LP \wedge RP' = RP \wedge CP' = CP - 1 \end{array} \right) \end{array} \right]$$

This program can be implemented by an assignment.

$\sqsubseteq \mathit{assigI}$

$CP := CP - 1$

The proof-obligation generated by this application of *assigI* is trivial.

Every operation that modifies the document is specified by a composition whose first schema is *Standardize*. This is an operation that sets the state to a standard configuration without modifying either the document or the cursor position. The second schema of the composition defines the effect of the operation on a state in this standard configuration.

The operation *Standardize* can be defined as shown below.

$$\begin{array}{|l}
\hline
\multicolumn{1}{l}{\mathit{Standardize}} \\
\Delta \mathit{ConcDoc1} \\
\hline
LP' = CP \\
CP' = CP \\
(1 \mathrel{..} LP' \cup RP' + 1 \mathrel{..} Max) \upharpoonright Arr' = (1 \mathrel{..} LP \cup RP + 1 \mathrel{..} Max) \upharpoonright Arr \\
\hline
\end{array}$$

When the state is in the standard configuration, CP and LP are equal. Operations that modify the document can be more easily specified under the assumption that the state is in this configuration because changes are always made at the cursor position.

As an example, we take the *LeftDeleteCharC* operation which deletes the character to the left of the cursor.

$$LeftDeleteCharC \mathrel{\widehat=} Standardize \mathbin{⨟} LeftDeleteCharCStandard$$

The operation *LeftDeleteCharCStandard* can be specified as follows.

$$\begin{array}{|l}
\textit{LeftDeleteCharCStandard} \\
\hline
\Delta ConcDoc1 \\
\hline
CP \neq 0 \\
Arr' = Arr \\
LP' = LP - 1 \\
RP' = RP \\
CP' = CP - 1 \\
\hline
\end{array}$$

The precondition of this operation is $CP \neq 0 \land LP \neq 0$. If the cursor is at the top of the document ($CP = 0$), there is no character to its left to be deleted. If, otherwise, $CP \neq 0$, as the state is assumed to be in the standard configuration, the character to the left of the cursor can be removed by simply decrementing LP and CP.

By applying the law *scompC* (schema composition conversion) to *LeftDeleteCharC* we can transform it into a sequential program composition.

$$\begin{array}{l}
\quad LeftDeleteCharC \\
\sqsubseteq \ scompC \\
\quad Standardize \ ; \ LeftDeleteCharCStandard
\end{array}$$

The precondition of *LeftDeleteCharC* is $CP \neq 0$. Therefore, the proof-obligation generated by the above application of *scompC* consists of proving that $CP \neq 0$ and *Standardize* imply $CP' \neq 0$ and $LP' \neq 0$. Since *Standardize* does not modify CP and sets LP to CP, this implication can be easily established.

The result of applying *bC* to *Standardize* is the following specification statement.

$$\begin{array}{l} Arr, \\ LP, \\ RP, \\ CP \end{array} : \left[\begin{array}{l} LP \leq RP \land CP \leq Max + LP - RP, \\ \left(\begin{array}{l} LP' \leq RP' \land CP' \leq Max + LP' - RP' \\ LP' = CP \\ CP' = CP \\ (1 \mathinner{..} LP' \cup RP' + 1 \mathinner{..} Max) \upharpoonleft Arr' = (1 \mathinner{..} LP \cup RP + 1 \mathinner{..} Max) \upharpoonleft Arr \end{array} \right) \end{array} \right]$$

We implement *Standardize* with an alternation that distinguishes the cases $LP \geq CP$ and $LP \leq CP$.

Before introducing it, however, we remove CP from the frame of the above specification statement.

$\sqsubseteq$ *cfR*

$$\begin{array}{l} Arr, \\ LP, \\ RP \end{array} : \left[\begin{array}{l} LP \le RP \wedge CP \le Max + LP - RP, \\ \left(\begin{array}{l} LP' \le RP' \wedge CP \le Max + LP' - RP' \\ LP' = CP \\ (1\,..\,LP' \cup RP' + 1\,..\,Max) \rceil Arr' = (1\,..\,LP \cup RP + 1\,..\,Max) \rceil Arr \end{array} \right) \end{array} \right]$$

$\sqsubseteq$ *altI*

if $LP \ge CP \rightarrow$

$$\begin{array}{l} Arr, \\ LP, \\ RP \end{array} : \left[\begin{array}{l} LP \ge CP \wedge LP \le RP \wedge CP \le Max + LP - RP, \\ \left(\begin{array}{l} LP' \le RP' \wedge CP \le Max + LP' - RP' \\ LP' = CP \\ (1\,..\,LP' \cup RP' + 1\,..\,Max) \rceil Arr' = (1\,..\,LP \cup RP + 1\,..\,Max) \rceil Arr \end{array} \right) \end{array} \right] \lhd$$

$[]\ LP \le CP \rightarrow$

$$\begin{array}{l} Arr, \\ LP, \\ RP \end{array} : \left[\begin{array}{l} LP \le CP \wedge LP \le RP \wedge CP \le Max + LP - RP, \\ \left(\begin{array}{l} LP' \le RP' \wedge CP \le Max + LP' - RP' \\ LP' = CP \\ (1\,..\,LP' \cup RP' + 1\,..\,Max) \rceil Arr' = (1\,..\,LP \cup RP + 1\,..\,Max) \rceil Arr \end{array} \right) \end{array} \right] (i)$$

fi

If $LP \ge CP$, *Standardize* can be implemented by an iteration that moves the part of the document in the positions between $CP + 1$ and LP to the right and joins it to the part in the positions from $RP + 1$ to Max. In order to express the invariant of this iteration, we need logical constants.

$\sqsubseteq$ *fiV*

$[\![$ **con** $ARRC : 1\,..\,Max \rightarrow Char;\ LPC, RPC : 0\,..\,Max\ \bullet$

$$\begin{array}{l} Arr, \\ LP, \\ RP \end{array} : \left[\begin{array}{l} \left(\begin{array}{l} LP \ge CP \wedge LP \le RP \wedge CP \le Max + LP - RP \\ ARRC = Arr \wedge LPC = LP \wedge RPC = RP \end{array} \right), \\ \left(\begin{array}{l} LP' \le RP' \wedge CP \le Max + LP' - RP' \\ LP' = CP \\ (1\,..\,LP' \cup RP' + 1\,..\,Max) \rceil Arr' = (1\,..\,LP \cup RP + 1\,..\,Max) \rceil Arr \end{array} \right) \end{array} \right]$$

$]\!]$

Using *sP* (strengthen postcondition) and *wP* (weakening precondition) we can refine the above specification statement to that presented below, which is written in a form appropriate to the application of the *itI* (iteration introduction) law.

$$\begin{array}{l} Arr, \\ LP, \\ RP \end{array} : \left[\begin{array}{l} \left(\begin{array}{l} LP \le RP \wedge CP \le Max + LP - RP \\ (1\,..\,LP \cup RP + 1\,..\,Max) \rceil Arr = (1\,..\,LPC \cup RPC + 1\,..\,Max) \rceil ARRC \\ LP \ge CP \end{array} \right), \\ \left(\begin{array}{l} LP' \le RP' \wedge CP \le Max + LP' - RP' \\ (1\,..\,LP' \cup RP' + 1\,..\,Max) \rceil Arr' = (1\,..\,LPC \cup RPC + 1\,..\,Max) \rceil ARRC \\ LP' \ge CP \\ LP' = CP \end{array} \right) \end{array} \right]$$

The proof-obligations generated by *sP* and *wP* in this case are trivial. The variant of the iteration

```
if $LP \geq CP \rightarrow$
    do $LP \neq CP \rightarrow Arr, LP, RP := Arr \oplus \{RP \mapsto Arr\ LP\}, LP - 1, RP - 1$ od
[] $LP \leq CP \rightarrow$
    do $LP \neq CP \rightarrow Arr, LP, RP := Arr \oplus \{LP + 1 \mapsto Arr\ (RP + 1)\}, LP + 1, RP + 1$ od
fi
```

Figure 4.5: Implementation of *Standardize*

is $LP - CP$.

$\sqsubseteq$ *itI*

$$\textbf{do}\ LP \neq CP \rightarrow$$

$$\begin{array}{l} Arr, \\ LP, : \\ RP \end{array}\left[\begin{pmatrix} LP \leq RP \wedge CP \leq Max + LP - RP \\ (1\,..\,LP \cup RP + 1\,..\,Max) \upharpoonright Arr = (1\,..\,LPC \cup RPC + 1\,..\,Max) \upharpoonright ARRC \\ LP > CP \end{pmatrix}, \begin{pmatrix} LP' \leq RP' \wedge CP \leq Max + LP' - RP' \\ (1\,..\,LP' \cup RP' + 1\,..\,Max) \upharpoonright Arr' = (1\,..\,LPC \cup RPC + 1\,..\,Max) \upharpoonright ARRC \\ LP' \geq CP \wedge LP' - CP < LP - CP \end{pmatrix}\right] \lhd$$

$$\textbf{od}$$

The body of this iteration is refined by the following assignment.

$\sqsubseteq$ *assigI*

$$Arr, LP, RP := Arr \oplus \{RP \mapsto Arr\ LP\}, LP - 1, RP - 1$$

The interesting part of the proof-obligation generated by this application of *assigI* (assignment introduction) consists of showing that the predicate below is satisfied when the precondition of the above specification statement holds.

$$(1\,..\,LP - 1 \cup RP\,..\,Max) \upharpoonright (Arr \oplus \{RP \mapsto Arr\ LP\}) = (1\,..\,LPC \cup RPC + 1\,..\,Max) \upharpoonright ARRC$$

We establish this equality as follows.

$$\begin{array}{ll}
(1\,..\,LP - 1 \cup RP\,..\,Max) \upharpoonright (Arr \oplus \{RP \mapsto Arr\ LP\}) & \\
= (1\,..\,LP - 1 \cup \{RP\} \cup RP + 1\,..\,Max) \upharpoonright (Arr \oplus \{RP \mapsto Arr\ LP\}) & \text{[by a property of sets]} \\
= ((1\,..\,LP - 1) \upharpoonright (Arr \oplus \{RP \mapsto Arr\ LP\}))^\frown(\{RP\} \upharpoonright (Arr \oplus \{RP \mapsto Arr\ LP\}))^\frown & \\
\quad ((RP + 1\,..\,Max) \upharpoonright (Arr \oplus \{RP \mapsto Arr\ LP\})) & \text{[by a property of } \upharpoonright] \\
= ((1\,..\,LP - 1) \upharpoonright Arr)^\frown \langle Arr\ LP \rangle^\frown ((RP + 1\,..\,Max) \upharpoonright Arr) & \text{[by } LP \leq RP] \\
= ((1\,..\,LP - 1) \upharpoonright Arr)^\frown (\{LP\} \upharpoonright Arr)^\frown ((RP + 1\,..\,Max) \upharpoonright Arr) & \text{[by a property of } \upharpoonright] \\
= (1\,..\,LP - 1 \cup \{LP\} \cup RP + 1\,..\,Max) \upharpoonright Arr & \text{[by a property of } \upharpoonright] \\
= (1\,..\,LP \cup RP + 1\,..\,Max) \upharpoonright Arr & \text{[by a property of sets]} \\
= (1\,..\,LPC \cup RPC + 1\,..\,Max) \upharpoonright ARRC & \text{[by assumption]}
\end{array}$$

Now, since $ARRC$, LPC, and RPC are not in use anymore, their declarations can be removed by an application of *conR* (constant removal). The specification statement (i), which standardises the state when $LP \leq CP$, can be refined to an iteration in a similar way. Figure 4.5 presents the collected code of *Standardize*.

In order to obtain an implementation for *LeftDeleteCharC* we still have to refine the operation *LeftDeleteCharCStandard*. This program can be implemented by $LP, CP := LP - 1, CP - 1$; this assignment can be derived by an application of *bC* and a subsequent application of *assigI*. The proof-obligation that arises is trivial.

The operation that inserts a character in the document is *InsertCharC*. As *LeftDeleteCharC*, it is specified by composing *Standardize* with an operation that acts on a state in the standard configuration. In this case, the operation is *InsertCharCStandard*.

$$InsertCharC \mathrel{\widehat=} Standardize \mathbin{;} InsertCharCStandard$$

The operation *InsertCharCStandard* (*InsertCharC*) takes as input the character $x?$ to be inserted in the document. If this character is a *tab*, it is not inserted. Instead, spaces are inserted until the cursor reaches the next tabstop or the document reaches its maximum size. The case in which $x?$ is a *tab* is distinguished in the specification of *InsertCharCStandard*.

$$InsertCharCStandard \mathrel{\widehat=} InsertNonTabCStandard \lor InsertTabCStandard$$

The operation *InsertNonTabCStandard* inserts in the document a character different from *tab*.

$$\begin{array}{|l}
\text{\_\_ } InsertNonTabCStandard \text{ \_\_\_\_}\\
\Delta ConcDoc1\\
x? : Char\\
\hline
x? \neq tab\\
LP \neq RP\\
Arr = Arr \oplus \{LP + 1 \mapsto x?\}\\
LP' = LP + 1\\
CP' = CP + 1\\
RP' = RP\\
\hline
\end{array}$$

The precondition of this operation is $x? \neq tab \land LP \neq RP$. If $LP = RP$, the document has already got to its maximum size and no additional character can be inserted. The case in which $x? = tab$ is contemplated by *InsertTabCStandard*.

$$\begin{array}{|l}
\text{\_\_ } InsertTabCStandard \text{ \_\_\_\_}\\
\Delta ConcDoc1\\
x? : Char\\
\hline
x? = tab\\
LP \neq RP\\
\mathbf{let}\ lnl == max(\{\ i : 1 \mathbin{..} CP \bullet Arr\ i = nl\ \} \cup \{0\})\\
\quad\ nsp == min\{tabstop - (CP - lnl) \bmod tabstop, RP - LP\} \bullet\\
\quad\quad Arr' = Arr \oplus \{\ i : LP + 1 \mathbin{..} LP + nsp \bullet i \mapsto sp\ \}\\
\quad\quad LP' = LP + nsp\\
\quad\quad CP' = CP + nsp\\
\quad\quad RP' = RP\\
\hline
\end{array}$$

The character *nl* (newline) marks the end of the lines. The local variable *lnl* records the position where the line above that in which the cursor is positioned finishes. If the cursor is in the top line, *lnl* takes the value 0. The variable *nsp* records the number of spaces that must be inserted: either the number of positions between the cursor and the next tabstop or the number of characters that can be inserted in the document before it reaches its maximum size, whichever is smaller. The tabstop positions are those that are exact multiples of the constant *tabstop*.

As *LeftDeleteCharC*, *InsertCharC* can be converted to a sequential program composition using *scompC*.

$$\begin{array}{l} \quad InsertCharC \\ \sqsubseteq scompC \\ \quad Standardize \;;\; InsertCharCStandard \end{array}$$

The precondition of *InsertCharC* and of *InsertCharCStandard* is $LP \neq RP$. Therefore, this application of *scompC* gives rise to the proof-obligation $LP \neq RP \wedge Standardize \Rightarrow LP' \neq RP'$. Since *Standardize* establishes that $(1 \mathrel{..} LP' \cup RP' + 1 \mathrel{..} Max) \upharpoonright Arr' = (1 \mathrel{..} LP \cup RP + 1 \mathrel{..} Max) \upharpoonright Arr$, we deduce that the sizes of these sequences are equal and so $LP' + Max - RP' = LP + Max - RP$. As a consequence, we have that $LP' - RP' = LP - RP$. As $LP < RP$ (by $LP \leq RP$, according to the state invariant, and $LP \neq RP$), then $LP' - RP' < LP - LP = 0$. Therefore, $LP' < RP'$ and, in particular, $LP' \neq RP'$, as required.

We implement the operation *InsertCharCStandard* using an alternation that we introduce applying the law *sdisjC* (schema disjunction conversion).

```
    $InsertCharCStandard$
$\sqsubseteq sdisjC$
    if $x? \neq tab \wedge LP \neq RP \rightarrow InsertNonTabCStandard$
    $[]\ x? = tab \wedge LP \neq RP \rightarrow InsertTabCStandard$
    fi
```

Since $LP \neq RP$ is a conjunct of both guards of this alternation, it can be eliminated by an application of the refinement law *wG* (weakening guards).

```
$\sqsubseteq wG$
    if $x? \neq tab \rightarrow InsertNonTabCStandard$
    $[]\ x? = tab \rightarrow InsertTabCStandard$
    fi
```

It is not difficult to verify that the operation *InsertNonTabCStandard* can be refined to the assignment $Arr, LP, CP := Arr \oplus \{LP + 1 \mapsto x?\}, LP + 1, CP + 1$.

Our first step in the development of *InsertTabCStandard* is the application of *bC*.

$$\begin{array}{l} InsertTabCStandard \\ \sqsubseteq bC \end{array}$$

$$\begin{array}{l}Arr,\\LP,\\RP,\\CP\end{array} : \left[\begin{array}{l} LP \le RP \land CP \le Max + LP - RP \land x? = tab \land LP \ne RP, \\ \left(\begin{array}{l} LP' \le RP' \land CP' \le Max + LP' - RP' \land x? = tab \land LP \ne RP \\ \textbf{let } lnl == max(\{ i : 1 \mathinner{..} CP \bullet Arr\ i = nl \} \cup \{0\}) \\ \quad nsp == min\{tabstop - (CP - lnl) \bmod tabstop, RP - LP\} \bullet \\ \quad\quad Arr' = Arr \oplus \{ i : LP+1 \mathinner{..} LP + nsp \bullet i \mapsto sp \} \\ \quad\quad LP' = LP + nsp \\ \quad\quad CP' = CP + nsp \\ \quad\quad RP' = RP \end{array}\right)\end{array}\right]$$

At this point we introduce the program variables *pvlnl* and *pvnsp*. As the local variables *lnl* and *nsp* defined in the postcondition of the above specification statement, they are used to record the position of the last *nl* that appears before the cursor and the number of spaces to be inserted.

$$\begin{array}{l} \sqsubseteq ivB \\ \quad [\![\, \textbf{var } pvlnl, pvnsp : 0 \mathinner{..} Max \bullet \end{array}$$

$$\begin{array}{l}pvlnl,\\pvnsp,\\Arr,\\LP,\\RP,\\CP\end{array} : \left[\begin{array}{l} LP \le RP \land CP \le Max + LP - RP \land x? = tab \land LP \ne RP, \\ \left(\begin{array}{l} LP' \le RP' \land CP' \le Max + LP' - RP' \land x? = tab \land LP \ne RP \\ \textbf{let } lnl == max(\{ i : 1 \mathinner{..} CP \bullet Arr\ i = nl \} \cup \{0\}) \\ \quad nsp == min\{tabstop - (CP - lnl) \bmod tabstop, RP - LP\} \bullet \\ \quad\quad Arr' = Arr \oplus \{ i : LP+1 \mathinner{..} LP + nsp \bullet i \mapsto sp \} \\ \quad\quad LP' = LP + nsp \\ \quad\quad CP' = CP + nsp \\ \quad\quad RP' = RP \end{array}\right)\end{array}\right] \lhd$$

$$]\!]$$

Using *seqcI* (sequential composition introduction), we refine the body of this variable block to a sequential composition.

$$\sqsubseteq seqcI$$

$$pvlnl : \left[\begin{array}{l} LP \le RP \land CP \le Max + LP - RP \land x? = tab \land LP \ne RP, \\ \left(\begin{array}{l} LP \le RP \land CP \le Max + LP - RP \land x? = tab \land LP \ne RP \\ pvlnl' = max(\{ i : 1 \mathinner{..} CP \bullet Arr\ i = nl \} \cup \{0\}) \end{array}\right)\end{array}\right] ;$$

$$\begin{array}{l}pvlnl,\\pvnsp,\\Arr,\\LP,\\RP,\\CP\end{array} : \left[\begin{array}{l} \left(\begin{array}{l} LP \le RP \land CP \le Max + LP - RP \land x? = tab \land LP \ne RP \\ pvlnl = max(\{ i : 1 \mathinner{..} CP \bullet Arr\ i = nl \} \cup \{0\}) \end{array}\right), \\ \left(\begin{array}{l} LP' \le RP' \land CP' \le Max + LP' - RP' \land x? = tab \land LP \ne RP \\ \textbf{let } lnl == max(\{ i : 1 \mathinner{..} CP \bullet Arr\ i = nl \} \cup \{0\}) \\ \quad nsp == min\{tabstop - (CP - lnl) \bmod tabstop, RP - LP\} \bullet \\ \quad\quad Arr' = Arr \oplus \{ i : LP+1 \mathinner{..} LP + nsp \bullet i \mapsto sp \} \\ \quad\quad LP' = LP + nsp \\ \quad\quad CP' = CP + nsp \\ \quad\quad RP' = RP \end{array}\right)\end{array}\right] \lhd$$

The first program of this sequential composition initialises *pvlnl* with the position of *Arr* in which the last *nl* appearing before the cursor occurs. This program can be implemented using an iteration (see Figure 4.6). Its development presents no surprises and is not presented here. The invariant of the iteration is $nl \notin Arr(\!| \, pvlnl + 1 \mathinner{..} CP \, |\!)$ and the variant is *pvlnl*.

In the development of the second specification statement above we introduce yet another sequential composition.

$$
\begin{array}{l}
\sqsubseteq \; seqcI \\
pvnsp : \left[\begin{array}{l}
\left(\begin{array}{l} LP \leq RP \land CP \leq Max + LP - RP \land x? = tab \land LP \neq RP \\ pvlnl = max(\{ \, i : 1 \mathinner{..} CP \bullet Arr \; i = nl \, \} \cup \{0\}) \end{array} \right), \\
\left(\begin{array}{l} LP \leq RP \land CP \leq Max + LP - RP \land x? = tab \land LP \neq RP \\ pvlnl = max(\{ \, i : 1 \mathinner{..} CP \bullet Arr \; i = nl \, \} \cup \{0\}) \\ pvnsp' = min\{tabstop - (CP - pvlnl) \bmod tabstop, RP - LP\} \end{array} \right)
\end{array} \right] ; \\
\begin{array}{l} pvlnl, \\ pvnsp, \\ Arr, \\ LP, \\ RP, \\ CP \end{array} : \left[\begin{array}{l}
\left(\begin{array}{l} LP \leq RP \land CP \leq Max + LP - RP \land x? = tab \land LP \neq RP \\ pvlnl = max(\{ \, i : 1 \mathinner{..} CP \bullet Arr \; i = nl \, \} \cup \{0\}) \\ pvnsp = min\{tabstop - (CP - pvlnl) \bmod tabstop, RP - LP\} \end{array} \right), \\
\left(\begin{array}{l} LP' \leq RP' \land CP' \leq Max + LP' - RP' \land x? = tab \land LP \neq RP \\ \mathbf{let}\; lnl == max(\{ \, i : 1 \mathinner{..} CP \bullet Arr \; i = nl \, \} \cup \{0\}) \\ \quad nsp == min\{tabstop - (CP - lnl) \bmod tabstop, RP - LP\} \bullet \\ \qquad Arr' = Arr \oplus \{ \, i : LP + 1 \mathinner{..} LP + nsp \bullet i \mapsto sp \, \} \\ \qquad LP' = LP + nsp \\ \qquad CP' = CP + nsp \\ \qquad RP' = RP \end{array} \right)
\end{array} \right] \quad \lhd
\end{array}
$$

The first of these specification statements intialises *pvnsp*. With an application of *assigI* we can refine it to $pvnsp := min\{tabstop - (CP - pvlnl) \bmod tabstop, RP - LP\}$. The proof-obligation originated can be discharged with no difficulties.

The second specification statement inserts the spaces in *Arr*, and adjusts *LP* and *CP*. The appropriate assignment to these pointers can be introduced by *fassigI* (following assignment introduction).

$$
\begin{array}{l}
\sqsubseteq \; fassigI \\
\begin{array}{l} pvlnl, \\ pvnsp, \\ Arr, \\ LP, \\ RP, \\ CP \end{array} : \left[\begin{array}{l}
\left(\begin{array}{l} LP \leq RP \land CP \leq Max + LP - RP \land x? = tab \land LP \neq RP \\ pvlnl = max(\{ \, i : 1 \mathinner{..} CP \bullet Arr \; i = nl \, \} \cup \{0\}) \\ pvnsp = min\{tabstop - (CP - pvlnl) \bmod tabstop, RP - LP\} \end{array} \right), \\
\left(\begin{array}{l} LP' \leq RP' \land CP' \leq Max + LP' - RP' \land x? = tab \land LP \neq RP \\ \mathbf{let}\; lnl == max(\{ \, i : 1 \mathinner{..} CP \bullet Arr \; i = nl \, \} \cup \{0\}) \\ \quad nsp == min\{tabstop - (CP - lnl) \bmod tabstop, RP - LP\} \bullet \\ \qquad Arr' = Arr \oplus \{ \, i : LP + 1 \mathinner{..} LP + nsp \bullet i \mapsto sp \, \} \\ \qquad LP' + pvnsp' = LP + nsp \\ \qquad CP' + pvnsp' = CP + nsp \\ \qquad RP' = RP \end{array} \right)
\end{array} \right] ; \quad \lhd \\
LP, CP := LP + pvnsp, CP + pvnsp
\end{array}
$$

In order to simplify the above specification statement we use the *cfR* (contract frame) law to reduce its frame to Arr, and then apply sP and wP to obtain the program below.

$$Arr : [\text{true}, Arr' = Arr \oplus \{\ i : LP + 1 \mathrel{..} LP + pvnsp \bullet i \mapsto sp\ \}]$$

This specification statement can he implemented using an iteration.

Below, we introduce an auxiliary variable j, which ranges over indices of Arr.

$$\begin{array}{l} \sqsubseteq vrbI \\ \quad [\![\ \mathbf{var}\ j : 0 \mathrel{..} Max \bullet \\ \qquad j, Arr : [\text{true}, Arr' = Arr \oplus \{\ i : LP + 1 \mathrel{..} LP + pvnsp \bullet i \mapsto sp\ \}] \\ \quad]\!] \end{array} \qquad \lhd$$

Using $seqcI$, we introduce the iteration invariant.

$$\begin{array}{l} \sqsubseteq seqcI \\ \quad [\![\ \text{con}\ ARRC : 1 \mathrel{..} Max \to Char \bullet \\ \qquad j, Arr : [\text{true}, Arr' = Arr \oplus \{\ i : j' + 1 \mathrel{..} pvnsp \bullet LP + i \mapsto sp\ \}]\ ; \\ \qquad j, Arr : \left[\begin{array}{l} Arr = ARRC \oplus \{\ i : j + 1 \mathrel{..} pvnsp \bullet LP + i \mapsto sp\ \}, \\ Arr' = ARRC \oplus \{\ i : LP + 1 \mathrel{..} LP + pvnsp \bullet i \mapsto sp\ \} \end{array} \right] \\ \quad]\!] \end{array}$$

It is not difficult to verify that the first specification statement is refined by $j := pvnsp$.

After using sP to write the second specification statement above in a form suitable to the application of itI, we can apply this law to introduce the iteration below. Its variant is j.

$$\begin{array}{l} \mathbf{do}\ j \neq 0 \to \\ \qquad j, Arr : \left[\begin{array}{l} Arr = ARRC \oplus \{\ i : j + 1 \mathrel{..} pvnsp \bullet LP + i \mapsto sp\ \} \land j \neq 0, \\ Arr' = ARRC \oplus \{\ i : j' + 1 \mathrel{..} pvnsp \bullet LP + i \mapsto sp\ \} \land 0 \leq j' < j \end{array} \right] \\ \mathbf{od} \end{array}$$

The body of this iteration is refined by the following assignment.

$$\begin{array}{l} \sqsubseteq assigI \\ \quad Arr, j := Arr \oplus \{LP + j \mapsto sp\}, j - 1 \end{array}$$

The proof-obligation that arises from this application of *assigI* can be easily discharged. At this point, *conR* (constant removal) can be used to remove the declaration of *ARRC*. The collected code for *InsertCharCStandard* can he found in Figure 4.6.

The level 2 of the text editor specification defines a model for an unbounded display: basically a non-empty sequence of lines and a screen cursor identified by a pair of coordinates. At this level no length restrictions apply: neither the length of the lines nor the length of the sequence itself are restricted. A line is a sequence of characters that does not include *nl* among its elements.

The concrete state at this level includes that of level 1, *ConcDoc*1, and introduces additional

```
if $x? \neq tab \rightarrow Arr, LP, CP := Arr \oplus \{LP + 1 \mapsto x?\}, LP + 1, CP + 1$
[] $x? = tab \rightarrow$
    |[ var $pvlnl, pvnsp : 0 \,..\, Max$ •
        $pvlnl := CP$ ;
        do $pvlnl \neq 0 \wedge Arr\ pvlnl \neq nl \rightarrow pvlnl := pvlnl - 1$ od ;
        $pvnsp := min\{tabstop - (CP - pvlnl) \bmod tabstop, RP - LP\}$ ;
        |[ var $j : 0 \,..\, Max$ •
            $j := pvnsp$ ;
            do $j \neq 0 \rightarrow Arr, j := Arr \oplus \{LP + j \mapsto sp\}, j - 1$ od
        ]| ;
        $LP, CP := LP + pvnsp, CP + pvnsp$
    ]|
fi
```

Figure 4.6: Implementation of *InsertCharCStandard*

components that represent the unbounded display. Its specification is as follows.

ConcDoc2
$ConcDoc1$
$StartIn, EndIn, DocNL : 0 \,..\, Max$
$CurX, CurY : 1 \,..\, Max + 1$

$StartIn \leq CP \leq EndIn$
$nl \notin \operatorname{ran}((StartIn + 1 \,..\, EndIn) \upharpoonright ((1 \,..\, LP \cup RP + 1 \,..\, Max) \upharpoonright Arr))$
$StartIn \neq 0 \Rightarrow ((1 \,..\, LP \cup RP + 1 \,..\, Max) \upharpoonright Arr)\ StartIn = nl$
$EndIn \neq Max + LP - RP \Rightarrow ((1 \,..\, LP \cup RP + 1 \,..\, Max) \upharpoonright Arr)\ (EndIn + 1) = nl$
$DocNL = \#(((1 \,..\, LP \cup RP + 1 \,..\, Max) \upharpoonright Arr) \rhd \{nl\})$
$CurX = CP - StartIn + 1$
$CurY = \#(((1 \,..\, CP) \upharpoonright ((1 \,..\, LP \cup RP + 1 \,..\, Max) \upharpoonright Arr)) \rhd \{nl\}) + 1$

The sequence of lines of the unbounded display is that determined by the contents of the document in the obvious way. The components *StartIn* and *EndIn* are pointers; *StartIn* determines the position of the document that precedes the start of the cursor line, and *EndIn*, the position where this line ends. The component *DocNL* records the number of occurrences of the *nl* character in the document. Finally, *CurX* and *CurY* record the cursor coordinates in the display. The top left position has coordinates (1,1).

The first four conjuncts of the *ConcDoc2* invariant characterise *StartIn* and *EndIn*. The first conjunct states that the cursor is in the line delimited by these pointers. The second conjunct establishes that they indeed delimit a line: there is no *nl* in the positions from *StartIn* + 1 to *EndIn* of the document. The third and the fourth conjuncts require that this line is as long as possible: if *StartIn* is not pointing to the beginning of the document, then it is pointing to a *nl*; similarly, if *EndIn* is not pointing to the end of the document, then it is pointing to a position preceding a *nl*.

The states at levels 2 and 3 have the same components and differ only in their invariants: the state at level 3 includes that at level 2 and has a stronger invariant. For this reason, perhaps, in [51], the state at level 2 is data-refined during the development of the level 3. Consequently, there *ConcDoc*2 includes two additional components that are related to restrictions introduced at level 3. Here, for brevity, and as we do not consider the level 3, we omit these components.

The operations at level 2 are promotions of the operations at level 1. They are all defined as the conjunction of a level 1 operation with $\Delta\, ConcDoc2$. Below, we present an example.

$$LeftDeleteChar_{Doc2} \mathrel{\widehat=} LeftDeleteCharC \wedge \Delta\, ConcDoc2$$

This is the level 2 operation that deletes the character to the left of the cursor.

Intuition suggests that we can implement a level 2 operation as the sequential composition of the operation that it promotes with a program that updates the additional components of *ConcDoc*2 and so implements $\Delta\, ConcDoc2$. The conversion law *sconjC* (schema conjunction conversion) transforms schema conjunctions into sequential compositions, but cannot be applied to the level 2 operations, since *ConcDoc*1, the state over which the level 1 operations act, and *ConcDoc*2 are not disjoint. Nevertheless, motivated by this example, we present below an additional formulation of *sconjC* which can be used convert the level 2 operations. Its derivation can be found in Appendix D.

Law *sconjC* Schema conjunction conversion (hierarchical specification)

$$\begin{array}{ll}
& \langle \Delta S_1;\ di_1?;\ do_1! \mid p_1 \rangle \wedge \Delta S_2 \\
\sqsubseteq & sconjC \\
& [\![\, \mathbf{con}\ dcl \bullet \\
& \quad \langle \Delta S_1;\ di_1?;\ do_1! \mid p_1 \rangle\ ; \\
& \quad \langle \Xi S_1;\ d_2;\ d_2' \mid inv_1[cl/\alpha d_1] \wedge inv_2[cl/\alpha d_1] \wedge p_1[cl/\alpha d_1][\_/'] \wedge inv_2' \rangle \\
&]\!]
\end{array}$$

provided $(pre_C \wedge inv_1 \wedge inv_1' \wedge p_1) \Rightarrow pre_2'$

where

- $S_1 \mathrel{\widehat=} \langle d_1 \mid inv_1 \rangle$ and $S_2 \mathrel{\widehat=} \langle S_1;\ d_2 \mid inv_2 \rangle$;
- $\mathrm{pre}(\langle \Delta S_1;\ di_1?;\ do_1! \mid p_1 \rangle \wedge \Delta S_2) \equiv pre_C \wedge inv_1 \wedge inv_2 \wedge t_C$;
- $\mathrm{pre}(\Delta S_2 \wedge \Xi S_1) \equiv pre_2 \wedge inv_1 \wedge inv_2 \wedge t_2$;
- t_C and t_2 are the restrictions introduced by $d_1;\ d_2;\ di_1?$ and $d_1;\ d_2$, respectively;
- dcl declares the constants of cl.

Syntactic Restrictions

- The components of ΔS_1 are the only common free variables of $\langle \Delta S_1;\ di_1?;\ do_1! \mid p_1 \rangle$ and ΔS_2;
- The names of cl and cl' are not free in $\langle \Delta S_1;\ di_1?;\ do_1! \mid p_1 \rangle$ and ΔS_2;
- cl and αd_1 have the same length;
- The constants of cl have the same type as the corresponding variables of αd_1.

The state S_2 includes and extends the state S_1. The constants of cl are used to represent the initial values of the corresponding variables of od_1: those held by them before the execution of $\langle \Delta S_1;\ di_1?;\ do_1! \mid p_1\rangle$. The second schema in the sequential composition establishes inv_2 without modifying the components of S_1, but assuming that inv_1 and inv_2 bold before the execution of $\langle \Delta S_1;\ di_1?;\ do_1! \mid p_1\rangle$ and that this program establishes p_1. The proviso guarantees that this is not an impossihle task. The predicates pre_C and pre_2 are supposed to be those that are usually regarded as the precondition of $\langle \Delta S_1;\ di_1?;\ do_1! \mid p_1\rangle \wedge \Delta S_2$ and $\Delta S_2 \wedge \Xi S_1$, respectively.

By way of illustration, we take the level 2 operation that moves the cursor to the left by a character.

$$LeftMvChar_{Doc2}C \mathrel{\widehat=} LeftMvChar_{Doc1}C \wedge \Delta ConcDoe2$$

We can refine $LeftMvChar_{Doc2}C$ to the constant block below by applying $sconjC$.

$$\begin{array}{l}
\quad LeftMvChar_{Doc2}C \\
\sqsubseteq sconjC \\
\quad [\![\mathbf{con}\ ARRC : 1 \,..\, Max \to Char;\ LPC, RPC, CPC : 0 \,..\, Max \bullet \\
\qquad\quad LeftMvChar_{Doc1}C \ ;\ UpdateConcDoc2 \\
\quad]\!]
\end{array}$$

The schema $UpdateConcDoc2$ can he defined as follows.

$$\begin{array}{|l}
\multicolumn{1}{l}{UpdateConcDoc2} \\
\hline
\Xi ConcDoc1 \\
StartIn, EndIn, DocNL, StartIn', EndIn', DocNL' : 0 \,..\, Max \\
CurX, CurY, CurX', CurY' : 1 \,..\, Max + 1 \\
\hline
LPC \leq RPC \wedge CPC \leq Max + LPC - RPC \\
StartIn \leq CPC \leq EndIn \\
nl \notin \operatorname{ran}((StartIn + 1 \,..\, EndIn) \upharpoonleft ((1 \,..\, LPC \cup RPC + 1 \,..\, Max) \upharpoonleft ARRC)) \\
StartIn \neq 0 \Rightarrow ((1 \,..\, LPC \cup RPC + 1 \,..\, Max) \upharpoonleft ARRC)\ StartIn = nl \\
EndIn \neq Max + LPC - RPC \Rightarrow \\
\qquad ((1 \,..\, LPC \cup RPC + 1 \,..\, Max) \upharpoonleft ARRC)\ (EndIn + 1) = nl \\
DocNL = \#(((1 \,..\, LPC \cup RPC + 1 \,..\, Max) \upharpoonleft ARRC) \rhd \{nl\}) \\
CurX = CPC - StartIn + 1 \\
CurY = \#(((1 \,..\, CPC) \upharpoonleft ((1 \,..\, LPC \cup RPC + 1 \,..\, Max) \upharpoonleft ARRC)) \rhd \{nl\}) + 1 \\
CPC \neq 0 \\
Arr = ARRC \wedge LP = LPC \wedge RP = RPC \wedge CP = CPC - 1 \\
StartIn' \leq CP \leq EndIn' \\
nl \notin \operatorname{ran}((StartIn' + 1 \,..\, EndIn') \upharpoonleft ((1 \,..\, LP \cup RP + 1 \,..\, Max) \upharpoonleft Arr)) \\
StartIn' \neq 0 \Rightarrow ((1 \,..\, LP \cup RP + 1 \,..\, Max) \upharpoonleft Arr)\ StartIn' = nl \\
EndIn' \neq Max + LP - RP \Rightarrow ((1 \,..\, LP \cup RP + 1 \,..\, Max) \upharpoonleft Arr)\ (EndIn' + 1) = nl \\
DocNL' = \#(((1 \,..\, LP \cup RP + 1 \,..\, Max) \upharpoonleft Arr) \rhd \{nl\}) \\
CurX' = CP - StartIn' + 1 \\
CurY' = \#(((1 \,..\, CP) \upharpoonleft ((1 \,..\, LP \cup RP + 1 \,..\, Max) \upharpoonleft Arr)) \rhd \{nl\}) + 1 \\
\hline
\end{array}$$

The extra components of $ConcDoc2$, namely, $StartIn$, $EndIn$, $DoeNL$, $CurX$, and $CurY$, are derived components. Their values are well-defined for all possible values of the components of $ConcDoc1$.

```
if $(CP < LP \wedge Arr\ (CP+1) = nl) \vee (CP \geq LP \wedge Arr(RP - LP + (CP+1)) = nl) \rightarrow$
   $StartIn := CP$ ;
   do $StartIn \neq 0 \wedge$
      $((StartIn \leq LP \wedge Arr\ StartIn \neq nl) \vee (StartIn > LP \wedge Arr\ (RP - LP + StartIn) \neq nl)) \rightarrow$
         $StartIn := StartIn - 1$
   od ;
   $EndIn, CurX, CurY := CP, CP - StartIn + 1, CurY - 1$
[] $\neg ((CP < LP \wedge Arr\ (CP+1) = nl) \vee (CP \geq LP \wedge Arr(RP - LP + (CP+1)) = nl)) \rightarrow$
   $CurX := CurX - 1$
fi
```

Figure 4.7: Implementation of *UpdateConcDoc*2

Therefore, pre$(\Delta ConcDoc2 \wedge \Xi ConcDoc1)$ is true and the proof-obligation generated by the above application of *sconjC* is trivial.

For the sake of brevity, we do not refine *UpdateConcDoc*2 here. Since $LeftMvChar_{Doc1}C$ does not change the contents of the document, *UpdateConcDoc*2 does not need to update *DocNL*. The values of *StartIn*, *EndIn*, *CurX*, and *CurY*, however, may have to be changed. The implementation that we present in Figure 4.7 for *UpdateConcDoc*2 performs the necessary modifications taking into account that the invariant held before the execution of $LeftMvChar_{Doc1}C$ and that this operation simply decrements the value of *CP* by 1. This is the information recorded in *UpdateConcDoc*2 using the constants *ARRC*, *LPC*, *RPC*, and *CPC*. The alternation in Figure 4.7 identifies whether or not moving the cursor to the left has changed the cursor line.

4.3 The Airbus Cabin-Illumination System

The last case study that we present here is based on a Z specification presented in [23] for an Airbus cabin-illumination system. This specification has been intentionally written using mostly concrete data types. Therefore, only a few minor modifications are necessary to make it appropriate as a starting point for the development of an implementation for the illumination system using ZRC.

The Airbus cabin is divided into three zones and two entry areas; a zone may, for instance, accommodate the first class or the business class seats. The illumination system provides separate control for each of these parts of the cabin. The lights in a cabin zone or entry area are dimmable; they have three illumination levels. Additionally, the cabin zones may have an extra set of special night lights; if not, the ordinary lights are used to provide a night light service.

The free types *ZONES* and *EA* presented below contain identifiers for the cabin zones and the entry areas.

$$\begin{array}{lll} ZONES & ::= & z1 \mid z2 \mid z3 \\ EA & ::= & fwd \mid aft \end{array}$$

In [23] the zone and entry area identifiers are introduced as constants of a free type *LOCATION*, and *ZONES* and *EA* as abbreviations for the sets $\{z1, z2, z3\}$ and $\{fwd, aft\}$, respectively. Here,

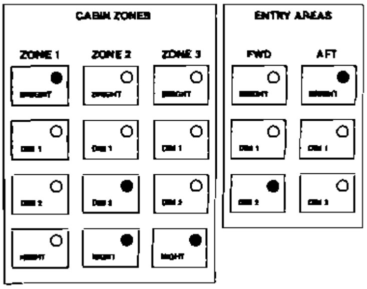

Figure 4.8: Command Panel of the Illumination System

since *LOCATION* is not used in the specification of either the state or the operations we examine, we omit its definition.

Figure 4.8, which has been extracted from [23], presents the panel used by the attendants to command the illumination system. For each of the cabin zones, this panel contains four buttons labelled BRIGHT, DIM1, DIM2, and NIGHT. For each of the entry areas, there are three buttons: BRIGHT, DIM1 and DIM2. A light indicator is associated with each of these buttons. The set DIM_0 defined in the sequel contains constants that represent the light indicators of a particular cabin zone or entry area. These constants are elements of the free type *DIM*, which is used to represent the light brightness levels.

$$\begin{aligned} DIM &::= dim1 \mid dim2 \mid bright \mid off \mid onNl2 \\ DIM_0 &== \{dim1, dim2, bright, off\} \end{aligned}$$

The free type *SWITCH* contains the constants *active* and *passive* which are used to indicate whether or not a light indicator associated to a NIGHT button is on.

$$SWITCH ::= active \mid passive$$

Actually, *SWITCH* is used in this specification for two different purposes. The second use of *SWITCH* is explained later on.

The BRIGHT, DIM1, and DIM2 buttons of the command panel are used to switch on and off and to adjust the level of brightness of the cabin zones and entry areas. The function of the NIGHT buttons and the way in which the night light service is controlled is determined by the value of the global variable *CAM_NLAUTO* defined below.

$$\mid \; CAM\_NLAUTO : FEATURE$$

Its type, *FEATURE*, is specified as follows.

$$FEATURE ::= disabled \mid enabled$$

If the value of *CAM_NLAUTO* is *enabled*, then the illumination system provides a night light autoservice. In this case the night lights and night light indicators in the command panel are automatically switched on (off) when the ordinary lights are switched off (on) and the NIGHT button is used only to switch off the night lights. If, otherwise, *CAM_NLAUTO* is equal to *disabled*, then the NIGHT buttons control the night lights. When a NIGHT button is pressed, the corresponding night light indicator is turned on, and the night lights are turned on if or when the ordinary lights are turned off. When the NIGHT button is pressed again the indicator and the night lights, if necessary, are turned off. Also, when any other button of the same zone is pressed, the night lights are turned off if they are on.

The schema *ZONEINDstate* defined below specifies part of the illumination system state. The component *zoneInd* represents the light indicators associated with the BRIGHT, DIM1, and DIM2 buttons that control cabin zone lights. Since, for each zone, at most one of these indicators is on, *zoneInd* is defined as a total function from *ZONES* to DIM_0. For a zone z, *zoneInd* z is the light indicator that is on in that zone: *dim*1, *dim*2, or *bright*, or takes the value *off* when none of them is on. The component *nlInd* represents the night light indicators; it is a function from *ZONES* to *SWITCH*: *nlInd* z is either *active* or *passive* depending on whether the NIGHT indicator of zone z is on or off.

ZONEINDstate

$$\begin{array}{l}
zoneInd : ZONES \rightarrow DIM_0 \\
nlInd : ZONES \rightarrow SWITCH \\
\hline
\forall z : ZONES \bullet nlInd\ z = active \Rightarrow (zoneInd\ z = off \lor CAM\_NLAUTO = disabled)
\end{array}$$

The invariant in *ZONEINDstate* establishes that, in all zones, if the NIGHT indicator is on, then either the *dim*1, *dim*2, and *bright* indicators are off so that the ordinary lights in the zone are off, or the night light autoservice is disabled so that the NIGHT button has been pressed to pre-select the night light service.

The *dim*1, *dim*2, and *bright* indicators of the entry areas are represented by the state component *eaInd*, which is introduced by the following schema.

EAINDstate

$$\begin{array}{l}
eaInd : EA \rightarrow DIM_0
\end{array}$$

By analogy with *zoneInd*, *eaInd* is a total function from *EA* to DIM_0.

When the Airbus is on the ground, the cabin illumination can be controlled from a MAIN button. Its indicator is represented by the state component *mainInd*, which is introduced by the schema *MAININDstate* that follows.

MAININDstate

$$\begin{array}{l}
ZONEINDstate \\
EAINDstate \\
mainInd : SWITCH \\
\hline
mainInd = passive \Leftrightarrow \\
\quad \mathrm{ran}\ nlInd = \{passive\} \land \mathrm{ran}\ zoneInd = \{off\} \land \mathrm{ran}\ eaInd = \{off\}
\end{array}$$

The type of *mainInd* is *SWITCH*. If *mainInd* is equal to *passive*, the MAIN indicator is off and so are all other indicators. This is the property stated by the invariant of *MAININDstate*.

The lights are identified by addresses in a bus, which, in [23], are elements of a given set $ADDRESS$. Here, in order to obtain a more concrete specification, we define that addresses are numbers in the interval from 1 to $maxad$, a global variable introduced below.

$$\begin{array}{|l}
maxad : \mathbb{N} \\
\hline
maxad > 0
\end{array}$$

The addresses of the lights and night lights in each of the zones and entry areas are identified by tables: partial functions from $1 \mathinner{..} maxad$ to $ZONES$ or EA.

$$\begin{array}{|l}
CAM\_CAB : 1 \mathinner{..} maxad \pfun ZONES \\
CAM\_EA : 1 \mathinner{..} maxad \pfun EA \\
CAM\_NL1 : 1 \mathinner{..} maxad \pfun ZONES \\
CAM\_NL2 : 1 \mathinner{..} maxad \pfun ZONES \\
\hline
CAM\_NL1 \subseteq CAM\_CAB \\
\operatorname{dom} CAM\_CAB \cap (\operatorname{dom} CAM\_EA \cup \operatorname{dom} CAM\_NL2) = \varnothing \\
\operatorname{dom} CAM\_EA \cap \operatorname{dom} CAM\_NL2 = \varnothing
\end{array}$$

The addresses in (the domain of) the table CAM_CAB are those of the ordinary lights in the cabin zones; if the address a is in CAM_CAB, then it identifies a light that is in the zone $CAM\_CAB\ a$. Similarly, CAM_EA distinguishes the addresses of the lights in the entry areas. If the cabin zones have special night lights, then their addresses are recorded in CAM_NL2. Otherwise, CAM_NL1 singles out the ordinary lights that are used to provide the night light service. The addresses in CAM_NL1 are also in CAM_CAB; and the sets of addresses in CAM_CAB, CAM_EA, and CAM_NL2 are pairwise disjoint.

The last component of the illumination system state, ill, represents the cabin zone, entry area, and night lights; it is introduced by the schema $ILLstate$.

$$\begin{array}{|l}
\multicolumn{1}{l}{ILLstate} \\
\hline
ill : 1 \mathinner{..} maxad \fun DIM \\
\hline
\forall a : 1 \mathinner{..} maxad \bullet ill\ a = onNl2 \Rightarrow a \notin (\operatorname{dom} CAM\_CAB \cup \operatorname{dom} CAM\_EA) \\
\hline
\end{array}$$

The light addresses of interest are those in the tables CAM_CAB, CAM_EA, CAM_NL1, and CAM_NL2. The constant $onNl2$ represents the on state of a special night light; it is different from $dim1$, $dim2$, and $bright$, since the special night lights and the ordinary lights are of different types. The invariant of $ILLstate$ asserts that, for every address a, if ill is $onNl2$, then a does not identify an ordinary light: it is either in CAM_NL2 or is an unused address. In [23], ill is defined as a partial function whose domain is the set of addresses in CAM_CAB, CAM_EA, and CAM_NL2. We define it as a total function (an array) as this data type is more readily available in most programming languages.

The first operation of the illumination system that we examine is $MAINop$, which is triggered by pressing the MAIN button. This operation has no effect if the Airbus is not on ground. The global constant $LGEARst$ defined below determines the current state of the landing gear; with this information, it is possible to work out whether or not the Airbus is on the ground.

$$\begin{array}{|l}
LGEARst : LGCIU
\end{array}$$

The free type $LGCIU$ contains three constants that represent the possible states of the landing gear.

$$LGCIU ::= downCompressed \mid downLocked \mid upLocked$$

The Airbus is in the air when the landing gear is either *downLocked* or *upLocked*. This situation is characterised by the schema *MAINisBlocked*.

$$\begin{array}{|l}
\hline
\textit{MAINisBlocked} \\
LGEARst \in \{downLocked, upLoeked\} \\
\hline
\end{array}$$

In this case, *MAINop* does not change the state: it behaves like the operation *NOop* defined below.

$$\begin{array}{|l}
\hline
\textit{NOop} \\
\Xi ZONEINDstate \\
\Xi EAINDstate \\
\Xi MAININDstate \\
\Xi ILLstate \\
\hline
\end{array}$$

If the Airbus is on the ground, the effect of *MAINop* depends on whether the MAIN indicator is on or off. If it is on, it is turned off, and so are all other light indicators. In the specification below, the new values of *zoneInd*, *nlInd*, and *eaInd* are determined by the state invariant.

$$\begin{array}{|l}
\hline
\textit{MAININDopPassive} \\
\Delta MAININDstate \\
\hline
mainInd = active \\
mainInd' = passive \\
\hline
\end{array}$$

The lights themselves are turned off as well.

$$\begin{array}{|l}
\hline
\textit{MAINILLopPassive} \\
\Delta ILLstate \\
\hline
ill' = \{\ a : 1 \mathinner{..} maxad \bullet a \mapsto off\ \} \\
\hline
\end{array}$$

If the MAIN indicator is currently turned off, then *MAINop* reinitialises the system. The MAIN indicator is turned on.

$$\begin{array}{|l}
\hline
\textit{MAININDINITop} \\
MAININDstate' \\
\hline
mainInd' = active \\
\hline
\end{array}$$

The BRIGHT indicators are turned on and the NIGHT indicators are turned off. This is specified by the schemas *ZONEINDINITop* and *EAINDINITop*.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{ZONEINDINITop}} \\ \hline
\textit{ZONEINDstate}' \\ \hline
\textit{zoneInd}' = \{\ z : \textit{ZONES} \bullet z \mapsto \textit{bright}\ \} \\
\textit{nlInd}' = \{\ z : \textit{ZONES} \bullet z \mapsto \textit{passive}\ \} \\ \hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{EAINDINITop}} \\ \hline
\textit{EAINDstate}' \\ \hline
\textit{eaInd}' = \{\ z : \textit{EA} \bullet z \mapsto \textit{bright}\ \} \\ \hline
\end{array}$$

Finally, the ordinary lights are switched to bright and the special night lights, switched off.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{ILLINITop}} \\ \hline
\textit{ILLstate}' \\ \hline
\{\ a : (\operatorname{dom} \textit{CAM\_CAB} \cup \operatorname{dom} \textit{CAM\_EA}) \bullet a \mapsto \textit{bright}\ \} \cup \\
\{\ a : \operatorname{dom} \textit{CAM\_NL2} \bullet a \mapsto \textit{off}\ \} \subseteq \textit{ill}' \\ \hline
\end{array}$$

The initialisation operation is defined as the conjunction of the last four schemas presented above.

$$\textit{INITop} \mathrel{\widehat{=}} \textit{ZONEINDINITop} \land \textit{EAINDINITop} \land \textit{MAININDINITop} \land \textit{ILLINITop}$$

The *MAINop* operation is specified as follows.

$$\begin{array}{ll}
\textit{MAINop} \mathrel{\widehat{=}} & (\textit{MAINisBlocked} \land \textit{NOop}) \lor \\
& (\neg\ \textit{MAINisBlocked} \land \\
& \quad (\textit{MAINILLopPassive} \land \textit{MAININDopPassive} \lor \\
& \quad\ [\textit{MAININDstate} \mid \textit{mainInd} = \textit{passive}] \land \textit{INITop}))
\end{array}$$

The precondition of the first disjunct of *MAINop*, namely, $\textit{MAINisBlocked} \land \textit{NOop}$, can be expressed as $\textit{LGEARst} = \textit{downLocked} \lor \textit{LGEARst} = \textit{upLocked}$; the precondition of the second disjunct of *MAINop* is $\textit{LGEARst} = \textit{downCompressed}$.

By applying the first formulation of *sdisjC* (schema disjunction conversion) to *MAINop*, we can obtain the following alternation.

$$\begin{array}{lr}
\textbf{if}\ \textit{LGEARst} = \textit{downLocked} \lor \textit{LGEARst} = \textit{upLocked} \rightarrow & \\
\qquad \textit{MAINisBlocked} \land \textit{NOop} & \lhd \\
[]\ \textit{LGEARst} = \textit{downCompressed} \rightarrow & \\
\qquad \neg\ \textit{MAINisBlocked} \land & \\
\qquad (\textit{MAINILLopPassive} \land \textit{MAININDopPassive} \lor & (i) \\
\qquad\ [\textit{MAININDstate} \mid \textit{mainInd} = \textit{passive}] \land \textit{INITop}) & \\
\textbf{fi} &
\end{array}$$

If we apply to $\textit{MAINisBlocked} \land \textit{NOop}$ the second formulation of *bC* (basic conversion), which

```
|[ var i : 1 .. maxad + 1 •
     i := 1 ;
     do i ≠ maxad + 1 →
          if i ∈ set → ill := ill ⊕ {i ↦ dim} [] i ∉ set → skip fi ;
          i := i + 1
     od
]|
```

Figure 4.9: Implementation of *updILL*

deals with operations that do not modify the state, we obtain the specification statement below.

$$\begin{array}{l}\sqsubseteq bC \\ :\left[\begin{array}{l} MAININDstate \land ILLstate \land LGEARst = downLocked \lor LGEARst = upLocked, \\ zoneInd' = zoneInd \land nlInd' = nlInd \land eaInd' = eaInd \land mainInd' = mainInd \land ill' = ill \end{array}\right]\end{array}$$

This program can be refined to **skip** using the law *skI* (skip introduction).

$$\begin{array}{l}\sqsubseteq skI \\ \quad \textbf{skip}\end{array}$$

The proof-obligation that arises requires us to prove that the state components are equal to themselves, which obviously is trivial.

The application of the first formulation of bC to the schema (i) generates the following specification statement.

$$\begin{array}{l} zoneInd, \\ nlInd, \\ eaInd, \\ mainInd, \\ ill \end{array} : \left[\begin{array}{l} MAININDstate \land ILLstate \land LGEARst = downCompressed, \\ \left(\begin{array}{l} MAININDstate' \land ILLstate' \land LGEARst = downCompressed \\ \left(\begin{array}{l} mainInd = active \\ mainInd' = passive \\ ill' = \{\ a : 1\,..\,maxad \bullet a \mapsto off\ \} \end{array}\right) \lor \\ \quad \left(\begin{array}{l} mainInd = passive \\ zoneInd' = \{\ z : ZONES \bullet z \mapsto bright\ \} \\ nlInd' = \{\ z : ZONES \bullet z \mapsto passive\ \} \\ eaInd' = \{\ z : EA \bullet z \mapsto bright\ \} \\ mainInd' = active \\ \{\ a : (\operatorname{dom} CAM\_CAB \cup \operatorname{dom} CAM\_EA) \bullet a \mapsto bright\ \} \cup \\ \{\ a : \operatorname{dom} CAM\_NL2 \bullet a \mapsto off\ \} \subseteq ill' \end{array}\right) \end{array}\right) \end{array}\right] \qquad (ii)$$

We implement this program using an alternation that distingnishes the cases $mainInd = active$ and $mainInd = passive$. Before we introduce this alternation, however, we use the law *prcI* (procedure introduction) to declare the procedure *updILL* presented below, which is used later on to update *ill*.

$$
\begin{array}{l}
updILL(1\,..\,maxad, off\,)\ ;\\
zoneInd, nlInd, eaInd, mainInd := \{z1 \mapsto off, z2 \mapsto off, z3 \mapsto off\},\\
\qquad\qquad\qquad\qquad\qquad\qquad \{z1 \mapsto passive, z2 \mapsto passive, z3 \mapsto passive\},\\
\qquad\qquad\qquad\qquad\qquad\qquad \{fwd \mapsto off, aft \mapsto off\}, passive
\end{array}
$$

Figure 4.10: Implementation of (iii)

The procedure block that is introduced by the application of *prcI* has the specification statement (ii) as its main program.

$$updILL \mathrel{\widehat=} (\mathbf{val}\ set : \mathbb{F}(1\,..\,maxad);\ dim : DIM \bullet ill : [\text{true}, ill' = ill \oplus \{\ a : set \bullet a \mapsto dim\ \}])$$

The procedure *updILL* has two value parameters: *set* and *dim*. It updates *ill* by setting to *dim* the brightness level of the lights whose addresses are in *set*. We assume that a data type corresponding to the type constructor $\mathbb{F}$ is available (or has been implemented) in the target programming language. In fact, the majority of the traditional imperative programming languages do not include a type constructor like $\mathbb{F}$, but in the library of most object-oriented programming languages there is a class that defines a set type.

The specification statement in the body of *updILL* can be implemented using an iteration; a possible implementation for this program is presented in Figure 4.9. Its refinement is not difficult and, for the sake of conciseness, is not presented here.

Applying *altI* (alternation introduction) to (ii) and then using *sP* (strengthen postcondition) and *wP* (weakening precondition) to simplify the specification statements in the branches of the resulting alternation, we can obtain the following program.

if $mainInd = active \rightarrow$

$$
\begin{array}{l} zoneInd,\\ nlInd,\\ eaInd,\\ mainInd,\\ ill \end{array} :
\left[\begin{array}{l}
\text{true},\\
\left(\begin{array}{l}
mainInd' = passive\\
\operatorname{ran} nlInd' = \{passive\} \land \operatorname{ran} zoneInd' = \{off\} \land \operatorname{ran} eaInd' = \{off\}\\
ill' = \{\ a : 1\,..\,maxad \bullet a \mapsto off\ \}
\end{array}\right)
\end{array}\right] \qquad (iii)
$$

$[\!]$ $mainInd = passive \rightarrow$

$$
\begin{array}{l} zoneInd,\\ nlInd,\\ eaInd,\\ mainInd,\\ ill \end{array} :
\left[\text{true}, \left(\begin{array}{l}
zoneInd' = \{\ z : ZONES \bullet z \mapsto bright\ \}\\
nlInd' = \{\ z : ZONES \bullet z \mapsto passive\ \}\\
eaInd' = \{\ z : EA \bullet z \mapsto bright\ \}\\
mainInd' = active\\
\{\ a : (\operatorname{dom} CAB\_CAB \cup \operatorname{dom} CAM\_EA) \bullet a \mapsto bright\ \} \cup\\
\{\ a : \operatorname{dom} CAM\_NL2 \bullet a \mapsto off\ \} \subseteq ill'
\end{array}\right)\right] \qquad (iv)
$$

fi

The specification statements (iii) and (iv) can be refined in much the same way and here we proceed to refine only (iv). In Figure 4.10 we present an implementation for (iii).

By applying *fassigI* (following assignment introduction) to (*iv*) we can introduce assignments to *zoneInd*, *nlInd*, *eaInd*, and *mainInd*.

$$
\begin{array}{l}
\sqsubseteq \mathit{fassigI} \\
\quad \begin{array}{l} \mathit{zoneInd}, \\ \mathit{nlInd}, \\ \mathit{eaInd}, \\ \mathit{mainInd}, \\ \mathit{ill} \end{array} : \left[\begin{array}{l} \text{true}, \\ \{\ a : (\operatorname{dom} \mathit{CAM\_CAB} \cup \operatorname{dom} \mathit{CAM\_EA}) \bullet a \mapsto \mathit{bright}\ \} \cup \\ \{\ a : \operatorname{dom} \mathit{CAM\_NL2} \bullet a \mapsto \mathit{off}\ \} \subseteq \mathit{ill}' \end{array} \right] ; \\
\quad \mathit{zoneInd}, \mathit{nlInd}, \mathit{eaInd}, \mathit{mainInd} := \{z1 \mapsto \mathit{bright}, z2 \mapsto \mathit{bright}, z3 \mapsto \mathit{bright}\}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \{z1 \mapsto \mathit{passive}, z2 \mapsto \mathit{passive}, z3 \mapsto \mathit{passive}\}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \{\mathit{fwd} \mapsto \mathit{bright}, \mathit{aft} \mapsto \mathit{bright}\}, \mathit{active}
\end{array}
$$

With a view of updating *ill* using calls to *updILL*, we use *cfR* (contract frame) to reduce the frame of the above specification statement to *ill*, and then apply *sP* to obtain the program below.

$$
\mathit{ill} : \left[\begin{array}{l} \text{true}, \\ \mathit{ill}' = \mathit{ill} \oplus \{\ a : (\operatorname{dom} \mathit{CAM\_CAB} \cup \operatorname{dom} \mathit{CAM\_EA}) \bullet a \mapsto \mathit{bright}\ \} \oplus \\ \qquad \{\ a : \operatorname{dom} \mathit{CAM\_NL2} \bullet a \mapsto \mathit{off}\ \} \end{array} \right]
$$

Two calls to *updILL* are necessary to update *ill* in the required way. We introduce the sequential composition as follows.

$$
\begin{array}{ll}
\sqsubseteq \mathit{seqcI} & \\
\quad [\![\textbf{con}\ \mathit{CILL} : 1 \mathrel{..} \mathit{maxad} \nrightarrow \mathit{DIM} \bullet & \\
\qquad \mathit{ill} : [\text{true}, \mathit{ill}' = \mathit{ill} \oplus \{\ a : (\operatorname{dom} \mathit{CAM\_CAB} \cup \operatorname{dom} \mathit{CAM\_EA}) \bullet a \mapsto \mathit{bright}\ \}] ; & (v) \\
\qquad \mathit{ill} : \left[\begin{array}{l} \mathit{ill} = \mathit{CILL} \oplus \{\ a : (\operatorname{dom} \mathit{CAM\_CAB} \cup \operatorname{dom} \mathit{CAM\_EA}) \bullet a \mapsto \mathit{bright}\ \}, \\ \mathit{ill}' = \mathit{CILL} \oplus \{\ a : (\operatorname{dom} \mathit{CAM\_CAB} \cup \operatorname{dom} \mathit{CAM\_EA}) \bullet a \mapsto \mathit{bright}\ \} \oplus \\ \qquad \{\ a : \operatorname{dom} \mathit{CAM\_NL2} \bullet a \mapsto \mathit{off}\ \} \end{array} \right] & (vi) \\
\quad]\!] &
\end{array}
$$

Using *vS* (value specification) we can refine (*v*) to a parametrised statement which, with an application of *pcallI* (procedure call introduction), can be transformed into a call to *updILL* with parameters ($\operatorname{dom} \mathit{CAM\_CAB} \cup \operatorname{dom} \mathit{CAM\_EA}$) and *off*. The dom operator is not available in most programming languages. We assume, however, that a data type called *Table*, for instance, is used to represent *CAM_CAB*, *CAM_EA*, *CAM_NL1*, and *CAM_NL2*, and that it has operators like dom and others we use use in the sequel.

As to (*vi*), we would rather simplify it before applying *vS*; using *sP* and then *wP* we refine it to the following program.

$$
\mathit{ill} : [\text{true}, \mathit{ill}' = \mathit{ill} \oplus \{\ a : \operatorname{dom} \mathit{CAM\_NL2} \bullet a \mapsto \mathit{off}\ \}]
$$

As with (*v*), this program can be refined to a call to *updILL* with parameters $\operatorname{dom} \mathit{CAM\_NL2}$ and *off* with the use of *vS* and *pcallI*.

As the constant *CILL* is not being used anymore, we can apply *conR* (constant removal) to eliminate its declaration. This concludes the refinement of *MAINop*.

Another operation of the illumination system that we consider here is $EAop$, which controls the illnmination of the entry areas. This is the operation activated by pressing the DIM1, DIM2, or BRIGHT button of one of the entry areas. In the specification of $EAop$, the input variables $ea?$ and $dim?$ determine, respectively, the chosen entry area and brightness level. The type of $dim?$ is the subset of DIM defined below.

$$DIM_1 == \{dim1, dim2, bright\}$$

If the cockpit door is open and the oil pressure is high, which indicates that there is an engine running, the illumination of the fwd entry area cannot be changed arbitrarily to avoid blinding the cockpit personnel. The table CAM_EAD establishes the maximum brightness to which some of the fwd entry areas lights can be switched in this situation.

$$\begin{array}{|l}
CAM\_EAD : ADDRESS \nrightarrow \{off, dim1, dim2\} \\
\hline
\mathrm{dom}\ CAM\_EAD \subseteq \mathrm{dom}(CAM\_EA \rhd \{fwd\})
\end{array}$$

The operator $\_ <_{dim} \_$ defines an order for the brightness levels according to their intensity.

$$\begin{array}{|l}
\_ <_{dim} \_ : DIM_0 \leftrightarrow DIM_0 \\
\hline
off <_{dim} dim2 \land dim2 <_{dim} dim1 \land dim1 <_{dim} bright \\
\forall a, b, c : DIM_0 \mid a <_{dim} b \land b <_{dim} c \bullet a <_{dim} c
\end{array}$$

The global variables $cockDoor$ and $oilPres$ determine, respectively, whether or not the door is open and the oil pressnre.

$$\begin{array}{|l}
cockDoor : DOOR \\
oilPres : PRESSURE
\end{array}$$

Their types are defined as follows.

$$\begin{array}{lll}
DOOR & ::= & closed \mid open \\
PRESSURE & ::= & low \mid high
\end{array}$$

The behaviour of $EAop$ depends on whether the light indicator associated with the button pressed is on or off. If it is on, the lights are at the brightness level chosen and both they and the light indicator are turned off. If it is off, then it is turned on and the lights are switched to the chosen brightness level. If the chosen entry area is fwd, the cockpit door is open, and the oil pressure is high, then the lights addressed in the table CAM_EAD are switched to the chosen brightness level or to the level indicated in CAM_EAD, whichever is lower. The effect of $EAop$ on the light indicator is specified by $EAINDop$.

$$\begin{array}{|l}
\multicolumn{1}{l}{EAINDop} \\
\hline
\Delta EAINDstate \\
\Xi ZONEINDstate \\
ea? : EA \\
dim? : DIM_1 \\
\hline
eaInd(ea?) = dim? \Rightarrow eaInd' = eaInd \oplus \{ea? \mapsto off\} \\
eaInd(ea?) \neq dim? \Rightarrow eaInd' = eaInd \oplus \{ea? \mapsto dim?\} \\
\hline
\end{array}$$

The schema $EAILLopPassive$ defines the effect of $EAop$ on ill when the light indicator associated with the button pressed is on.

$$
\begin{array}{|l}
\textit{EAILLopPassive} \\
\hline
\Delta ILLstate \\
EAINDstate \\
ea? : EA \\
dim? : DIM_1 \\
\hline
eaInd(ea?) = dim? \\
ill' = ill \oplus \{\ x : \mathrm{dom}(CAM\_EA \rhd \{ea?\}) \bullet x \mapsto off\ \} \\
\hline
\end{array}
$$

The effect of *EAop* on *ill* when the indicator is off is specified by *EAILLopActive*.

$$
\begin{array}{|l}
\textit{EAILLopActive} \\
\hline
\Delta ILLstate \\
EAINDstate \\
ea? : EA \\
dim? : DIM_1 \\
\hline
eaInd(ea?) \neq dim? \\
ill' = ill \oplus \textbf{if } ea? = fwd \wedge cockDoor = open \wedge oilPres = high \\
\qquad\qquad \textbf{then } \{\ x : \mathrm{dom}(CAM\_EA \rhd \{ea?\}) \bullet x \mapsto dim?\ \} \oplus \\
\qquad\qquad\quad \{\ x : \mathrm{dom}\ CAM\_EAD \mid CAM\_EAD\ x <_{dim} dim? \bullet x \mapsto CAM\_EAD\ x\ \} \\
\qquad\qquad \textbf{else } \{\ x : \mathrm{dom}(CAM\_EA \rhd \{ea?\}) \bullet x \mapsto dim?\ \} \\
\hline
\end{array}
$$

The definition of *EAop* is as follows. The precondition of this operation is true.

$$EAop \mathrel{\widehat=} EAINDop \wedge (EAILLopActive \vee EAILLopPassive)$$

The refinement of *EAop* can start with an application of the first formulation of bC. Since this operation does not modify *zoneInd*, however, we can transform it into a shorter program using a third formulation of bC that we present below. Its derivation is presented in Appendix D.

Law bC Basic conversion (operations that do not modify some state components)

$$
\begin{array}{ll}
 & (\Delta S;\ \Xi T;\ di?;\ do! \mid p) \\
\sqsubseteq & bC \\
 & \alpha d_S, \alpha do! : [inv_S \wedge inv_T \wedge \exists\, d'_S;\ do! \bullet (inv'_S \wedge p)[\alpha d_T / \alpha d'_T], (inv'_S \wedge p)[\alpha d_T/\alpha d'_T]\ ] \\
\multicolumn{2}{l}{\textbf{where } S \mathrel{\widehat=} (T;\ d_S \mid inv_S) \textbf{ and } T \mathrel{\widehat=} (d_T \mid inv_T)}
\end{array}
$$

The state is specified by S, which includes T. The operation $(\Delta S;\ \Xi T;\ di?;\ do! \mid p)$ modifies the state, but not the components of T. Therefore, the specification statement generated by bC does not include them in its frame and does not enforce in its postcondition the maintenance of the part of the state invariant defined in T. The predicate $\exists\, d'_S;\ do! \bullet (inv'_S \wedge p)[\alpha d_T/\alpha d'_T]$ is (what is commonly regarded as) the precondition of $(\Delta S;\ \Xi T;\ di?;\ do! \mid p)$.

The application of this formulation of bC to $EAop$ yields the specification statement below.

$$\begin{array}{l} eaInd, \\ ill \end{array} : \left[\begin{array}{l} ZONEINDstate \land ILLstate, \\ \left(\begin{array}{l} ILLstate' \\ eaInd(ea?) = dim? \Rightarrow eaInd' = eaInd \oplus \{ea? \mapsto off\} \\ eaInd(ea?) \neq dim? \Rightarrow eaInd' = eaInd \oplus \{ea? \mapsto dim?\} \\ \left(\begin{array}{l} eaInd(ea?) = dim? \\ ill' = ill \oplus \{\ x : \mathrm{dom}(CAM\_EA \rhd \{ea?\}) \bullet x \mapsto off\ \} \end{array} \right) \lor \\ \quad \left(\begin{array}{l} eaInd(ea?) \neq dim? \\ ill' = ill \oplus \mathbf{if}\ ea? = fwd \land eockDoor = open \land oilPres = high \\ \qquad\qquad \mathbf{then}\ \{\ x : \mathrm{dom}(CAM\_EA \rhd \{ea?\}) \bullet x \mapsto dim?\ \} \oplus \\ \qquad\qquad\quad \{\ x : \mathrm{dom}\, CAM\_EAD \mid CAM\_EAD\ x <_{dim} dim? \bullet \\ \qquad\qquad\qquad\qquad\qquad\qquad x \mapsto CAM\_EAD\ x\ \} \\ \qquad\qquad \mathbf{else}\ \{\ x : \mathrm{dom}(CAM\_EA \rhd \{ea?\}) \bullet x \mapsto dim?\ \} \end{array} \right) \end{array} \right) \end{array} \right]$$

We use the *altI* law to introduce an alternation that determines whether the indicator associated with the button pressed is on or off. After applying *sP* and *wP* to the branches of this alternation, we obtain the following program.

$$\begin{array}{l} \mathbf{if}\ eaInd = dim? \rightarrow \\ \quad \begin{array}{l} eaInd, \\ ill \end{array} : \left[\mathrm{true}, \left(\begin{array}{l} eaInd' = eaInd \oplus \{ea? \mapsto off\} \\ ill' = ill \oplus \{\ x : \mathrm{dom}(CAM\_EA \rhd \{ea?\}) \bullet x \mapsto off\ \} \end{array} \right) \right] \qquad (vii) \\ [\!]\ eaInd \neq dim? \rightarrow \\ \quad \begin{array}{l} eaInd, \\ ill \end{array} : \left[\begin{array}{l} \mathrm{true}, \\ \left(\begin{array}{l} eaInd' = eaInd \oplus \{ea? \mapsto dim?\} \\ ill' = ill \oplus \mathbf{if}\ ea? = fwd \land eockDoor = open \land oilPres = high \\ \qquad\qquad \mathbf{then}\ \{\ x : \mathrm{dom}(CAM\_EA \rhd \{ea?\}) \bullet x \mapsto dim?\ \} \oplus \\ \qquad\qquad\quad \{\ x : \mathrm{dom}\, CAM\_EAD \mid CAM\_EAD\ x <_{dim} dim? \bullet \\ \qquad\qquad\qquad\qquad\qquad\qquad x \mapsto CAM\_EAD\ x\ \} \\ \qquad\qquad \mathbf{else}\ \{\ x : \mathrm{dom}(CAM\_EA \rhd \{ea?\}) \bullet x \mapsto dim?\ \} \end{array} \right) \end{array} \right] \qquad (viii) \\ \mathbf{fi} \end{array}$$

In what follows we refine the specification statement $(viii)$. The refinement of (vii) is similar and simpler.

Applying *fassigI* to $(viii)$ in order to introduce an assignment to *eaInd*, and *cfR* in order to simplify the remaining specification statement, we derive the following program.

$$\begin{array}{l} ill : \left[\begin{array}{l} \mathrm{true}, \\ ill' = ill \oplus \mathbf{if}\ ea? = fwd \land coekDoor = open \land oilPres = high \\ \qquad\qquad \mathbf{then}\ \{\ x : \mathrm{dom}(CAM\_EA \rhd \{ea?\}) \bullet x \mapsto dim?\ \} \oplus \\ \qquad\qquad\quad \{\ x : \mathrm{dom}\, CAM\_EAD \mid CAM\_EAD\ x <_{dim} dim? \bullet x \mapsto CAM\_EAD\ x\ \} \\ \qquad\qquad \mathbf{else}\ \{\ x : \mathrm{dom}(CAM\_EA \rhd \{ea?\}) \bullet x \mapsto dim?\ \} \end{array} \right]; \\ eaInd := eaInd \oplus \{ea? \mapsto dim?\} \end{array}$$

The form of the postcondition of the above specification statement suggests the introduction of an alternation. With this purpose, we apply *altI* to this specification statement and, following

```
if $eaInd = dim? \rightarrow$
        $updILL(\mathrm{dom}(CAM\_EA \rhd \{ea?\}), off)$ ; $eaInd := eaInd \oplus \{ea? \mapsto off\}$
 [] $eaInd \neq dim? \rightarrow$
        if $ea? = fwd \land cockDoor = open \land oilPres = high \rightarrow$
            |[ var $i : 1 \mathinner{..} maxad + 1 \bullet$
                $i := 1$ ;
                do $i \neq maxad + 1 \rightarrow$
                    if $i \in \mathrm{dom}(CAM\_EA \rhd \{ea?\}) \rightarrow$
                        if $i \in \mathrm{dom}\, CAM\_EAD \land CAM\_EAD\ i <_{dim} dim? \rightarrow$
                            $ill := ill \oplus \{i \mapsto CAM\_EAD\ i\}$
                        [] $\neg\,(i \in \mathrm{dom}\, CAM\_EAD \land CAM\_EAD\ i <_{dim} dim?) \rightarrow$
                            $ill := ill \oplus \{i \mapsto dim?\}$
                        fi
                    [] $i \notin \mathrm{dom}(CAM\_EA \rhd \{ea?\}) \rightarrow$ skip
                    fi ;
                    $i := i + 1$
                od
            ]|
        [] $\neg\,(ea? = fwd \land cockDoor = open \land oilPres = high) \rightarrow$
            $updILL(\mathrm{dom}(CAM\_EA \rhd \{ea?\}), dim?)$
        fi
        $eaInd := eaInd \oplus \{ea? \mapsto dim?\}$
fi
```

Figure 4.11: Implementation of *EAop*

the application of *sP* and *wP* to the branches of the resulting alternation, we get to the following program.

$$
\begin{array}{ll}
\textbf{if}\ ea? = fwd \land cockDoor = open \land oilPres = high \rightarrow & \\
\quad ill : \left[\begin{array}{l} \text{true}, \\ ill' = ill \oplus \{\, x : \mathrm{dom}(CAM\_EA \rhd \{ea?\}) \bullet x \mapsto dim?\ \} \oplus \\ \qquad \{\, x : \mathrm{dom}\, CAM\_EAD \mid CAM\_EAD\ x <_{dim} dim? \bullet x \mapsto CAM\_EAD\ x\ \} \end{array}\right] & (ix) \\
[\!]\ \neg\,(ea? = fwd \land cockDoor = open \land oilPres = high) \rightarrow & \\
\quad ill : [\text{true}, ill' = ill \oplus \{\, x : \mathrm{dom}(CAM\_EA \rhd \{ea?\}) \bullet x \mapsto dim?\ \}] & (x) \\
\textbf{fi} &
\end{array}
$$

With an application of *vS* followed by an application of *pcallI*, we can refine (x) to a call to *updILL* with parameters $\mathrm{dom}(CAM\_EA \rhd \{ea?\})$ and *dim?*. As to (ix), since this program does not switch lights to a common brightness level, we would rather implement it without using *updILL*; Figure 4.11 presents an implementation for (xii) that uses an iteration directly. The development of this program poses no difficulties and is not discussed here.

In Figure 4.11 we present the collected code of *EAop*. The other operations of the illumination system either can he refined in much the same way as *MAINop* aud *EAop* or have specifications that are too long to be considered here.

4.4 Conclusions

In this chapter, with the objective of showing that ZRC is a suitable starting point for the study of refinement of Z specifications, we have preseuted three case studies. The examples we have discussed are not exactly realistic. The class manager and the text editor do not beloug to the class of systems that motivate the study of formal methods of software developmeut and, due to space restrictions, we have considered only two of the simpler operations of the cabin-illumination system. Even so, our case studies bring forward a few important points about ZRC.

Since the Z style of structuring specifications is open, the schema calculus can be employed to specify operations in a wide variety of ways. For this reason, it is to be expected that the proposal of additional conversion and even refinement laws become uecessary or appropriate as ZRC is applied in the development of a larger range of systems. The refinement of the text editor, for instance, has prompted the introduction of a formulation of *sconjC* (schema conjunction conversion) that had not been included in the original set of conversion laws. Also, in the development of the cabin-illuminatiou system, the proposal of an additional formulation of *bC* (basic couversion) has been proved to be useful.

In this respect, what distinguishes ZRC from other methods of refining Z specifications is its formalisation. Based on this work, the soundness of any new conversiou or refinement law that becomes necessary can be established and the risk of mistakes minimised. The mauy examples of law derivations provided in Appendix D can be of assistance in this kind of effort. Altogether, ZRC is not only a collection of laws that can be applied to refine Z specifications, but also a theory of refinement for Z.

The use of the Z dashing convention, as opposed to the 0-subscript conventiou adopted in Morgan's calculus to represent initial variables, may have caused some coucern as to the complexity of the refinement laws. Indeed, if compared to corresponding laws of Morgau's calculus, some of the ZRC refinement laws have a slightly more complex formulation which involves additional substitutions to remove or introduce dashes. By now, however, it should be clear that this does not lead to more complex proof-obligations and that refinements in ZRC can be carried out in much the same way as they can in Morgan's calculus. Furthermore, we believe that, at least for those used to the Z style, the formulations of the ZRC laws are not obstructive.

As with Morgan's calculus, the application of ZRC may involve long programs and proof-obligations. Even though the conversion laws of ZRC aim at exploiting the structure of the schema definitions and avoiding unnecessary expansions, lengthy schemas, specificatiou statements, and proof-obligations may eventually come about or be part of the initial specificatiou. The text editor case study, for instance, involves quite a few long programs and, as we have said, this is not even a realistic example. Specifications of systems whose development requires the use of formal methods are usually much more complex and lengthy. The effective application of ZRC to refine these systems requires the assistance of a tool. Without this support, since the activities involved in the refinement process are extremely error-prone, the reliability of the results obtained is compromised.

In the next chapter, we conclude our presentation of ZRC by considering related works and possible lines of future research. There we compare ZRC to the techniques employed by King and Neilson to develop implementations for the class manager and the text editor.

Chapter 5

Conclusions

At present, if the use of a formal method covering all phases of software development, possibly with the support of a tool, is required, then Z is not a feasible or a straightforward answer. This is one of the major criticisms that have been levelled at Z, which, nevertheless, is a highly successful specification language. In this context, ZRC comes as a modest but promising step forward in the direction of further encouraging the application in practice of Z and, more generally, of formal methods.

As a refinement calculus, ZRC integrates a successful specification language to a most promising method of developing programs. The refinement calculus builds upon results of years of research on (formal) program development. As with Back's [1, 4] and Morris's [48, 50] work, Morgan's calculus formalises the stepwise refinement technique of program development, but goes further and proposes an innovative style of presenting developments and calculating programs based on an extensive set of refinement laws.

The possibility of calculating, as opposed to verifying, programs accounts for developments that can be uniformly presented as sequences of simple refinement steps. Each step can be justified by the application of a refinement law and, possibly, the discharge of corresponding proof-obligations. Moreover, refinement laws provide guidance on the construction of programs.

Although there seems to be no report of applications of the refinement calculus in industry or of case studies of substantial size, we are convinced that external factors are responsible for this situation. The refinement calculus is still in its relatively early days: it was only in 1990, when the first edition of Morgan's book [44] went into press, that the refinement calculus was put together and more widely publicised.

Moreover, the application of the refinement calculus involves heavy formula manipulations and the proof of long theorems, and so is practically infeasible without the support of a tool when larger examples are considered; apparently, at the moment, no reliable and effective tool that supports the application of the refinement calculus on this scale is available. Also, the benefits of applying the refinement calculus in a rigorous way, leaving proof-obligations unproven or providing only informal arguments to discharge them, do not seem to have been emphasised.

The specification facilities of the refinement calculus are also a cause of concern because the lack of a structuring mechanism like the schema calculus can be a difficulty in the treatment of more complex examples. With ZRC, this last problem is solved. Nevertheless, it must be said that a lot of effort is still required before the use of a refinement calculus becomes widespread.

The integration of Z to a refinement calculus was first proposed in [34] by King, and, in [64, 65] and [66], Woodcock and Wordsworth also follow this approach. To the best of our knowledge, however, ZRC is unique in that it is completely justified in terms of a well-established mathematical model of program development: weakest preconditions. Moreover, ZRC adopts the conventions of Z, avoiding a change of style during the development process, and includes support for the development of, possibly recursive and parametrised, procedures and a calculational technique of data refinement.

In summary, ZRC is a comprehensive technique of program development which can be used to calculate programs from Z specifications in a smooth way, and which is firmly based on mathematical principles. Its design has taken advantage of existing resnlts on refinement of Z specifications and, furthermore, its formalisation makes it extensible. In view of that, we believe ZRC to be a source of encouragement for further study on refinement of Z specifications; the application of ZRC in the development of complex realistic systems in a rigorous way or with the support of a tool can teach us many lessons. In tbe next section, we discuss related works and in Section 5.2 we propose a few lines for fnture research.

5.1 Related Work

Most conversion laws of ZRC are based on those proposed by King in [34]. There are, however, some fundamental differences between ZRC and the techniqne proposed in [34] to refine Z specifications. In general terms, King's work is not a refinement calcnlus for Z, as ZRC is, but a method for integrating Z with Morgan's calculus. As such, King's technique provides the same specification and design resources of Z and the refinement calculus, but, on the other hand, its application requires a change of notation and style dnring the development of a program.

When refining a schema using King's technique, we first translate it to a program of the refinement calculus. In this process, the decoration conventions of Z are forgone, the 0-subscript convention for initial variables of Morgan's calculus are adopted, and the names of the state components and of the input and output variables are shortened. For those familiar with both Z and the refinement calculus, this change of notation may not be a major hindrance, but ZRC is a proof that it is not necessary. Moreover, we believe that, for Z users, the notation employed by ZRC is both natural and elegant.

In [34] Z specifications are translated to modules; the structure employed is that presented in [45]. The clause **var** is used to declare the state components, the **and** clause, to introduce the state invariant, and the operations are declared as procedures. In contrast, as we point out in the next section, ZRC is concerned only with the translation of individual operations. We further discuss the issues of modules and invariants in Section 5.2.

The techniqne proposed by Wordsworth in [66] to refine Z specifications is, as ZRC, tailored to the Z notation and style. In tbis work, Wordsworth defines a refinement relation between schemas using the relational view of operations (instead of weakest preconditions). Assignment is defined as a schema, so that refinement of a schema by an assignment can be proved using the definition of refinement between schemas. Other programming constructions (alternation, sequential composition, variable blocks, and iteration) can be introduced using refinement rules; some of them correspond to conversion rules of [34] and some of them correspond to laws of

Morgan's calculus. As in ZRC, schemas are regarded as commands. In [52], Potter, Sinclair, and Till provide an alternative presentation of Wordsworth's technique.

Specification statements are not part of the language considered by Wordsworth. Even the refinement rules of [66] that correspond to laws of Morgan's calculus apply to schemas. As a consequence, they give rise to more complex proof-obligations and provide little gnidance to the development; they are better suited to the verification, rather than to the calculation, of programs.

Wordsworth proposes the use of schemas as procedures. More precisely, he presents an example where a parametrised call to a procedure (schema) with an input and an output is equivalently defined as a schema and can, therefore, be used as a command; refinement can proceed as usual since procedure calls are schemas. As a matter of fact, Wordsworth does not present his approach to procedures and parameters in details, but it is clear that it is not as general as Back's approach, in which procedures may be defined by any form of program and not only schemas.

As mentioned in Chapter 4, in [51] Neilson develops a C implementation for a text editor based on its Z specification; there he also introduces the technique of development that he employs. Besides considering the development of programs from concrete Z specifications, Neilson proposes a technique for data refining Z specifications different from that in [58, 52, 16, 65]. Since ZRC is not concerned with this stage of the refinement of a Z specification, we do not discuss this part of Neilson's work here.

There are many similarities between Neilson's and Wordsworth's technique for (algorithmically) refining schemas. Neilson defines refinement between schemas in the same way as Wordsworth and proposes the same refinement rnles to introduce programming constructs (except for that concerning assignment). As opposed to Wordsworth, however, Neilson does not present refinement rules corresponding to King's conversion rules. Instead, Neilson proves a number of properties of the refinement relation and presents several refinement rnles that are used in the development of the text editor. In order to justify his refinement rules, Neilson defines the programming constructs as schemas.

Another approach to the refinement of Z specifications is suggested in [63]. There Ward introduces in the language of Morgan's calculus generalisations of the Z conjunction and disjunction schema operators so that specification statements can be combined and the Z incremental style of building specifications can be used. The aim is to achieve a refinement calculus that can cope with larger specifications itself.

Ward, however, does not consider the other Z schema operators, which also contribute to the success of the Z style, and it is not clear how they can be added to the refinement calculus. Moreover, the conjunction and disjunction operators that he defines are not monotonic with respect to the refinement relation. The techniqne that Ward suggests for refining programs built as conjunctions or disjunctions consists of using either the weakest precondition definitions directly or refinement laws similar to the rules presented in [34] for translating schema expressions.

In [16] Diller proposes a method of program verification for Z. As King integrates Z with Morgan's calculus, Diller integrates Z with a Floyd-Hoare logic. In developing a program to implement an operation specified by a Z schema using Diller's technique, we first transform the schema into a Hoare triple and then proceed to write and verify the program as usual in methods based on Floyd-Hoare logics. The conversion procedure presented by Diller transforms a schema that specifies an operation into a Hoare triple whose pre and postcondition are determined by the schema (and program variables) in consideration and whose program component is to be guessed.

This work does not take advantage of the structure of schema expressions.

As we have briefly mentioned in Chapter 3, in [32] Josephs defines *wp* as a schema operator. For a schema *Op* that specifies an operation over a state defined by a schema *S*, and for a schema *R* that specifies a postcondition, $wp_{S'}(Op, R)$ is defined in [32] as a schema that specifies the weakest precondition that guarantees that *Op* terminates in a state that satisfies *R*. In spite of this, our characterisation of *wp* in terms of Z predicates (Theorem 2.6 in Chapter 2) is similar to that in [32]. Operations with input and output, however, are not treated in [32].

The *wp* schema operator is used in [32] to define schemas and schema operators that represent programs and program constructors of Dijkstra's langnage of guarded commands, and to define a refinement relation between schemas. Based on these definitions, Josephs proposes a few refinement rules. These rules mention the *wp* operator and, in summary, Josephs's method is a refinement-*wp* calcnlus for Z. As in [66], specification statements are not considered in Josephs's work; the comments made above about Wordsworth's refinement rules are also valid for Josephs's rules: they are in general difficult to apply and more appropriate for program verification instead of calculation.

5.2 Future Research

Throughout this work, we have assumed that the components of a schema that specifies an operation are the before and after-state, input, and output variables. Initialisation operations, however, characterise states, and not state transitions. Therefore, their components are just the after-state, input, and output variables. As a consequence, ZRC cannot be used to refine initialisation operations. This is not a major problem because in general initialisation operations are very simple so that their implementation does not require the use of a refinement calculus. Nevertheless, it should not be too difficult to extend ZRC (and its formalisation) to deal with initialisation operations. Schemas with no state components and initial variables, like that named *Success* which has been defined in the specification of the class manager (see Section 4.1), and which are often used in the Oxford style of error treatment, are yet to be considered as well.

The formalisation of ZRC involves several proofs of theorems, lemmas, and corollaries, and, in particular, many law derivations, all of which have been carefully checked. Since the activities involved in the elaboration and presentation of proofs are admittedly very error-prone, however, by using a theorem prover to check the formalisation of ZRC, we can improve its reliability and, consequently, that of the ZRC laws. A similar work is presented in [7], where Back and Wright describe how the HOL proof assistant system can be used to formalise a refinement technique largely based on Back's work.

The variable blocks with invariants (and invariant hlocks) of Morgan's calculus are not part of ZRC-L. Their treatment incurs in considerable modifications to the *wp* semantics of ZRC-L, to the definition of refinement, and, consequently, to the whole formalisation of ZRC. The definition of an invariant in a variable block has influence on the behaviour of the program in its body: it may assume and must preserve the invariant. In [43], Morgan defines weakest preconditions in relation to an invariant, which is an additional parameter of *wp*. The task of establishing a correspondence between the weakest precondition of a schema relative to an invariant and its relational semantics may not be trivial. Moreover, as far as we know, the formalisation of invariants has not yet been considered in conjunction with procedures and data refinement. Invariants can be of help in rigorous developments, where their elimination is either ignored or justified informally. Their usefulness in completely formal developments, however, can be discussed.

As already remarked, King implements a Z specification using a module written using the language of Morgan's calculus. In [52], Potter, Sinclair, and Till show how to implement a driver program which controls the execution of the operations of a Z specification. In contrast, ZRC concentrates only on the refinement of individual operations. The ultimate implementation of a Z specification does involve the embedding of its operations, or rather, of their implementations, in a program that provides an interface for them. The development of these programs, however, may not be trivial and involve complex questions of modularisation. We do not believe that this issue can be addressed lightly in a general context.

The schema calculus is largely responsible for the success of Z, as it encourages and supports the development of structured specifications. Many have argued that the schema calculus is not enough and have proposed modular extensions to Z [38, 9, 37, 59, 36]. Whether or not the structure of a specification should be used in its implementation, however, is still an open question. Works on this area include [12, 53, 10]. Our hope is that, as the issue of modularisation seems to be fairly independent from that of implementing individual operations, ZRC can be integrated without many difficulties with design methods concerned with the architectural aspects of programming.

The fact that the use of a refinement calculus in practice requires the support of a tool is widely recognised. The works in [62, 21, 7, 68, 67, 22] describe different tools that have been developed to support the application of Morgan's calculus and other refinement techniques. Before ZRC can be seriously considered in practice, a tool that supports its application has to be made available.

The consolidation of ZRC also depends on the development of more case studies. Our ambition is that, by establishing a solid foundation for the refinement of Z specifications, ZRC and its formalisation become an additional motivation for further investigations in this field. Much work is yet to be done on strategies for refining Z specifications.

Appendix A

Mathematical Notation

In Chapter 2 we have used the semantic metalanguage introduced in [8]. In this appendix, which is partially extracted from [8] itself, we summarise the less familiar symbols of this language that we have actually employed.

$\ni$ Choice relation: associates a set with each of its elements. It is the inverse of the element relation.

$$x \ni y \Leftrightarrow y \in x$$

$\{\ldots\}$ Set extension function: takes a tuple of values as argument and yields the set containing them.

$$\{\ldots\}.(x_1, \ldots, x_n) = \{x_1, \ldots, x_n\}$$

$\_^\circ$ Constant function constructor

$$x^\circ.y = x$$

$\sqcup$ Compatible union: this function forms the union of compatible fnnctions or, in other words, functions whose uniou is still a function.

$$f \sqcup g = f \cup g \text{ provided } \forall x : \mathrm{dom} f \cap \mathrm{dom}\, g \bullet f.x = g.x$$

Relational Constructors

$\langle R_1, \ldots, R_n \rangle$ Tupling construction

$$x\ \langle R_1, \ldots, R_n \rangle\ (y_1, \ldots, y_n)\ \Leftrightarrow\ x\ R_1\ y_1\ \wedge \ldots \wedge\ x\ R_n\ y_n$$

$(R_1 \times \ldots \times R_n)$ Product of relations

$$(x_1, \ldots, x_n)\ (R_1 \times \ldots \times R_n)\ (y_1, \ldots, y_n)\ \Leftrightarrow\ x_1\ R_1\ y_1\ \wedge \ldots \wedge\ x_n\ R_n\ y_n$$

R^{-1} Relatioual inverse

$$x\ R^{-1}\ y\ \Leftrightarrow\ y\ R\ x$$

Cartesian products

A^n Enumerated product: set of tuples $(x_1, \ldots, x_n)$ such that $x_1, \ldots, x_n \in A$.

A^+ Generalised product

$$A^+ = \bigcup_{i>0} A^i$$

Appendix B

Proofs of Some Theorems

In this appendix we present the proof of some of the theorems that have been proposed in Chapter 2, but first we introduce a lemma.

Lemma B.1 *For every schema $(d;\ d';\ di?;\ do!\mid p)$, and postcondition ψ,*

$$\begin{aligned}&(\forall\, d';\ do! \bullet p \Rightarrow \psi) \equiv\\ &\quad wp.(d;\ d';\ di?;\ do!\mid p).\mathit{true} \Rightarrow wp.(d;\ d';\ di?;\ do!\mid p).\psi\end{aligned}$$

Proof

$$\begin{aligned}
&\forall\, d';\ do! \bullet p \Rightarrow \psi\\
&\equiv \forall\, d';\ do! \bullet ((\exists\, d';\ do! \bullet p) \wedge (p \Rightarrow \psi)) \vee (\neg\,(\exists\, d';\ do! \bullet p) \wedge (p \Rightarrow \psi))\\
&&&\text{[by predicate calculus]}\\
&\equiv \forall\, d';\ do! \bullet ((\exists\, d';\ do! \bullet p) \wedge (p \Rightarrow \psi)) \vee (\neg\,(\exists\, d';\ do! \bullet p) \wedge \neg\, p \wedge (p \Rightarrow \psi))\\
&&&\text{[by predicate calculus]}\\
&\equiv \forall\, d';\ do! \bullet ((\exists\, d';\ do! \bullet p) \wedge (p \Rightarrow \psi)) \vee \neg\,(\exists\, d';\ do! \bullet p) &&\text{[by predicate calculus]}\\
&\equiv ((\exists\, d';\ do! \bullet p) \wedge (\forall\, d';\ do! \bullet p \Rightarrow \psi)) \vee \neg\,(\exists\, d';\ do! \bullet p) &&\text{[by predicate calculus]}\\
&\equiv wp.(d;\ d';\ di?;\ do!\mid p).\text{true} \Rightarrow wp.(d;\ d';\ di?;\ do!\mid p).\psi &&\text{[by definition of of } wp\text{]}
\end{aligned}$$

□

This lemma is used in the proof of the next two theorems.

Theorem 2.8 *For all schemas Op_1 and Op_2 that specify operations over the same state and with the same inputs and outputs, and for every postcondition ψ,*

$$\begin{aligned}&wp.(Op_1 \vee Op_2).\psi \equiv\\ &\quad (wp.Op_1.\text{true} \vee wp.Op_2.\text{true}) \wedge\\ &\quad (wp.Op_1.\text{true} \Rightarrow wp.Op_1.\psi) \wedge (wp.Op_2.\text{true} \Rightarrow wp.Op_2.\psi)\end{aligned}$$

Proof Without loss of generality, we assume that Op_1 and Op_2 can be written in the form $\langle d;\ d';\ di?;\ do! \mid p_1\rangle$ and $\langle d;\ d';\ di?;\ do! \mid p_2\rangle$, respectively.

$$
\begin{aligned}
& wp.(Op_1 \lor Op_2).\psi \\
\equiv\ & wp.\langle d;\ d';\ di?;\ do! \mid p_1 \lor p_2\rangle.\psi && \text{[by a property of schema disjunction]} \\
\equiv\ & (\exists\, d';\ do! \bullet p_1 \lor p_2) \land (\forall\, d';\ do! \bullet p_1 \lor p_2 \Rightarrow \psi) && \text{[by definition of } wp\text{]} \\
\equiv\ & ((\exists\, d';\ do! \bullet p_1) \lor (\exists\, d';\ do! \bullet p_2)) \land (\forall\, d';\ do! \bullet p_1 \lor p_2 \Rightarrow \psi) && \text{[by predicate calculus]} \\
\equiv\ & (wp.Op_1.\text{true} \lor wp.Op_2.\text{true}) \land (\forall\, d';\ do! \bullet p_1 \lor p_2 \Rightarrow \psi) && \text{[by definition of } wp\text{]} \\
\equiv\ & (wp.Op_1.\text{true} \lor wp.Op_2.\text{true}) \land (\forall\, d';\ do! \bullet p_1 \Rightarrow \psi) \land (\forall\, d';\ do! \bullet p_2 \Rightarrow \psi) \\
& && \text{[by predicate calculus]} \\
\equiv\ & (wp.Op_1.\text{true} \lor wp.Op_2.\text{true}) \land && \text{[by Lemma B.1]} \\
& \quad (wp.Op_1.\text{true} \Rightarrow wp.Op_1.\psi) \land (wp.Op_2.\text{true} \Rightarrow wp.Op_2.\psi)
\end{aligned}
$$

□

Theorem 2.11 *For every schema Op that specifies an operation, all declarations d, d', $di?$, and $do!$ that introduce components of Op, and every postcondition ψ,*

$$
\begin{aligned}
& wp.(\exists\, d;\ d';\ di?;\ do! \bullet Op).\psi \equiv \\
& \qquad (\exists\, d;\ di? \bullet wp.Op.\text{true}) \land (\forall\, d;\ di? \bullet wp.Op.\text{true} \Rightarrow wp.Op.\psi)
\end{aligned}
$$

provided the variables of αd, $\alpha d'$, $\alpha di?$, and $\alpha do!$ do not occur free in ψ.

Proof We consider an existential quantification $\exists\, d_1;\ d_1';\ di_1?;\ do_1! \bullet Op$, where Op is the schema $\langle d_1;\ d_2;\ d_1';\ d_2';\ di_1?;\ di_2?;\ do_1!;\ do_2! \mid p\rangle$, and $\alpha d_1 \cap \alpha d_2 = \varnothing$, $\alpha d_1' \cap \alpha d_2' = \varnothing$, $\alpha di_1? \cap \alpha di_2? = \varnothing$, and $\alpha do_1! \cap \alpha do_2! = \varnothing$.

$$
\begin{aligned}
& wp.(\exists\, d_1;\ d_1';\ di_1?;\ do_1! \bullet Op).\psi \\
\equiv\ & wp.\langle d_2;\ d_2';\ di_2?;\ do_2! \mid \exists\, d_1;\ d_1';\ di_1?;\ do_1! \bullet p\rangle.\psi \\
& && \text{[by a property of schema existential quantification]} \\
\equiv\ & (\exists\, d_2';\ do_2! \bullet \exists\, d_1;\ d_1';\ di_1?;\ do_1! \bullet p) \land (\forall\, d_2';\ do_2! \bullet (\exists\, d_1;\ d_1';\ di_1?;\ do_1! \bullet p) \Rightarrow \psi) \\
& && \text{[by definition of } wp\text{]} \\
\equiv\ & (\exists\, d_2';\ do_2! \bullet \exists\, d_1;\ d_1';\ di_1?;\ do_1! \bullet p) \land (\forall\, d_2';\ do_2! \bullet \forall\, d_1;\ d_1';\ di_1?;\ do_1! \bullet p \Rightarrow \psi) \\
& && \text{[by } \alpha d_1,\ \alpha d_1',\ \alpha di_1?, \text{ and } \alpha do_1! \text{ are not free in } \psi\text{]} \\
\equiv\ & (\exists\, d_1;\ di_1? \bullet \exists\, d_1';\ d_2';\ do_1!;\ do_2! \bullet p) \land (\forall\, d_1;\ di_1? \bullet \forall\, d_1';\ d_2';\ do_1!;\ do_2! \bullet p \Rightarrow \psi) \\
& && \text{[by predicate calculus]} \\
\equiv\ & (\exists\, d_1;\ di_1? \bullet wp.Op.\text{true}) \land (\forall\, d_1;\ di_1? \bullet wp.Op.\text{true} \Rightarrow wp.Op.\psi) \\
& && \text{[by definition of } wp \text{ and Lemma B.1]}
\end{aligned}
$$

□

Theorem 2.14 *For every schema Op that specifies an operation, all lists of variables os, $oi?$, $oo!$, ns, $ni?$, and $no!$ without duplicates, and every postcondition ψ where the variables of os, os', $oi?$, and $oo!$ do not occur free,*

$$\begin{aligned}&wp.Op[ns, ns', ni?, no!/os, os', oi?, oo!].\psi \equiv\\ &\quad (wp.Op.\psi[os, os', oi?, oo!/ns, ns', ni?, no!])[ns, ni?/os, oi?]\end{aligned}$$

We assume that the variables of ns, ns', $ni?$, and $no!$ are not components of Op; and that the variables of os, $oi?$, ns, ns', $ni?$, and $no!$ do not occur as global variables in Op.

Proof We consider a schema Op of the form $\langle d;\ dos;\ d';\ dos';\ di?;\ doi?;\ do!;\ doo! \mid p\rangle$, where dos, dos', $doi?$, and $doo!$ declare the variables of os, os', $oi?$, and $oo!$, respectively. We assume that d, d', $di?$, and $do!$ declare the components not affected by the renaming. We also assume that the variables of ns, ns', $ni?$, and $no!$ are declared by dns, dns', $dni?$, and $dno!$.

$$\begin{aligned}
&wp.(Op[ns, ns', ni?, no!/os, os', oi?, oo!]).\psi\\
&\equiv (\exists\, d';\ dns';\ do!;\ dno! \bullet p[ns, ns', ni?, no!/os, os', oi?, oo!]) \wedge\\
&\quad (\forall\, d';\ dns';\ do!;\ dno! \bullet p[ns, ns', ni?, no!/os, os', oi?, oo!] \Rightarrow \psi)\\
&\hfill [\text{by a property of renaming and the definition of } wp]\\
&\equiv (\exists\, d';\ dos';\ do!;\ doo! \bullet p)[ns, ni?/os, oi?] \wedge\\
&\quad (\forall\, d';\ dns';\ do!;\ dno! \bullet p[ns, ns', ni?, no!/os, os', oi?, oo!] \Rightarrow \psi)\\
&\quad [\text{by } ns' \text{ and } no! \text{ are not free in } p\text{, and } os \text{ and } oi? \text{ are not free in } d', dos', do!, \text{ and } doo!]\\
&\equiv (\exists\, d';\ dos';\ do!;\ doo! \bullet p)[ns, ni?/os, oi?] \wedge \qquad [\text{by } os, os', oi?, \text{ and } oo! \text{ are not free in } \psi]\\
&\quad (\forall\, d';\ dns';\ do!;\ dno! \bullet p[ns, ni?/os, oi?][ns', no!/os', oo!] \Rightarrow\\
&\qquad \psi[os, os', oi?, oo!/ns, ns', ni?, no!][ns, ni?/os, oi?][ns', no!/os', oo!])\\
&\equiv (\exists\, d';\ dos';\ do!;\ doo! \bullet p)[ns, ni?/os, oi?] \wedge\\
&\quad (\forall\, d';\ dos';\ do!;\ doo! \bullet p \Rightarrow \psi[os, os', oi?, oo!/ns, ns', ni?, no!])[ns, ni?/os, oi?]\\
&\quad [\text{by } ns' \text{ and } no! \text{ are not free in } p\text{, and } os \text{ and } oi? \text{ are not free in } d', dos', do!, \text{ and } doo!]\\
&\equiv ((\exists\, d';\ dos';\ do!;\ doo! \bullet p) \wedge \qquad [\text{by a property of substitution}]\\
&\quad (\forall\, d';\ dos';\ do!;\ doo! \bullet p \Rightarrow \psi[os, os', oi?, oo!/ns, ns', ni?, no!]))[ns, ni?/os, oi?]\\
&\equiv (wp.Op.\psi[os, os', oi?, oo!/ns, ns', ni?, no!])[ns, ni?/os, oi?] \qquad [\text{by definition of } wp]
\end{aligned}$$

□

Theorem 2.15 *For every generic schema designator $Op[e_1, e_2, \ldots, e_n]$, where Op is a generic schema that specifies an operation and has parameters $x_1, x_2, \ldots, x_n$; and for every postcondition ψ where $x_1, x_2, \ldots, x_n$ do not occur free,*

$$wp.Op[e_1, e_2, \ldots, e_n].\psi \equiv (wp.Op.\psi)[e_1, e_2, \ldots, e_n/x_1, x_2, \ldots, x_n]$$

provided the components of Op are not free in $e_1, e_2, \ldots, e_n$.

Proof As Op is a generic schema that specifies an operation and has parameters $x_1, x_2, \ldots, x_n$, we can assume that it can be written in the form $\langle d;\ d';\ di?;\ do! \mid p\rangle[x_1, x_2, \ldots, x_n]$. In this case, the generic schema designator $Op[e_1, e_2, \ldots, e_n]$ can be expanded to the schema below, since the variables of αd, $\alpha d'$, $\alpha di?$, and $\alpha do!$ are not free in $e_1, e_2, \ldots, e_n$.

$$\begin{array}{l}
\langle d[e_1, e_2, \ldots, e_n/x_1, x_2, \ldots, x_n];\ d'[e_1, e_2, \ldots, e_n/x_1, x_2, \ldots, x_n];\\
\ di?[e_1, e_2, \ldots, e_n/x_1, x_2, \ldots, x_n];\ do![e_1, e_2, \ldots, e_n/x_1, x_2, \ldots, x_n] \mid\\
\quad p[e_1, e_2, \ldots, e_n/x_1, x_2, \ldots, x_n]\rangle
\end{array}$$

Theorem 2.6 applies to this schema and hence we can make the following deduction.

$$\begin{array}{ll}
wp.(Op[e_1, e_2, \ldots, e_n]).\psi & \\
\equiv (\exists\, d'[e_1, e_2, \ldots, e_n/x_1, x_2, \ldots, x_n];\ do![e_1, e_2, \ldots, e_n/x_1, x_2, \ldots, x_n] \bullet & \\
\qquad p[e_1, e_2, \ldots, e_n/x_1, x_2, \ldots, x_n]) \wedge & \\
\quad (\forall\, d'[e_1, e_2, \ldots, e_n/x_1, x_2, \ldots, x_n];\ do![e_1, e_2, \ldots, e_n/x_1, x_2, \ldots, x_n] \bullet & \\
\qquad p[e_1, e_2, \ldots, e_n/x_1, x_2, \ldots, x_n] \Rightarrow \psi) & [\text{by definition of } wp] \\
\equiv ((\exists\, d';\ do! \bullet p) \wedge (\forall\, d';\ do! \bullet p \Rightarrow \psi))[e_1, e_2, \ldots, e_n/x_1, x_2, \ldots, x_n] & \\
\qquad [\text{by } x_1, x_2, \ldots, x_n \text{ are not free in } \psi, \text{ and } \alpha d' \text{ and } \alpha do! \text{ are not free in } e_1, e_2, \ldots, e_n] & \\
\equiv (wp.Op.\psi)[e_1, e_2, \ldots, e_n/x_1, x_2, \ldots, x_n] & [\text{by definition of } wp]
\end{array}$$

□

Appendix C

Weakest Precondition Definitions

In this appendix we provide a weakest precondition semantics for ZRC-L. Most of the definitions that we present have already been introduced and discussed in Chapters 2 and 3; this appendix is a summary.

For every program p and postcondition ψ which possibly contains free program variables, we have the definition below, where vl is the list of all program variables and cl is a list of fresh constants, none of which is free in p or ψ.

$$wp.p.\psi = (wp.p.\psi[cl/vl])[vl/cl]$$

For postconditions ψ that do not contain free occurrences of program variables, we define $wp.p.\psi$ by recursion over the structure of p as follows.

1. $wp.\langle d;\ d';\ di?;\ do! \mid p\rangle.\psi \equiv (\exists\, d';\ do! \bullet p) \wedge (\forall\, d';\ do! \bullet p \Rightarrow \psi)$
2. $wp.w : [pre, post].\psi \equiv pre \wedge (\forall\, dw' \bullet post \Rightarrow \psi)[\_/']$
   provided dw declares the variables of w.
3. $wp.\mathbf{skip}.\psi' \equiv \psi$
4. $wp.\{pre\}.\psi' \equiv pre \wedge \psi$
5. $wp.[post].\psi' \equiv post[\_/'] \Rightarrow \psi$
6. $wp.vl := el.\psi' \equiv \psi[el/vl]$
7. $wp.(p_1\ ;\ p_2).\psi \equiv wp.p_1.(wp.p_2.\psi)'$
8. $wp.\mathbf{if}\ [\!]\ i \bullet g_i \rightarrow p_i\ \mathbf{fi}.\psi \equiv (\bigvee i \bullet g_i) \wedge (\bigwedge i \bullet g_i \Rightarrow wp.p_i.\psi)$
9. $wp.\ |\![\ \mathbf{var}\ dvl \bullet p\ ]\!|\ .\psi \equiv \forall\, dl' \bullet wp.p[l, l'/vl, vl'].\psi$
   provided
   - dvl and dl declare the variables of vl and l, respectively, and differ just in the names of the variables that they declare;
   - The names of l and l' are not free in p and ψ.

10. $wp.\lbrack\!\lbrack\, \mathbf{con}\ dcl \bullet p \,\rbrack\!\rbrack\, .\psi \equiv \exists\, dl \bullet wp.p[l/cl].\psi$

    provided

    - dcl and dl declare the constants of cl and l, respectively, and differ just in the names of the variables that they declare;
    - The names of l and l' are not free in p and ψ .

11. $(\mathbf{val}\ dvl \bullet p)(el) = \lbrack\!\lbrack\, \mathbf{var}\ dl \bullet l := el\ ;\ \ p[l, l'/vl, vl']\ \rbrack\!\rbrack$

    provided

    - dvl and dl declare the variables of vl and l, respectively, and differ just in the names of the variables that they declare;
    - The names of l and l' are not free in p and el.

12. $(\mathbf{res}\ dvl_1 \bullet p)(vl_2) = \lbrack\!\lbrack\, \mathbf{var}\ dl \bullet p[l, l'/vl_1, vl_1']\ ;\ \ vl_2 := l \,\rbrack\!\rbrack$

    provided

    - dvl_1 and dl declare the variables of vl_1 and l, respectively, and differ just in the names of the variables that they declare;
    - The names of l and l' are not free in p, and are not in vl_2.

13. $(\mathbf{val\text{-}res}\ dvl_1 \bullet p)(vl_2) = \lbrack\!\lbrack\, \mathbf{var}\ dl \bullet l := vl_2\ ;\ \ p[l, l'/vl_1, vl_1']\ ;\ \ vl_2 := l \,\rbrack\!\rbrack$

    provided

    - dvl_1 and dl declare the variables of vl_1 and l, respectively, and differ just in the names of the variables that they declare;
    - The names of l and l' are not free in p, and are not in vl_2.

14. $(\mathbf{res}\ v : t \bullet p)(f\ x) = \lbrack\!\lbrack\, \mathbf{var}\ u : t \bullet (\mathbf{res}\ v : t \bullet p)(u)\ ;\ f := f \oplus \{x \mapsto u\}\ \rbrack\!\rbrack$

    provided u and u' are not free in p.

15. $(\mathbf{val\text{-}res}\ v : t \bullet p)(f\ x) = \lbrack\!\lbrack\, \mathbf{var}\ u : t \bullet u := f\ x\ ;\ (\mathbf{val\text{-}res}\ v : t \bullet p)(u)\ ;\ f := f \oplus \{x \mapsto u\}\ \rbrack\!\rbrack$

    provided u and u' are not free in p.

16. $(\mathbf{par}\ dvl_1;\ fpd \bullet p)(el_1, el_2) = (\mathbf{par}\ dvl_1 \bullet (fpd \bullet p)(el_2))(el_1)$

    provided the variables declared by dvl_1 are not free in el_2.

17. $\lbrack\!\lbrack\, \mathbf{proc}\ pn \mathrel{\widehat{=}} (fpd \bullet p_1)(pn) \bullet p_2(pn) \,\rbrack\!\rbrack = p_2(\mu(fpd \bullet p_1))$

18. $\mathbf{do}\ [\!]\ i \bullet g_i \rightarrow p_i\ \mathbf{od} = \lbrack\!\lbrack\, \mathbf{proc}\ it \mathrel{\widehat{=}} \mathbf{if}\ [\!]\ i \bullet g_i \rightarrow p_i\ ;\ \ it\ [\!]\ \neg(\bigvee i \bullet g_i) \rightarrow \mathbf{skip}\ \mathbf{fi} \bullet it \,\rbrack\!\rbrack$

    provided it is not free in g_i and p_i.

19. $\lbrack\!\lbrack\, \mathbf{proc}\ pn \mathrel{\widehat{=}} (fpd \bullet p_1)(pn)\ \mathbf{variant}\ n\ \mathbf{is}\ e \bullet p_2(pn) \,\rbrack\!\rbrack = p_2(\mu(fpd \bullet \lbrack\!\lbrack\, \mathbf{con}\ n : \mathbb{Z} \bullet p_1 \,\rbrack\!\rbrack))$

Appendix D

Laws of ZRC and Their Derivations

This appendix enumerates the conversion and refinement laws of ZRC, together with their derivations. The data refinement laws come at the end.

D.1 Conversion Laws

Some conversion laws of ZRC apply to schemas of the form $\langle \Delta S;\ di?;\ do! \mid p\rangle$. The lemma below characterises their weakest precondition.

Lemma D.1 *For every postcondition* ψ,

$$wp.\langle \Delta S;\ di?;\ do! \mid p\rangle.\psi \equiv inv \wedge (\exists\, d';\ do! \bullet inv' \wedge p) \wedge (\forall\, d';\ do! \bullet inv' \wedge p \Rightarrow \psi)$$

where $S \mathrel{\widehat{=}} \langle d \mid inv\rangle$.

Proof

$wp.\langle \Delta S;\ di?;\ do! \mid p\rangle.\psi$

$\equiv wp.\langle d;\ d';\ di?;\ do! \mid inv \wedge inv' \wedge p\rangle.\psi$ [by a property of Δ-schemas and inclusion]

$\equiv (\exists\, d';\ do! \bullet inv \wedge inv' \wedge p) \wedge (\forall\, d';\ do! \bullet inv \wedge inv' \wedge p \Rightarrow \psi)$ [by definition of wp]

$\equiv inv \wedge (\exists\, d';\ do! \bullet inv' \wedge p) \wedge (\forall\, d';\ do! \bullet inv \wedge inv' \wedge p \Rightarrow \psi)$
[by $\alpha d'$ and $\alpha do!$ are not free in inv]

$\equiv (\exists\, d';\ do! \bullet inv' \wedge p) \wedge (\forall\, d';\ do! \bullet inv \wedge (inv \wedge inv' \wedge p \Rightarrow \psi))$
[by $\alpha d'$ and $\alpha do!$ are not free in inv]

$\equiv (\exists\, d';\ do! \bullet inv' \wedge p) \wedge (\forall\, d';\ do! \bullet inv \wedge (inv' \wedge p \Rightarrow \psi))$ [by predicate calculus]

$\equiv inv \wedge (\exists\, d';\ do! \bullet inv' \wedge p) \wedge (\forall\, d';\ do! \bullet inv' \wedge p \Rightarrow \psi)$
[by $\alpha d'$ and $\alpha do!$ are not free in inv]

□

In this proof we have relied on the fact that the after-state and output variables are not free in inv. This is a consequence of our assumption that the decorations "′", "?" and "!" are not used for any purpose other than those established by the Oxford style of writing Z specifications.

Basic Conversion

Law bC Basic conversion

$$\begin{array}{ll} & \langle \Delta S;\ di?;\ do! \mid p\rangle \\ = & bC \\ & \alpha d, \alpha do! : [inv \land \exists\, d';\ do! \bullet inv' \land p, inv' \land p] \end{array}$$

where $S \mathrel{\hat=} \langle d \mid inv\rangle$

Derivation

$wp.\langle \Delta S;\ di?;\ do! \mid p\rangle.\psi$

$\equiv inv \land (\exists\, d';\ do! \bullet inv' \land p) \land (\forall\, d';\ do! \bullet inv' \land p \Rightarrow \psi)$ [by Lemma D.1]

$\equiv inv \land (\exists\, d';\ do! \bullet inv' \land p) \land (\forall\, d';\ do! \bullet inv' \land p \Rightarrow \psi)[\alpha d/\alpha d']$ [by $\alpha d'$ are not free in d']

$\equiv wp.\alpha d', \alpha do! : [inv \land \exists\, d';\ do! \bullet inv' \land p, inv' \land p]$ [by definition of wp]

□

Law bC Basic conversion (operations that do not modify the state)

$$\begin{array}{ll} & \langle \Xi S;\ di?;\ do! \mid p\rangle \\ = & bC \\ & \alpha do! : [inv \land \exists\, do! \bullet p[\alpha d/\alpha d'], p] \end{array}$$

where $S \mathrel{\hat=} \langle d \mid inv\rangle$

Derivation

$\langle \Xi S;\ di?;\ do! \mid p\rangle$

$=$ by $c_1, \ldots, c_n$ are the state components (elements of αd)

$\langle \Delta S;\ di?;\ do! \mid p \land c_1' = c_1 \land \ldots \land c_n' = c_n\rangle$

$= bC$

$$\alpha d, \alpha do! : \left[\begin{array}{l} inv \land \exists\, d';\ do! \bullet inv' \land p \land c_1' = c_1 \land \ldots \land c_n' = c_n, \\ inv' \land p \land c_1' = c_1 \land \ldots \land c_n' = c_n \end{array}\right]$$

$=$ by predicate calculus

$\alpha d, \alpha do! : [inv \land \exists\, do! \bullet p[\alpha d/\alpha d'], inv' \land p \land c_1' = c_1 \land \ldots c_n' = c_n]$

$= efR$

$\alpha do! : [inv \land \exists\, do! \bullet p[\alpha d/\alpha d'], inv \land p]$

$= sP$ (in both directions)

$\alpha do! : [inv \land \exists\, do! \bullet p[\alpha d/\alpha d'], p]$

□

Law bC Basic conversion (operations that do not modify some state components)

$$\begin{array}{l} \quad \langle \Delta S;\ \Xi T;\ di?;\ do! \mid p \rangle \\ \sqsubseteq \ bC \\ \quad \alpha d_S, \alpha do! : [inv_S \land inv_T \land \exists\, d'_S;\ do! \bullet (inv'_S \land p)[\alpha d_T / \alpha d'_T], (inv'_S \land p)[\alpha d_T / \alpha d'_T]\ ] \end{array}$$

where $S \mathrel{\widehat{=}} \langle T;\ d_S \mid inv_S \rangle$ and $T \mathrel{\widehat{=}} \langle d_T \mid inv_T \rangle$

Derivation

$$\begin{array}{l} \quad \langle \Delta S;\ \Xi T;\ di?;\ do! \mid p \rangle \\ = bC \\ \quad \alpha d_S, \alpha d_T, \alpha do! : \left[\begin{array}{l} inv_S \land inv_T \land \exists\, d'_S;\ d'_T;\ do! \bullet inv'_S \land inv'_T \land p \land \Xi T, \\ inv'_S \land inv'_T \land p \land \Xi T \end{array} \right] \\ \sqsubseteq cfR \\ \quad \alpha d_S, \alpha do! : \left[\begin{array}{l} inv_S \land inv_T \land \exists\, d'_S;\ d'_T;\ do! \bullet inv'_S \land inv'_T \land p \land \Xi T, \\ inv'_S[\alpha d_T / \alpha d'_T] \land inv_T \land p[\alpha d_T / \alpha d'_T] \end{array} \right] \\ \sqsubseteq sP \\ \quad \alpha d_S, \alpha do! : [inv_S \land inv_T \land \exists\, d'_S;\ d'_T;\ do! \bullet inv'_S \land inv'_T \land p \land \Xi T, (inv'_S \land p)[\alpha d_T / \alpha d'_T]\ ] \\ \sqsubseteq wP \\ \quad \alpha d_S, \alpha do! : [inv_S \land inv_T \land \exists\, d'_S;\ do! \bullet (inv'_S \land p)[\alpha d_T / \alpha d'_T], (inv'_S \land p)[\alpha d_T / \alpha d'_T]\ ] \end{array}$$

□

Schema Disjunction

The lemma below is used in the derivation of two formulations of *sdisjC* (schema disjunction conversion).

Lemma D.2 *For all schemas Op_1 and Op_2 that specify operations which act over the same state and have the same input and output variables,*

$$wp.(Op_1 \lor Op_2).\psi \Rightarrow (pre_1 \lor pre_2) \land (pre_1 \Rightarrow wp.Op_1.\psi) \land (pre_2 \Rightarrow wp.Op_2.\psi)$$

where pre $Op_1 \equiv pre_1 \land inv \land t$, pre $Op_2 \equiv pre_2 \land inv \land t$, *$inv$ is the state invariant, and t is the restriction that is introduced by the declarations of the state components and input variables.*

Proof The schemas Op_1 and Op_2 can be written as $\langle \Delta S;\ di?;\ do! \mid p_1 \rangle$ and $\langle \Delta S;\ di?;\ do! \mid p_2 \rangle$, respectively, where $S \mathrel{\widehat{=}} \langle d \mid inv \rangle$.

(Case $pre_1 \equiv inv \land \exists\, d';\ do! \bullet inv' \land p_1$ **and** $pre_2 \equiv inv \land \exists\, d';\ do! \bullet inv' \land p_2$**)**

$$\begin{array}{ll} \quad wp.(Op_1 \lor Op_2).\psi & \\ \equiv\ (wp.Op_1.\mathrm{true} \lor wp.Op_2.\mathrm{true}) \land & \text{[by definition of } wp\text{]} \\ \quad (wp.Op_1.\mathrm{true} \Rightarrow wp.Op_1.\psi) \land (wp.Op_2.\mathrm{true} \Rightarrow wp.Op_2.\psi) & \end{array}$$

$\equiv ((inv \land \exists\, d';\ do! \bullet inv' \land p_1) \lor (inv \land \exists\, d';\ do! \bullet inv' \land p_2)) \land$ [by Lemma D.1]
$((inv \land \exists\, d';\ do! \bullet inv' \land p_1) \Rightarrow wp.Op_1.\psi) \land ((inv \land \exists\, d';\ do! \bullet inv' \land p_2) \Rightarrow wp.Op_2.\psi)$

(Case $pre_1 \equiv inv \land \exists\, d';\ do! \bullet inv' \land p_1$ **and** $pre_2 \equiv \exists\, d';\ do! \bullet inv' \land p_2$**)**

$wp.(Op_1 \lor Op_2).\psi$
$\equiv ((inv \land \exists\, d';\ do! \bullet inv' \land p_1) \lor (inv \land \exists\, d';\ do! \bullet inv' \land p_2)) \land$ [by the previous case]
$((inv \land \exists\, d';\ do! \bullet inv' \land p_1) \Rightarrow wp.Op_1.\psi) \land ((inv \land \exists\, d';\ do! \bullet inv' \land p_2) \Rightarrow wp.Op_2.\psi)$
$\equiv ((inv \land \exists\, d';\ do! \bullet inv' \land p_1) \lor (\exists\, d';\ do! \bullet inv' \land p_2)) \land inv \land$ [by predicate calculus]
$((inv \land \exists\, d';\ do! \bullet inv' \land p_1) \Rightarrow wp.Op_1.\psi) \land ((inv \land \exists\, d';\ do! \bullet inv' \land p_2) \Rightarrow wp.Op_2.\psi)$
$\Rightarrow ((inv \land \exists\, d';\ do! \bullet inv' \land p_1) \lor (\exists\, d';\ do! \bullet inv' \land p_2)) \land$ [by predicate calculus]
$((inv \land \exists\, d';\ do! \bullet inv' \land p_1) \Rightarrow wp.Op_1.\psi) \land ((\exists\, d';\ do! \bullet inv' \land p_2) \Rightarrow wp.Op_2.\psi)$

The eases in which $pre_1 \equiv \exists\, d';\ do! \bullet inv' \land p_1$ and $pre_2 \equiv inv \land \exists\, d';\ do! \bullet inv' \land p_2$, and in which $pre_1 \equiv \exists\, d';\ do! \bullet inv' \land p_1$ and $pre_2 \equiv \exists\, d';\ do! \bullet inv' \land p_2$ are similar. Sinee t is an axiom (it reflects type declarations), we do not need to consider the cases in which t is a conjunct of pre_1 or pre_2.

□

Law *sdisjC* Schema disjunction conversion

$Op_1 \lor Op_2$
$\sqsubseteq$ *sdisjC*
if $pre_1 \rightarrow Op_1\ [\!]\ pre_2 \rightarrow Op_2$ **fi**

where

- pre $Op_1 \equiv pre_1 \land inv \land t$;
- pre $Op_2 \equiv pre_2 \land inv \land t$;
- inv is the state invariant;
- t is the restriction that is introduced by the declarations of the state components and input variables.

Syntactic Restriction Op_1 and Op_2 act over the same state and have the same input and output variables.

Derivation

$wp.(Op_1 \lor Op_2).\psi$
$\Rightarrow (pre_1 \lor pre_2) \land (pre_1 \Rightarrow wp.Op_1.\psi) \land (pre_2 \Rightarrow wp.Op_2.\psi)$ [by Lemma D.2]
$\equiv wp.$**if** $pre_1 \rightarrow Op_1\ [\!]\ pre_2 \rightarrow Op_2$ **fi**$.\psi$ [by definition of wp]

□

Law *sdisjC* Schema disjunction conversion with variable introduction

$Op_1 \lor Op_2$

$\sqsubseteq$ *sdisjC*

$|[\,\mathbf{var}\ v : t \bullet v : [\mathrm{true}, \phi[v'/v]\,]\ ;\ \mathbf{if}\ \psi_1 \rightarrow \{\phi \land \psi_1\}\ Op_1\ []\ \psi_2 \rightarrow \{\phi \land \psi_2\}\ Op_2\ \mathbf{fi}\,]|$

provided

- $\phi \land (pre_1 \lor pre_2) \Rightarrow \psi_1 \lor \psi_2$
- $\phi \land (pre_1 \lor pre_2) \Rightarrow (\psi_i \Rightarrow pre_i)$ for $i = 1, 2$

where

- $\mathrm{pre}\, Op_1 = pre_1 \land inv \land t$;
- $\mathrm{pre}\, Op_2 = pre_2 \land inv \land t$;
- inv is the state invariant;
- t is the restriction that is introduced by the declarations of the state components and input variables.

Syntactic Restrictions

- ϕ, ψ_1, and ψ_2 are well-scoped and well-typed predicates;
- ϕ, ψ_1, and ψ_2 have no free dashed variables;
- Op_1 and Op_2 act over the same state and have the same input and output variables;
- v and v' are not free in Op_1 and Op_2.

Derivation

$wp.(Op_1 \lor Op_2).\psi$

$\equiv \forall v : t \bullet wp.(Op_1 \lor Op_2).\psi$ [by v and v' are not free in Op_1, Op_2, and ψ]

$\Rightarrow \forall v : t \bullet (pre_1 \lor pre_2) \land (pre_1 \Rightarrow wp.Op_1.\psi) \land (pre_2 \Rightarrow wp.Op_2.\psi)$ [by Lemma D.2]

$\Rightarrow \forall v : t \bullet \phi \Rightarrow \phi \land (pre_1 \lor pre_2) \land (pre_1 \Rightarrow wp.Op_1.\psi) \land (pre_2 \Rightarrow wp.Op_2.\psi)$
[by predicate calculus]

$\Rightarrow \forall v : t \bullet \phi \Rightarrow (\psi_1 \lor \psi_2) \land (\psi_1 \Rightarrow wp.Op_1.\psi) \land (\psi_2 \Rightarrow wp.Op_2.\psi)$ [by the provisos]

$\equiv \forall v : t \bullet \phi \Rightarrow (\psi_1 \lor \psi_2) \land (\psi_1 \Rightarrow \phi \land \psi_1 \land wp.Op_1.\psi) \land (\psi_2 \Rightarrow \phi \land \psi_2 \land wp.Op_2.\psi)$
[by predicate calculus]

$\equiv \forall v : t \bullet \forall v : t \bullet \phi \Rightarrow$ [by v is not free in t]
$(\psi_1 \lor \psi_2) \land (\psi_1 \Rightarrow \phi \land \psi_1 \land wp.Op_1.\psi) \land (\psi_2 \Rightarrow \phi \land \psi_2 \land wp.Op_2.\psi)$

$\equiv \forall v : t \bullet (\forall v' : t \bullet \phi[v'/v] \Rightarrow$ [by predicate calculus]
$((\psi_1 \lor \psi_2) \land (\psi_1 \Rightarrow \phi \land \psi_1 \land wp.Op_1.\psi) \land (\psi_2 \Rightarrow \phi \land \psi_2 \land wp.Op_2.\psi))[v'/v])$

$\equiv \forall v : t \bullet (\forall v' : t \bullet \phi[v'/v] \Rightarrow$ [by a property of substitution]
$((\psi_1 \lor \psi_2) \land (\psi_1 \Rightarrow \phi \land \psi_1 \land wp.Op_1.\psi) \land (\psi_2 \Rightarrow \phi \land \psi_2 \land wp.Op_2.\psi))[v'/v]')[\_/']$

$\equiv \forall v : t \bullet (\forall v' : t \bullet \phi[v'/v] \Rightarrow ((\psi_1 \vee \psi_2) \wedge$ [a property of substitution]

$(\psi_1 \Rightarrow \phi \wedge \psi_1 \wedge wp.Op_1.\psi) \wedge (\psi_2 \Rightarrow \phi \wedge \psi_2 \wedge wp.Op_2.\psi))[v'/v]')[v/v'][\_/']$

$\equiv wp. \,|[\, \mathbf{var}\ v : t \bullet v : [\mathrm{true}, \phi[v'/v]\,] \;;\ \mathbf{if}\ \psi_1 \rightarrow \{\phi \wedge \psi_1\}\ Op_1 \,[]\, \psi_2 \rightarrow \{\phi \wedge \psi_2\}\ Op_2\ \mathbf{fi}\,]| \,.\, \psi$

[by definition of wp]

□

Law *sdisjC* Schema disjunction conversion with boolean variable introduction

$Op_1 \vee Op_2$

$\sqsubseteq$ *sdisjC*

$|[\, \mathbf{var}\ b : Boolean \bullet b : [\mathrm{true}, b' \Leftrightarrow pre_1] \;;\ \mathbf{if}\ b \rightarrow Op_1 \,[]\, pre_2 \rightarrow Op_2\ \mathbf{fi}\,]|$

where

- pre $Op_1 \equiv pre_1 \wedge inv \wedge t$;
- pre $Op_2 \equiv pre_2 \wedge inv \wedge t$;
- inv is the state invariant;
- t is the restriction that is iutroduced by the declarations of the state components and input variables.

Syntactic Restrictions

- Op_1 and Op_2 act over the same state and have the same input and output variables;
- b and b' are not free in Op_1 and Op_2.

Derivation

$Op_1 \vee Op_2$

$\sqsubseteq$ *sdisjC*

$|[\, \mathbf{var}\ b : Boolean \bullet$

$b : [\mathrm{true}, (b \Leftrightarrow pre_1)[b'/b]\,] \;;$ (*i*)

$\mathbf{if}\ b \rightarrow \{(b \Leftrightarrow pre_1) \wedge b\}\ Op_1$ (*ii*)

$[]\ pre_2 \rightarrow \{(b \Leftrightarrow pre_1) \wedge pre_2\}\ Op_2$ (*iii*)

$\mathbf{fi}$

$]|$

The proof-obligations that are generated by this application of *sdisjC* are discharged in what follows.

$(b \Leftrightarrow pre_1) \wedge (pre_1 \vee pre_2)$

$\equiv (b \Leftrightarrow pre_1) \wedge (b \vee pre_2)$ [by predicate calculus]

$\Rightarrow b \vee pre_2$ [by predicate calculus]

$(b \Leftrightarrow pre_1) \wedge (pre_1 \vee pre_2) \wedge b$

$\equiv (b \Leftrightarrow pre_1) \wedge (pre_1 \vee pre_2) \wedge pre_1$ [by predicate calculus]

$\Rightarrow pre_1$ [by predicate calculus]

$(b \Leftrightarrow pre_1) \wedge (pre_1 \vee pre_2) \wedge pre_2$

$\Rightarrow pre_2$ [by predicate calculus]

The programs (i), (ii), and (iii) are further refined below.

$(i) \sqsubseteq sP$
    $b : [\text{true}, b' \Leftrightarrow pre_1]$

$(ii) \sqsubseteq assumpR$
    skip ; Op_1
  $= slC$
    Op_1

$(iii) \sqsubseteq assumpR$
    skip ; Op_2
  $= slC$
    Op_2

The collected code is exactly the variable block in the above formulation of *sdisjC*.

□

Schema Conjunction

Law *sconjC* Schema conjunction conversion

$Op_1 \wedge Op_2$
$\sqsubseteq$ *sconjC*
Op_1 ; Op_2

Syntactic Restriction Op_1 and Op_2 have no common free variables.

Derivation The schemas Op_1 and Op_2 can be written in the form $(d_1;\ d_1';\ di_1?;\ do_1! \mid p_1)$ and $(d_2;\ d_2';\ di_2?;\ do_2! \mid p_2)$, respectively.

$wp.(Op_1 \wedge Op_2).\psi$

$\equiv wp.(d_1;\ d_2;\ d_1';\ d_2';\ di_1?;\ di_2?;\ do_1!;\ do_2! \mid p_1 \wedge p_2).\psi$
[by a property of schema conjunction]

$\equiv (\exists\, d_1';\ d_2';\ do_1!;\ do_2! \bullet p_1 \wedge p_2) \wedge (\forall\, d_1';\ d_2';\ do_1!;\ do_2! \bullet p_1 \wedge p_2 \Rightarrow \psi)$ [by definition of wp]

$\equiv (\exists d_1';\ do_1! \bullet p_1 \wedge \exists d_2';\ do_2! \bullet p_2) \wedge (\forall d_1';\ d_2';\ do_1!;\ do_2! \bullet p_1 \wedge p_2 \Rightarrow \psi)$

[by $\alpha d_2'$ and $\alpha do_2!$ are not free in p_1]

$\equiv (\exists d_1';\ do_1! \bullet p_1) \wedge (\exists d_2';\ do_2! \bullet p_2) \wedge (\forall d_1';\ d_2';\ do_1!;\ do_2! \bullet p_1 \wedge p_2 \Rightarrow \psi)$

[by $\alpha d_1'$ and $\alpha do_1!$ are not free in d_2', $do_2!$, and p_2]

$\equiv (\exists d_1';\ do_1! \bullet p_1) \wedge (\exists d_2';\ do_2! \bullet p_2) \wedge (\forall d_1';\ do_1! \bullet p_1 \Rightarrow \forall d_2';\ do_2! \bullet p_2 \Rightarrow \psi)$

[by $\alpha d_2'$ and $\alpha do_2!$ are not free in p_1]

$\equiv (\exists d_1';\ do_1! \bullet p_1) \wedge (\forall d_1';\ do_1! \bullet (\exists d_2';\ do_2! \bullet p_2) \wedge (p_1 \Rightarrow (\forall d_2';\ do_2! \bullet p_2 \Rightarrow \psi)))$

[by $\alpha d_1'$ and $\alpha do_1!$ are not free in d_2', $do_2!$, and p_2]

$\Rightarrow (\exists d_1';\ do_1! \bullet p_1) \wedge (\forall d_1';\ do_1! \bullet p_1 \Rightarrow (\exists d_2';\ do_2! \bullet p_2) \wedge (\forall d_2';\ do_2! \bullet p_2 \Rightarrow \psi))$

[by predicate calculus]

$\equiv (\exists d_1';\ do_1! \bullet p_1) \wedge (\forall d_1';\ do_1! \bullet p_1 \Rightarrow ((\exists d_2';\ do_2! \bullet p_2) \wedge (\forall d_2';\ do_2! \bullet p_2 \Rightarrow \psi))[\alpha d_1'/\alpha d_1])$

[by αd_1 are not free in d_2', $do_2!$, p_2, and ψ]

$\equiv wp.(Op_1\ ;\ Op_2).\psi$ [by definition of wp]

$\Box$

Law *sconjC* Schema conjunction conversion (hierarchical specification)

$\langle \Delta S_1;\ di_1?;\ do_1! \mid p_1 \rangle \wedge \Delta S_2$

$\sqsubseteq$ *sconjC*

$[\![$ **con** $dcl \bullet$

$\langle \Delta S_1;\ di_1?;\ do_1! \mid p_1 \rangle$;

$\langle \Xi S_1;\ d_2;\ d_2' \mid inv_1[cl/\alpha d_1] \wedge inv_2[cl/\alpha d_1] \wedge p_1[cl/\alpha d_1][\_/'] \wedge inv_2' \rangle$

$]\!]$

provided $(pre_C \wedge inv_1 \wedge inv_1' \wedge p_1) \Rightarrow pre_2'$

where

- $S_1 \mathrel{\widehat=} \langle d_1 \mid inv_1 \rangle$ and $S_2 \mathrel{\widehat=} \langle S_1;\ d_2 \mid inv_2 \rangle$;
- $\mathrm{pre}(\langle \Delta S_1;\ di_1?;\ do_1! \mid p_1 \rangle \wedge \Delta S_2) \equiv pre_C \wedge inv_1 \wedge inv_2 \wedge t_C$;
- $\mathrm{pre}(\Delta S_2 \wedge \Xi S_1) \equiv pre_2 \wedge inv_1 \wedge inv_2 \wedge t_2$;
- t_C and t_2 are the restrictions introduced by $d_1;\ d_2;\ di_1?$ and $d_1;\ d_2$, respectively;
- dcl declares the constants of cl.

Syntactic Restrictions

- The components of ΔS_1 are the only common free variables of $\langle \Delta S_1;\ di_1?;\ do_1! \mid p_1 \rangle$ and ΔS_2;
- The names of cl and cl' are not free in $\langle \Delta S_1;\ di_1?;\ do_1! \mid p_1 \rangle$ and ΔS_2;
- cl and αd_1 have the same length;
- The constants of cl have the same type as the corresponding variables of αd_1.

Derivation

$$
\begin{array}{l}
\langle \Delta S_1;\ di_1?;\ do_1! \mid p_1 \rangle \land \Delta S_2 \\
= \text{by a property of schema conjunction} \\
\langle \Delta S_2;\ di_1?;\ do_1! \mid p_1 \rangle \\
= bC \\
\alpha d_1, \alpha d_2, \alpha do_1! : [inv_1 \land inv_2 \land \exists\, d_1';\ d_2';\ do_1! \bullet inv_1' \land inv_2' \land p_1, inv_1' \land inv_2' \land p_1] \\
\sqsubseteq seqcl \\
|[\,\mathbf{con}\ dcl\ \bullet \\
\qquad \alpha d_1, \alpha do_1! : \left[\begin{array}{l} inv_1 \land inv_2 \land \exists\, d_1';\ d_2';\ do_1! \bullet inv_1' \land inv_2' \land p_1, \\ (\exists\, d_1';\ d_2';\ do_1! \bullet inv_1' \land inv_2' \land p_1) \land inv_1 \land inv_2 \land inv_1' \land p_1 \end{array} \right]; \qquad \lhd \\
\qquad \alpha d_1, \alpha d_2, \alpha do_1! : \left[\begin{array}{l} \left(\begin{array}{l} \exists\, d_1';\ d_2';\ do_1! \bullet inv_1' \land inv_2' \land p_1 \\ inv_1 \land inv_2 \land inv_1' \land p_1 \end{array} \right) [cl/\alpha d_1][\_/'], \\ (inv_1' \land inv_2' \land p_1)[cl/\alpha d_1] \end{array} \right] \qquad (i) \\
]| \\
\sqsubseteq sP \\
\alpha d_1, \alpha do_1! : [inv_1 \land inv_2 \land \exists\, d_1';\ d_2';\ do_1! \bullet inv_1' \land inv_2' \land p_1, inv_1' \land p_1] \\
\sqsubseteq wP \\
\alpha d_1, \alpha do_1! : [inv_1 \land \exists\, d_1';\ do_1! \bullet inv_1' \land p_1, inv_1' \land p_1] \\
= bC \\
\langle \Delta S_1;\ di_1?;\ do_1! \mid p_1 \rangle
\end{array}
$$

The refinement of (i) proceeds as follows.

$$
\begin{array}{l}
(i) \sqsubseteq sP \\
\begin{array}{l} \alpha d_1, \\ \alpha d_2, \\ \alpha do_1! \end{array} : \left[\begin{array}{l} \left(\begin{array}{l} \exists\, d_1';\ d_2';\ do_1! \bullet inv_1' \land inv_2' \land p_1 \\ inv_1 \land inv_2 \land inv_1' \land p_1 \end{array} \right) [cl/\alpha d_1][\_/'], \\ inv_1' \land inv_1[cl/\alpha d_1] \land inv_2[cl/\alpha d_1] \land p_1[cl/\alpha d_1][\_/'] \land inv_2' \land \Xi S_1 \end{array} \right]
\end{array}
$$

Below we discharge the proof-obligation generated by this application of sP.

$$
\begin{array}{ll}
((\exists\, d_1';\ d_2';\ do_1! \bullet inv_1' \land inv_2' \land p_1) \land inv_1 \land inv_2 \land inv_1' \land p_1)[cl/\alpha d_1][\_/'] \land & \\
inv_1' \land inv_1[cl/\alpha d_1] \land inv_2[cl/\alpha d_1] \land p_1[cl/\alpha d_1][\_/'] \land inv_2' \land \Xi S_1 & \\
\Rightarrow inv_1' \land inv_2' \land p_1[cl/\alpha d_1][\_/'] \land \Xi S_1 & \text{[by predicate calculus]} \\
\equiv inv_1' \land inv_2' \land p_1[cl/\alpha d_1][\_/'][\alpha d_1'/\alpha d_1] \land \Xi S_1 & \text{[by } d_1 \text{ declares the components of } S_1] \\
\equiv inv_1' \land inv_2' \land p_1[cl/\alpha d_1] \land \Xi S_1 & \text{[by } \alpha d_2' \text{ are not free in } p_1] \\
\Rightarrow (inv_1' \land inv_2' \land p_1)[cl/\alpha d_1] & \text{[by predicate calculns and a property of substitution]}
\end{array}
$$

We continue as follows.

$$\begin{array}{l} \sqsubseteq wP \\ \begin{array}{l}\alpha d_1,\\ \alpha d_2,\\ \alpha do_1!\end{array} : \left[\begin{array}{l} inv_1 \land \exists\, d_2' \bullet inv_1[cl/\alpha d_1] \land inv_2[cl/\alpha d_1] \land p_1[cl/\alpha d_1][\_/'] \land inv_2'[\alpha d_1/\alpha d_1'], \\ inv_1' \land inv_1[cl/\alpha d_1] \land inv_2[cl/\alpha d_1] \land p_1[cl/\alpha d_1][\_/'] \land inv_2' \land \Xi S_1 \end{array}\right] \end{array}$$

This application of *wP* gives rise to the proof-obligation that we discharge below.

$$\begin{array}{ll} ((\exists\, d_1';\ d_2';\ do_1! \bullet inv_1' \land inv_2' \land p_1) \land inv_1 \land inv_2 \land inv_1' \land p_1)[cl/\alpha d_1][\_/'] & \\ \equiv\ inv_1 \land inv_1[cl/\alpha d_1] \land inv_2[cl/\alpha d_1] \land p_1[cl/\alpha d_1][\_/'] \land & \text{[by a property of substitution]} \\ \quad ((\exists\, d_1';\ d_2';\ do_1! \bullet inv_1' \land inv_2' \land p_1) \land inv_1 \land inv_2 \land inv_1' \land p_1)[cl/\alpha d_1][\_/'] & \\ \Rightarrow inv_1 \land inv_1[cl/\alpha d_1] \land inv_2[cl/\alpha d_1] \land p_1[cl/\alpha d_1][\_/'] \land (\exists\, d_2' \bullet inv_2'[\alpha d_1/\alpha d_1'])'[cl/\alpha d_1][\_/'] & \\ & \text{[by the proviso (in its weakest form)]} \\ \equiv\ inv_1 \land inv_1[cl/\alpha d_1] \land inv_2[cl/\alpha d_1] \land p_1[cl/\alpha d_1][\_/'] \land (\exists\, d_2' \bullet inv_2'[\alpha d_1/\alpha d_1']) & \\ & \text{[by } \alpha d_1' \text{ and } \alpha d_2' \text{ are not free in } d_2'] \\ \equiv\ inv_1 \land \exists\, d_2' \bullet inv_1[cl/\alpha d_1] \land inv_2[cl/\alpha d_1] \land p_1[cl/\alpha d_1][\_/'] \land inv_2'[\alpha d_1/\alpha d_1'] & \\ & \text{[by } \alpha d_2' \text{ are not free in } inv_1[cl/\alpha d_1],\ inv_2[cl/\alpha d_1], \text{ and } p_1[cl/\alpha d_1][\_/']] \end{array}$$

Finally we get to the required result.

$$\begin{array}{l} \sqsubseteq cfR \\ \alpha d_1, \alpha d_2 : \left[\begin{array}{l} inv_1 \land \exists\, d_2' \bullet inv_1[cl/\alpha d_1] \land inv_2[cl/\alpha d_1] \land p_1[cl/\alpha d_1][\_/'] \land inv_2'[\alpha d_1/\alpha d_1'], \\ inv_1' \land inv_1[cl/\alpha d_1] \land inv_2[cl/\alpha d_1] \land p_1[cl/\alpha d_1][\_/'] \land inv_2' \land \Xi S_1 \end{array}\right] \\ = bC \\ \quad \langle \Xi S_1;\ d_2;\ d_2' \mid inv_1[cl/\alpha d_1] \land inv_2[cl/\alpha d_1] \land p_1[cl/\alpha d_1][\_/'] \land inv_2' \rangle \end{array}$$

□

Assignment

Law *assC* Assignment conversion

$$\begin{array}{ll} & \langle \Delta S;\ di?;\ do! \mid c_1' = e_1 \land \ldots \land c_n' = e_n \land o_1! = e_{n+1} \land \ldots \land o_m! = e_{n+m} \rangle \\ \sqsubseteq & assC \\ & c_1, \ldots, c_n, o_1!, \ldots, o_m! := e_1, \ldots, e_{n+m} \end{array}$$

provided $inv[e_1, \ldots, e_n / c_1, \ldots, c_n]$

where

- $S \mathrel{\widehat{=}} \langle d \mid inv \rangle$
- $c_1, \ldots, c_n$ are state components (elements of αd);
- $o_1!, \ldots, o_m!$ are output variables (elements of $\alpha do!$).

Syntactic Restriction $\alpha d'$ and $\alpha do!$ are not free in $e_1, \ldots, e_{n+m}$.

Derivation

$$
\begin{array}{l}
wp.(\Delta S;\ di?;\ do! \mid c_1' = c_1 \land \ldots \land c_n' = e_n \land o_1! = e_{n+1} \land \ldots \land o_m! = e_{n+m}).\psi' \\
\Rightarrow (\forall d';\ do! \bullet inv' \land c_1' = e_1 \land \ldots \land c_n' = e_n \land o_1! = e_{n+1} \land \ldots \land o_m! = e_{n+m} \Rightarrow \psi') \\
\hfill [\text{by definition of } wp] \\
\Rightarrow (\forall d';\ do! \bullet inv' \land c_1' = e_1 \land \ldots \land c_n' = e_n \land \\
\quad c_{n+1}' = c_{n+1} \land \ldots \land c_{n+l}' = c_{n+l} \land o_1! = e_{n+1} \land \ldots \land o_m! = e_{n+m} \Rightarrow \psi') \\
\quad [\text{by predicate calculus } (c_{n+1}, \ldots, c_{n+l} \text{ are the state components not in } c_1, \ldots, c_n)] \\
\equiv inv'[e_1, \ldots, e_n, c_{n+1}, \ldots, c_{n+l}/c_1', \ldots, c_{n+l}'] \Rightarrow \\
\quad \psi'[e_1, \ldots, e_n, c_{n+1}, \ldots, c_{n+l}, e_{n+1}, \ldots, e_{n+m}/c_1', \ldots, c_{n+l}', o_1!, \ldots, o_m!] \\
\hfill [\text{by } \alpha d' \text{ and } \alpha do! \text{ are not free in } e_1, \ldots, e_{n+m}] \\
\equiv inv[c_1, \ldots, e_n/c_1, \ldots, c_n] \Rightarrow \psi[e_1, \ldots, e_{n+m}/c_1, \ldots, c_n, o_1!, \ldots, o_m!] \\
\hfill [\text{by } c_{n+1}, \ldots, c_{n+l} \text{ are the state components not in } c_1, \ldots, c_n] \\
\Rightarrow \psi[e_1, \ldots, e_{n+m}/c_1, \ldots, c_n, o_1!, \ldots, o_m!] \hfill [\text{by the proviso}] \\
\equiv wp.c_1, \ldots, c_n, o_1!, \ldots, o_m! := e_1, \ldots, e_{n+m}.\psi' \hfill [\text{by definition of } wp]
\end{array}
$$

□

Schema Composition

Law *scompC* Schema composition conversion

$$
\begin{array}{ll}
 & Op_1 \mathbin{⨾} Op_2 \\
\sqsubseteq & scompC \\
 & Op_1\ ;\ Op_2
\end{array}
$$

provided $(pre_C \land Op_1) \Rightarrow pre_2'$

where

- $\text{pre}(Op_1 \mathbin{⨾} Op_2) \equiv pre_C \land inv \land t \land t_1 \land t_2$;
- $\text{pre } Op_2 \equiv pre_2 \land inv \land t \land t_2$;
- inv is the state invariant;
- t, t_1, and t_2 are the restrictions that are introduced by the declarations of the state components, of the input variables of Op_1, and of the input variables of Op_2, respectively.

Syntactic Restrictions

- Op_1 and Op_2 act over the same state;
- Op_1 and Op_2 have no common output variables.

Derivation The schemas Op_1 and Op_2 can be written in the form $\langle \Delta S;\ di_1?;\ do_1! \mid p_1\rangle$ and $\langle \Delta S;\ di_2?;\ do_2! \mid p_2\rangle$, respectively, where $S \mathrel{\widehat=} \langle d \mid inv\rangle$ and $\alpha do_1! \cap \alpha do_2! = \varnothing$.

$$\langle \Delta S;\ di_1?;\ do_1! \mid p_1\rangle \mathbin{;} \langle \Delta S;\ di_2?;\ do_2! \mid p_2\rangle$$

$=$ by a property of schema composition

$$\langle \Delta S;\ di_1?;\ di_2?;\ do_1!;\ do_2! \mid \exists\, d'' \bullet inv'' \wedge p_1[\alpha d''/\alpha d'] \wedge p_2[\alpha d''/\alpha d]\rangle$$

$\sqsubseteq$ *bC*

$$\alpha d, \alpha do_1!, \alpha do_2! : \begin{bmatrix} inv \wedge \exists\, d';\ do_1!;\ do_2! \bullet inv' \wedge \exists\, d'' \bullet inv'' \wedge p_1[\alpha d''/\alpha d'] \wedge p_2[\alpha d''/\alpha d], \\ inv' \wedge \exists\, d'' \bullet inv'' \wedge p_1[\alpha d''/\alpha d'] \wedge p_2[\alpha d''/\alpha d] \end{bmatrix}$$

$\sqsubseteq$ *seqcI*

$|[$ **con** $dcl \bullet$

$$\begin{array}{l} \alpha d, \\ \alpha do_1!, : \\ \alpha do_2! \end{array} \begin{bmatrix} inv \wedge \exists\, d';\ do_1!;\ do_2! \bullet inv' \wedge \exists\, d'' \bullet inv'' \wedge p_1[\alpha d''/\alpha d'] \wedge p_2[\alpha d''/\alpha d], \\ \left(\begin{array}{l} \exists\, d';\ do_1!;\ do_2! \bullet inv' \wedge \exists\, d'' \bullet inv'' \wedge p_1[\alpha d''/\alpha d'] \wedge p_2[\alpha d''/\alpha d] \\ inv \wedge inv' \wedge p_1 \end{array}\right) \end{bmatrix}; \quad \lhd$$

$$\begin{array}{l} \alpha d, \\ \alpha do_1!, : \\ \alpha do_2! \end{array} \begin{bmatrix} \left(\begin{array}{l} \exists\, d';\ do_1!;\ do_2! \bullet inv' \wedge \\ \qquad \exists\, d'' \bullet inv'' \wedge p_1[\alpha d''/\alpha d'] \wedge p_2[\alpha d''/\alpha d] \\ inv \wedge inv' \wedge p_1 \end{array}\right)[cl/\alpha d][\_/'], \\ (inv' \wedge \exists\, d'' \bullet inv'' \wedge p_1[\alpha d''/\alpha d'] \wedge p_2[\alpha d''/\alpha d])[cl/\alpha d] \end{bmatrix} \quad (i)$$

$]|$

$\sqsubseteq$ *sP*

$$\begin{array}{l} \alpha d, \\ \alpha do_1!, : \\ \alpha do_2! \end{array} \begin{bmatrix} inv \wedge \exists\, d';\ do_1!;\ do_2! \bullet inv' \wedge \exists\, d'' \bullet inv'' \wedge p_1[\alpha d''/\alpha d'] \wedge p_2[\alpha d''/\alpha d], \\ inv' \wedge p_1 \end{bmatrix}$$

$\sqsubseteq$ *wP*

$$\alpha d, \alpha do_1!, \alpha do_2! : [inv \wedge \exists\, d';\ do_1! \bullet inv' \wedge p_1, inv' \wedge p_1]$$

The proof-obligation that is generated by the application of *wP* is discharged below.

$inv \wedge \exists\, d';\ do_1!;\ do_2! \bullet inv' \wedge \exists\, d'' \bullet inv'' \wedge p_1[\alpha d''/\alpha d'] \wedge p_2[\alpha d''/\alpha d]$

$\Rightarrow inv \wedge \exists\, d';\ do_1!;\ do_2! \bullet \exists\, d'' \bullet inv'' \wedge p_1[\alpha d''/\alpha d']$ [by predicate calculus]

$\equiv inv \wedge \exists\, d';\ do_1! \bullet \exists\, d'' \bullet inv'' \wedge p_1[\alpha d''/\alpha d']$ [by $\alpha do_2!$ are not free in d'', inv'', and p_1]

$\equiv inv \wedge \exists\, d';\ do_1! \bullet \exists\, d'' \bullet (inv' \wedge p_1)[\alpha d''/\alpha d']$ [by a property of substitution]

$\equiv inv \wedge \exists\, d';\ do_1! \bullet \exists\, d' \bullet inv' \wedge p_1$ [by $\alpha d''$ are not free in inv' and p_1]

$\equiv inv \wedge \exists\, d';\ do_1! \bullet inv' \wedge p_1$ [by $\alpha d'$ $(\alpha do_1!)$ are not free in $do_1!$ (d')]

The refinement proceeds as follows.

$\sqsubseteq$ *cfR*

$$\alpha d, \alpha do_1! : [inv \wedge \exists\, d';\ do_1! \bullet inv' \wedge p_1, inv' \wedge p_1]$$

$= bC$

$(\Delta S;\ di_1?, do_1! \mid p_1)$

The refinement of (i) is as follows.

$(i) \sqsubseteq cfR$

$$\alpha d, \alpha do_2! : \left[\begin{array}{l} \left(\begin{array}{l} \exists\, d';\ do_1!;\ do_2! \bullet inv' \wedge \\ \qquad \exists\, d'' \bullet inv'' \wedge p_1[\alpha d''/\alpha d'] \wedge p_2[\alpha d''/\alpha d] \\ inv \wedge inv' \wedge p_1 \end{array} \right) [cl/\alpha d][\_/'], \\ (inv' \wedge \exists\, d'' \bullet inv'' \wedge p_1[\alpha d''/\alpha d'] \wedge p_2[\alpha d''/\alpha d])[cl/\alpha d] \end{array} \right]$$

$\sqsubseteq sP$

$$\alpha d, \alpha do_2! : \left[\left(\begin{array}{l} \exists\, d';\ do_1!;\ do_2! \bullet inv' \wedge \\ \qquad \exists\, d'' \bullet inv'' \wedge p_1[\alpha d''/\alpha d'] \wedge p_2[\alpha d''/\alpha d] \\ inv \wedge inv' \wedge p_1 \end{array} \right) [cl/\alpha d][\_/'], inv' \wedge p_2 \right]$$

The application of sP generates a proof-obligation that is discharged below.

$$\left(\begin{array}{l} \exists\, d';\ do_1!;\ do_2! \bullet inv' \wedge \exists\, d'' \bullet inv'' \wedge p_1[\alpha d''/\alpha d'] \wedge p_2[\alpha d''/\alpha d] \\ inv \wedge inv' \wedge p_1 \end{array} \right) [cl/\alpha d][\_/'] \wedge inv' \wedge p_2$$

$\Rightarrow inv' \wedge (inv' \wedge p_1)[cl/\alpha d][\_/'] \wedge p_2$ [by predicate calculus]

$\Rightarrow inv' \wedge \exists\, d'' \bullet (inv' \wedge p_1)[cl/\alpha d][\_/'][\alpha d''/\alpha d] \wedge p_2[\alpha d''/\alpha d]$ [by predicate calculus]

$\equiv inv' \wedge \exists\, d'' \bullet (inv' \wedge p_1)[cl/\alpha d][\alpha d''/\alpha d'] \wedge p_2[\alpha d''/\alpha d]$

[by d declares the state components (program variables)]

$\equiv inv' \wedge \exists\, d'' \bullet (inv' \wedge p_1)[\alpha d''/\alpha d'][cl/\alpha d] \wedge p_2[\alpha d''/\alpha d]$ [by cl are fresh]

$\equiv inv' \wedge \exists\, d'' \bullet (inv' \wedge p_1)[\alpha d''/\alpha d'][cl/\alpha d] \wedge p_2[\alpha d''/\alpha d][cl/\alpha d]$

[by a property of substitution]

$\equiv inv' \wedge (\exists\, d'' \bullet (inv' \wedge p_1)[\alpha d''/\alpha d'] \wedge p_2[\alpha d''/\alpha d])[cl/\alpha d]$

[by αd are not free in d'' and cl are fresh]

$\equiv inv' \wedge (\exists\, d'' \bullet inv'' \wedge p_1[\alpha d''/\alpha d'] \wedge p_2[\alpha d''/\alpha d])[cl/\alpha d]$ [by a property of substitution]

Below we continue with the refinement.

$\sqsubseteq wP$

$\alpha d, \alpha do_2! : [inv \wedge \exists\, d';\ do_2! \bullet inv' \wedge p_2, inv' \wedge p_2]$

The application of wP generates a proof-obligation that is discharged below.

$$\left(\begin{array}{l} \exists\, d';\ do_1!;\ do_2! \bullet inv' \wedge \exists\, d'' \bullet inv'' \wedge p_1[\alpha d''/\alpha d'] \wedge p_2[\alpha d''/\alpha d] \\ inv \wedge inv' \wedge p_1 \end{array} \right) [cl/\alpha d][\_/']$$

$\Rightarrow (inv \wedge \exists\, d';\ do_2! \bullet inv' \wedge p_2)'[cl/\alpha d][\_/']$ [by the proviso (in its weakest form)]

$\equiv inv \wedge \exists\, d';\ do_2! \bullet inv' \wedge p_2$ [by a property of substitution]

Finally, we get to the conclusion below.

$$\begin{array}{l} = bC \\ \quad \langle \Delta S;\ di_2?;\ do_2! \mid p_2 \rangle \end{array}$$

As the constants of *cl* are not free in either $\langle \Delta S;\ di_1?;\ do_1! \mid p_1 \rangle$ or $\langle \Delta S;\ di_2?;\ do_2! \mid p_2 \rangle$, we can use *conR* to remove the constant block.

□

Promotion

Law *promC* Promotion conversion

$$\begin{array}{ll} & \exists \Delta L \bullet \Phi \wedge Op \\ \sqsubseteq & promC \\ & [\![\, \mathbf{proc}\ pn \mathrel{\widehat=} (\mathbf{val\text{-}res}\ r : L \bullet \langle r, r' : L \mid (inv \wedge inv' \wedge p)[r.x_i, r'.x_i, /x_i, x_i'] \rangle) \bullet pn(f\ x?)\ ]\!] \end{array}$$

where

$$\begin{array}{l} L \mathrel{\widehat=} \langle x_1 : t_1;\ \ldots;\ x_n : t_n \mid inv \rangle \\ Op \mathrel{\widehat=} \langle \Delta L \mid p \rangle \\ G \mathrel{\widehat=} \langle f : X \twoheadrightarrow L \rangle \end{array}$$

$$\begin{array}{|l} \multicolumn{1}{l}{\Phi} \\ \hline \Delta G \\ \Delta L \\ x? : X \\ \hline x? \in \operatorname{dom} f \\ \theta L = f\ x? \\ \{x?\} \mathbin{⩤} f' = \{x?\} \mathbin{⩤} f \\ f'\ x? = \theta L' \\ \hline \end{array}$$

Syntactic Restriction *pn*, *r*, and *f* are not free in *Op*.

Derivation This proof relies on the lemma below which defines the precondition of a promoted operation in terms of that of the local operation. This result was presented in [64], but here we express it in a slightly different way.

Lemma D.3 *If* L, Op, G, *and* Φ *are defined as above,*

$$\text{pre}\ (\exists \Delta L \bullet \Phi \wedge Op) = \langle G;\ x? : X \mid x? \in \operatorname{dom} f \wedge (\text{pre}\ Op)[(f\ x?).x_i / x_i] \rangle$$

Proof

$$\begin{array}{ll} \text{pre}\ (\exists \Delta L \bullet \Phi \wedge Op) & \\ = \exists L \bullet \text{pre}\ \Phi \wedge \text{pre}\ Op & \text{[by a theorem in [64, p.356]]} \end{array}$$

$= \exists L \bullet$ [by a property of pre]

$$\left\langle \begin{array}{l} G;\ L;\ x? : X \mid \\ \exists G';\ L' \bullet x? \in \operatorname{dom} f \land \theta L = f\ x? \land \{x?\} \lhd f' = \{x?\} \lhd f \land f'\ x? = \theta L' \end{array} \right\rangle \land$$

pre Op

$= \exists L \bullet \langle G;\ L;\ x? : X \mid x? \in \operatorname{dom} f \land \theta L = f\ x? \rangle \land \text{pre } Op$ [by predicate calculus]

$= \langle G;\ x? : X \mid \exists L \bullet x? \in \operatorname{dom} f \land \theta L = f\ x? \land \text{pre } Op \rangle$

[by properties of the schema calculus]

$= \langle G;\ x? : X \mid \exists r : L \bullet (x? \in \operatorname{dom} f \land \theta L = f\ x? \land \text{pre } Op)[r.x_i/x_i] \rangle$

[by a property of bindings]

$= \langle G;\ x? : X \mid \exists r : L \bullet (x? \in \operatorname{dom} f \land \langle\!| x_i := x_i |\!\rangle = f\ x? \land \text{pre } Op)[r.x_i/x_i] \rangle$

[by a property of θ]

$= \langle G;\ x? : X \mid \exists r : L \bullet x? \in \operatorname{dom} f \land \langle\!| x_i := r.x_i |\!\rangle = f\ x? \land (\text{pre } Op)[r.x_i/x_i] \rangle$

[by a property of substitution]

$= \langle G;\ x? : X \mid \exists r : L \bullet x? \in \operatorname{dom} f \land r = f\ x? \land (\text{pre } Op)[r.x_i/x_i] \rangle$

[by a property of bindings]

$= \langle G;\ x? : X \mid x? \in \operatorname{dom} f \land (\text{pre } Op)[(f\ x?).x_i/x_i] \rangle$ [by predicate calculus]

□

$\exists \Delta L \bullet \Phi \land Op$

= by properties of the schema calculus

$$\left\langle \begin{array}{l} \Delta G;\ x? : X \mid \\ \exists \Delta L \bullet x? \in \operatorname{dom} f \land \theta L = f\ x? \land \{x?\} \lhd f' = \{x?\} \lhd f \land f'\ x? = \theta L' \land Op \end{array} \right\rangle$$

$= bC$

$$f : \left[\begin{array}{l} x? \in \operatorname{dom} f \land (\text{pre } Op)[(f\ x?).x_i/x_i], \\ \exists \Delta L \bullet x? \in \operatorname{dom} f \land \theta L = f\ x? \land \{x?\} \lhd f' = \{x?\} \lhd f \land f'\ x? = \theta L' \land Op \end{array} \right]$$

$\sqsubseteq$ *procI*

$[\![$ **proc** $pn \mathrel{\widehat=} (\textbf{val-res}\ r : L \bullet (r, r' : L \mid (inv \land inv' \land p)[r.x_i, r'.x_i/x_i, x_i'])) \bullet$

$$f : \left[\begin{array}{l} x? \in \operatorname{dom} f \land (\text{pre } Op)[(f\ x?).x_i/x_i], \\ \exists \Delta L \bullet x? \in \operatorname{dom} f \land \theta L = f\ x? \land \{x?\} \lhd f' = \{x?\} \lhd f \land f'\ x? = \theta L' \land Op \end{array} \right] \lhd$$

$]\!]$

= by a property of bindings

$$f : \left[\begin{array}{l} x? \in \operatorname{dom} f \land (\text{pre } Op)[(f\ x?).x_i/x_i], \\ \exists r, r' : L \bullet \\ \qquad \left(\begin{array}{l} x? \in \operatorname{dom} f \land \theta L = f\ x? \\ \{x?\} \lhd f' = \{x?\} \lhd f \land f'\ x? = \theta L' \land Op \end{array} \right) [r.x_i, r'.x_i/x_i, x_i'] \end{array} \right]$$

= by a property of θ

$$f : \left[\begin{array}{l} x? \in \operatorname{dom} f \land (\operatorname{pre} Op)[(f\ x?).x_i/x_i], \\ \exists r, r' : L \bullet \\ \quad \left(\begin{array}{l} x? \in \operatorname{dom} f \land \lparen\!| x_i := x_i |\!\rparen = f\ x? \\ \{x?\} \lhd f' = \{x?\} \lhd f \land f'\ x? = \lparen\!| x_i := x_i' |\!\rparen \land Op \end{array} \right) [r.x_i, r'.x_i/x_i, x_i'] \end{array} \right]$$

= by a property of substitution

$$f : \left[\begin{array}{l} x? \in \operatorname{dom} f \land (\operatorname{pre} Op)[(f\ x?).x_i/x_i], \\ \left(\begin{array}{l} \exists r, r' : L \bullet \\ \quad x? \in \operatorname{dom} f \land \lparen\!| x_i := r.x_i |\!\rparen = f\ x? \land \\ \quad \{x?\} \lhd f' = \{x?\} \lhd f \land f'\ x? = \lparen\!| x_i := r'.x_i |\!\rparen \land Op[r.x_i, r'.x_i/x_i, x_i'] \end{array} \right) \end{array} \right]$$

= by a property of bindings

$$f : \left[\begin{array}{l} x? \in \operatorname{dom} f \land (\operatorname{pre} Op)[(f\ x?).x_i/x_i], \\ \left(\begin{array}{l} \exists r, r' : L \bullet \\ \quad x? \in \operatorname{dom} f \land r = f\ x? \land \\ \quad \{x?\} \lhd f' = \{x?\} \lhd f \land f'\ x? = r' \land Op[r.x_i, r'.x_i/x_i, x_i'] \end{array} \right) \end{array} \right]$$

= by predicate calculus

$$f : \left[\begin{array}{l} x? \in \operatorname{dom} f \land (\operatorname{pre} Op)[(f\ x?).x_i/x_i], \\ x? \in \operatorname{dom} f \land \{x?\} \lhd f' = \{x?\} \lhd f \land Op[r.x_i, r'.x_i/x_i, x_i'][f\ x?, f'\ x?/r, r'] \end{array} \right]$$

$\sqsubseteq sP$

$$f : \left[\begin{array}{l} x? \in \operatorname{dom} f \land (\operatorname{pre} Op)[(f\ x?).x_i/x_i], \\ \{x?\} \lhd f' = \{x?\} \lhd f \land Op[r.x_i, r'.x_i/x_i, x_i'][f\ x?, f'\ x?/r, r'] \end{array} \right]$$

= by r is not free in pre Op

$$f : \left[\begin{array}{l} (x? \in \operatorname{dom} f \land \operatorname{pre} Op)[r.x_i/x_i][f\ x?/r], \\ \{x?\} \lhd f' = \{x?\} \lhd f \land Op[r.x_i, r'.x_i/x_i, x_i'][f\ x?, f'\ x?/r, r'] \end{array} \right]$$

$\sqsubseteq vrS$

$$(\textbf{val-res}\ r : L \bullet r : [(x? \in \operatorname{dom} f \land \operatorname{pre} Op)[r.x_i/x_i], Op[r.x_i, r'.x_i/x_i, x_i']\ ])(f\ x?)$$

The body of the above parametrised statement can be refined as follows..

$$r : [(x? \in \operatorname{dom} f \land \operatorname{pre} Op)[r.x_i/x_i], Op[r.x_i, r'.x_i/x_i, x_i']\ ]$$

$\sqsubseteq wP$

$$r : [(\operatorname{pre} Op)[r.x_i/x_i], Op[r.x_i, r'.x_i/x_i, x_i']\ ]$$

= by definition of Op

$$r : [(inv \land \exists x_1' : t_1; \ldots; x_n' : t_n \bullet inv' \land p)[r.x_i/x_i], (inv \land inv' \land p)[r.x_i, r'.x_i/x_i, x_i']\ ]$$

= by a property of substitution and bindings

$$r : \left[\begin{array}{l} inv[r.x_i/x_i] \land \exists r' : L \bullet inv'[r.x_i/x_i'] \land p[r.x_i, r'.x_i/x_i, x_i'], \\ (inv \land inv' \land p)[r.x_i, r'.x_i/x_i, x_i']\ ] \end{array} \right]$$

$$\begin{array}{l}
\sqsubseteq sP \\
\quad r : \left[\begin{array}{l} inv[r.x_i/x_i] \land \exists r' : L \bullet inv'[r'.x_i/x_i'] \land p[r.x_i, r'.x_i/x_i, x_i'], \\ inv'[r'.x_i/x_i'] \land p[r.x_i, r'.x_i/x_i, x_i'] \end{array} \right] \\
= bC \\
\quad (r, r' : L \mid (inv \land inv' \land p)[r.x_i, r'.x_i/x_i, x_i'])
\end{array}$$

At this point we can apply *pcallI* to the procedure block and transform the parametrised statement into a call to *pn*.

□

D.2 Refinement Laws

In this section we present the ZRC refinement laws in alphabetical order, and the data refinement laws.

Law *abA* Absorb assumption

$$\begin{array}{ll}
 & \{pre_1\}\ w : [pre_2, post] \\
= & abA \\
 & w : [pre_1 \land pre_2, post]
\end{array}$$

Derivation

$$\begin{array}{lr}
wp.\{pre_1\}\ w : [pre_2, post].\psi & \\
\equiv pre_1 \land pre_2 \land (\forall dw' \bullet post \Rightarrow \psi)[\_/'] & \text{[by definition of } wp\text{]} \\
\equiv wp.w : [pre_1 \land pre_2, post].\psi & \text{[by definition of } wp\text{]}
\end{array}$$

□

Law *abC* Absorb coercion

$$\begin{array}{ll}
 & w : [pre, post_1]\ ;\ [post_2] \\
= & abC \\
 & w : [pre, post_1 \land post_2']
\end{array}$$

Derivation

$$\begin{array}{lr}
wp.w : [pre, post_1]\ ;\ [post_2].\psi' & \\
\equiv pre \land (\forall dw' \bullet post_1 \Rightarrow (post_2[\_/'] \Rightarrow \psi)')[\_/'] & \text{[by definition of } wp\text{]} \\
\equiv pre \land (\forall dw' \bullet post_1 \Rightarrow (post_2' \Rightarrow \psi'))[\_/'] & \text{[by a property of substitution]} \\
\equiv pre \land (\forall dw' \bullet post_1 \land post_2' \Rightarrow \psi')[\_/'] & \text{[by predicate calculus]} \\
\equiv wp.w : [pre, post_1 \land post_2'].\psi' & \text{[by definition of } wp\text{]}
\end{array}$$

□

Law *altI* Alternation introduction

$w : [pre, post]$

$\sqsubseteq$ *altI*

if $[] \, i \bullet g_i \rightarrow w : [g_i \land pre, post]$ **fi**

provided $pre \Rightarrow (\bigvee i \bullet g_i)$

Syntactic Restrictions

- Each g_i is a well-scoped predicate;
- No g_i has free dashed variables;
- $\{ \; i \bullet g_i \; \}$ is non-empty.

Derivation

$wp.w : [pre, post].\psi$

$\equiv pre \land (\forall dw' \bullet post \Rightarrow \psi)[\_/']$ [by definition of wp]

$\equiv (\bigvee i \bullet g_i) \land pre \land (\forall dw' \bullet post \Rightarrow \psi)[\_/']$ [by the proviso]

$\Rightarrow (\bigvee i \bullet g_i) \land (\bigwedge i \bullet g_i \Rightarrow g_i \land pre \land (\forall dw' \bullet post \Rightarrow \psi)[\_/'])$ [by predicate calculus]

$\equiv wp.\mathbf{if} \, [] \, i \bullet g_i \rightarrow w : [g_i \land pre, post] \; \mathbf{fi}.\psi$ [by definition of wp]

□

Law *assigI* Assignment introduction

$w, vl : [pre, post]$

$\sqsubseteq$ *assigI*

$vl := el$

provided $pre \Rightarrow post[el/vl'][\_/']$

Syntactic Restrictions

- vl contains no duplicated variables;
- vl and el have the same length;
- el is well-scoped and well-typed;
- el has no free dashed variables;
- The corresponding variables of vl and expressions of el have the same type.

Derivation

$wp.w, vl : [pre, post].\psi'$

$\equiv pre \land (\forall dw'; \; dvl' \bullet post \Rightarrow \psi')[\_/']$ [by definition of wp]

$\Rightarrow post[el/vl'][\_/'] \land (\forall dw'; \; dvl' \bullet post \Rightarrow \psi')[\_/']$ [by the proviso]

$\Rightarrow post[el/vl'][\_/'] \land (post[el/vl'] \Rightarrow \psi'[el/vl'])[\_/']$ [by predicate calculus]

$\equiv post[el/vl'][\_/'] \land (post[el/vl'][\_/'] \Rightarrow \psi'[el/vl'][\_/'])$ [by a property of substitution]

$\Rightarrow \psi'[el/vl'][\_/']$ [by predicate calculus]

$\equiv \psi[el/vl]$ [by a property of substitution]

$\equiv wp.vl := el.\psi'$ [by definition of wp]

□

Law *assumP* Assumption

$\{pre\}$

$=$ *assumP*

$: [pre, \mathrm{true}]$

Derivation

$wp.\{pre\}.\psi'$

$\equiv pre \land \psi$ [by definition of wp]

$\equiv pre \land (\mathrm{true} \Rightarrow \psi)[\_/']$ [by predicate calculus and no dashed variable is free in ψ]

$\equiv pre \land (\mathrm{true} \Rightarrow \psi')[\_/']$ [by a property of substitution]

$\equiv wp. : [pre, \mathrm{true}].\psi'$ [by definition of wp]

□

Law *assumpI* Assumption introduction

$[post]$

$=$ *assumpI*

$[post]\{post[\_/']\}$

Derivation

$wp.[post].\psi'$

$\equiv post[\_/'] \Rightarrow \psi$ [by definition of wp]

$\equiv post[\_/'] \Rightarrow post[\_/'] \land \psi$ [by predicate calculus]

$\equiv wp.[post]\{post[\_/']\}.\psi'$ [by definition of wp]

□

Law *assumpR* Assumption removal

$\{pre\}$

$\sqsubseteq$ *assumpR*

skip

Derivation

$wp.\{pre\}.\psi'$

$\equiv pre \wedge \psi$ [by definition of wp]

$\Rightarrow \psi$ [by predicate calculus]

$\equiv wp.\mathbf{skip}.\psi'$ [by definition of wp]

□

Law *cfR* Contract frame

$w, x : [pre, post]$

$\sqsubseteq$ *cfR*

$x : [pre, post[w/w']]$

Syntactic Restriction The variables of w are not in x.

Derivation

$wp.w, x : [pre, post].\psi$

$\equiv pre \wedge (\forall\, dw';\ dx' \bullet post \Rightarrow \psi)[\_/']$ [by definition of wp]

$\Rightarrow pre \wedge (\forall\, dx' \bullet post \Rightarrow \psi)[\_/']$ [by predicate calculus]

$\equiv pre \wedge (\forall\, dx' \bullet post[w/w'] \Rightarrow \psi)[\_/']$ [by w are not in x]

$\equiv wp.x : [pre, post[w/w']].\psi$ [by definition of wp]

□

Law *cO* Coercion

$[post]$

$=$ *cO*

$: [\mathrm{true}, post]$

Derivation

$wp.[post].\psi'$

$\equiv post[\_/'] \Rightarrow \psi$ [by definition of wp]

$\equiv (post \Rightarrow \psi)'[\_/']$ [by a property of substitution]

$\equiv (post \Rightarrow \psi')[\_/']$ [by a property of substitution]

$\equiv wp. : [\mathrm{true}, post].\psi'$ [by definition of wp]

□

Law *coI* Coercion introduction

skip

$\sqsubseteq$ *coI*

$[post]$

Derivation

$$\begin{array}{ll}
wp.\mathbf{skip}.\psi' & \\
\equiv \psi & \text{[by definition of } wp\text{]} \\
\Rightarrow post[\_/'] \Rightarrow \psi & \text{[by predicate calculus]} \\
\equiv wp.[post].\psi' & \text{[by definition of } wp\text{]}
\end{array}$$

□

Law *conI* Constant introduction

$$\begin{array}{ll}
 & p \\
= & conI \\
 & |[\,\mathbf{con}\ dcl \bullet \{init\}\ p\,]|
\end{array}$$

provided $\exists\, dcl \bullet init$.

Syntactic Restrictions

- *init* is well-scoped and well-typed;
- *init* has no free dashed variables;
- The constants declared by dcl and their dashed counterparts are not free in p.

Derivation

$$\begin{array}{ll}
wp.p.\psi & \\
\equiv (\exists\, dcl \bullet init) \wedge wp.p.\psi & \text{[by the proviso]} \\
\equiv \exists\, dcl \bullet init \wedge wp.p.\psi & \text{[by } \alpha dcl \text{ and } \alpha dcl' \text{ are not free in } p \text{ and } \psi\text{]} \\
\equiv wp.|[\,\mathbf{con}\ dcl \bullet \{init\}\ p\,]|\,.\psi & \text{[by definition of } wp\text{]}
\end{array}$$

□

Law *conI* Constant introduction (specification statement)

$$\begin{array}{ll}
 & w : [pre, post] \\
\sqsubseteq & conI \\
 & |[\,\mathbf{con}\ dcl \bullet w : [npre, post]\ ]|
\end{array}$$

provided $pre \Rightarrow \exists\, dcl \bullet npre$

Syntactic Restrictions

- *npre* is well-scoped and well-typed;
- *npre* has no free dashed variables;
- The constants declared by dcl and their dashed counterparts are not free in $w : [pre, post]$.

Derivation

$wp.w : [pre, post].\psi$

$\equiv pre \wedge (\forall dw' \bullet post \Rightarrow \psi)[\_/']$ [by definition of wp]

$\Rightarrow (\exists dcl \bullet npre) \wedge (\forall dw' \bullet post \Rightarrow \psi)[\_/']$ [by the proviso]

$\equiv \exists dcl \bullet npre \wedge (\forall dw' \bullet post \Rightarrow \psi)[\_/']$ [by αdcl and $\alpha dcl'$ are not free in dw', $post$, and ψ]

$\equiv wp.|[\ \mathbf{con}\ dcl \bullet w : [npre, post]\ ]|\ .\psi$ [by definition of wp]

□

Law *conR* Constant removal

$|[\ \mathbf{con}\ dcl \bullet p\ ]|$

$=$ *conR*

p

Syntactic Restriction The constants declared by *dcl* are not free in *p*.

Derivation

$wp.|[\ \mathbf{con}\ dcl \bullet p\ ]|\ .\psi$

$\equiv \exists dcl \bullet wp.p.\psi$ [by definition of wp]

$\equiv wp.p.\psi$ [by αdcl and $\alpha dcl'$ are not free in p and ψ]

□

Law *coR* Coercion removal

$\{pre\}[pre']$

$=$ *coR*

$\{pre\}$

Derivation

$wp.\{pre\}[pre'].\psi'$

$\equiv pre \wedge (pre'[\_/'] \Rightarrow \psi)$ [by definition of wp]

$\equiv pre \wedge (pre \Rightarrow \psi)$ [by a property of substitution]

$\equiv pre \wedge \psi$ [by predicate calculus]

$\equiv wp.\{pre\}.\psi'$ [by definition of wp]

□

Law *dimG* Diminish guards

$$\begin{array}{ll} & \textbf{if} \; [] \; i \bullet g_i \rightarrow p_i \; \textbf{fi} \\ \sqsubseteq & dimG \\ & \textbf{if} \; [] \; i \bullet h_i \rightarrow p_i \; \textbf{fi} \end{array}$$

provided

- $(\bigvee i \bullet g_i) \Rightarrow (\bigvee i \bullet h_i)$
- $(\bigvee i \bullet g_i) \Rightarrow (h_i \Rightarrow g_i)$, for each i.

Syntactic Restrictions

- Each h_i is a well-scoped and well-typed predicate;
- No h_i has free dashed variables;
- $\{ \; i \bullet g_i \; \}$ and $\{ \; i \bullet h_i \; \}$ have the same number of elements.

Derivation

$$\begin{array}{lr} wp.\text{if} \; [] \; i \bullet g_i \rightarrow p_i \; \text{fi}.\psi & \\ \equiv (\bigvee i \bullet g_i) \wedge (\bigwedge \bullet g_i \Rightarrow wp.p_i.\psi) & \text{[by definition of } wp\text{]} \\ \Rightarrow (\bigvee i \bullet g_i) \wedge (\bigwedge \bullet h_i \Rightarrow wp.p_i.\psi) & \text{[by the second proviso]} \\ \Rightarrow (\bigvee i \bullet h_i) \wedge (\bigwedge \bullet h_i \Rightarrow wp.p_i.\psi) & \text{[by the first proviso]} \\ \equiv wp.\text{if} \; [] \; i \bullet h_i \rightarrow p_i \; \text{fi}.\psi & \text{[by definition of } wp\text{]} \end{array}$$

□

Law *dR* Data refinement (restricted)

$$\begin{array}{ll} & [\![\; \textbf{var} \; dvl; \; davl \bullet p_1 \;]\!] \\ \sqsubseteq & dR \\ & [\![\; \textbf{var} \; dvl; \; dcvl \bullet p_2 \;]\!] \end{array}$$

provided

- $p_1 \preccurlyeq p_2$
- $\forall \, dcvl \bullet \exists \, davl \bullet ci$

where *davl* and *dcvl* declare the variables of *avl* (the abstract variables) and *cvl* (the concrete variables); and *ci* is the coupling invariant.

Syntactic Restrictions

- The variables of *cvl* and *cvl′* are not free in p_1, and are not in *avl*;
- The variables of *avl* and *avl′* are not free in p_2;
- *ci* is a well-scoped and well-typed predicate.

Derivation

$wp.\ |[\ \mathbf{var}\ dvl;\ davl \bullet p_1\ ]|\ .\psi$

$\equiv \forall\, dvl;\ davl \bullet wp.p_1.\psi$ [by definition of wp]

$\equiv \forall\, dvl \bullet (\forall\, dcvl \bullet \exists\, davl \bullet ci) \land (\forall\, davl \bullet wp.p_1.\psi)$ [by the second proviso]

$\equiv \forall\, dvl;\ dcvl \bullet (\exists\, davl \bullet ci) \land (\forall\, davl \bullet wp.p_1.\psi)$

[by cvl and cvl' are not free in $davl$, p_1, and ψ]

$\equiv \forall\, dvl;\ dcvl \bullet \exists\, davl \bullet ci \land \forall\, davl \bullet wp.p_1.\psi$ [by avl are not free in $davl$]

$\Rightarrow \forall\, dvl;\ dcvl \bullet \exists\, davl \bullet ci \land wp.p_1.\psi$ [by predicate calculus]

$\Rightarrow \forall\, dvl;\ dcvl \bullet wp.p_2.\exists\, davl' \bullet ci' \land \psi$ [by the first proviso]

$\equiv \forall\, dvl;\ dcvl \bullet wp.p_2.((\exists\, davl' \bullet ci') \land \psi)$ [by avl' are not free in ψ]

$\Rightarrow \forall\, dvl;\ dcvl \bullet wp.p_2.\psi$ [by monotonicity of wp]

$\equiv wp.\ |[\ \mathbf{var}\ dvl;\ dcvl \bullet p_2\ ]|\ .\psi$ [by definition of wp]

□

Law *dR* Data refinement (variable blocks with initialisation)

$|[\ \mathbf{var}\ dvl;\ davl \bullet avl : [true, init']\ ;\ p_1\ ]|$

$\sqsubseteq$ *dR*

$|[\ \mathbf{var}\ dvl;\ dcvl \bullet cvl : [\mathrm{true}, (\exists\, davl \bullet ci \land init)']\ ;\ p_2\ ]|$

provided $p_1 \preccurlyeq p_2$

where *davl* and *dcvl* declare the variables of *avl* (the abstract variables) and *cvl* (the concrete variables); and *ci* is the coupling invariant.

Syntactic Restrictions

- The variables of *cvl* and *cvl'* are not free in *init* and p_1, and are not in *avl*;
- The variables of *avl* and *avl'* are not free in p_2;
- *ci* is a well-scoped and well-typed predicate.

Derivation

$wp.\ |[\ \mathbf{var}\ dvl;\ davl \bullet avl : [\mathrm{true}, init']\ ;\ p_1\ ]|\ .\psi$

$\equiv \forall\, dvl;\ davl \bullet (\forall\, davl' \bullet init' \Rightarrow (wp.p_1.\psi)')[\_/']$ [by definition of wp]

$\equiv \forall\, dvl;\ davl \bullet (\forall\, davl \bullet init \Rightarrow (wp.p_1.\psi))'[\_/']$ [by program variables are not free in $davl'$]

$\equiv \forall\, dvl;\ davl \bullet init \Rightarrow wp.p_1.\psi$ [by a property of substitution]

$\equiv \forall\, dvl;\ dcvl;\ davl \bullet init \Rightarrow wp.p_1.\psi$

[by cvl are not free in $davl$, $init$, p_1, and ψ, and cvl' are not free in p_1 and ψ]

$\Rightarrow \forall\, dvl;\ dcvl;\ davl \bullet ci \land init \Rightarrow ci \land wp.p_1.\psi$ [by predicate calculus]

$\Rightarrow \forall\, dvl;\ dcvl;\ davl \bullet ci \land init \Rightarrow \exists\, davl \bullet ci \land wp.p_1.\psi$ [by predicate calculus]

$\equiv \forall\, dvl;\ dcvl \bullet (\exists\, davl \bullet ci \land init) \Rightarrow \exists\, davl \bullet ci \land wp.p_1.\psi$ [by *avl* are not free in *davl*]

$\Rightarrow \forall\, dvl;\ dcvl \bullet (\exists\, davl \bullet ci \land init) \Rightarrow wp.p_2.\exists\, davl' \bullet ci' \land \psi$ [by the proviso]

$\equiv \forall\, dvl;\ dcvl \bullet (\exists\, davl \bullet ci \land init) \Rightarrow wp.p_2.(\exists\, davl' \bullet ci') \land \psi$ [by *avl'* are not free in ψ]

$\Rightarrow \forall\, dvl;\ dcvl \bullet (\exists\, davl \bullet ci \land init) \Rightarrow wp.p_2.\psi$ [by monotonicity of *wp*]

$\equiv \forall\, dvl;\ dcvl \bullet (\forall\, dcvl \bullet (\exists\, davl \bullet ci \land init) \Rightarrow wp.p_2.\psi)$ [by predicate calculus]

$\equiv \forall\, dvl;\ dcvl \bullet (\forall\, dcvl \bullet (\exists\, davl \bullet ci \land init) \Rightarrow wp.p_2.\psi)'[\_/']$ [by a property of substitution]

$\equiv \forall\, dvl;\ dcvl \bullet (\forall\, dcvl' \bullet (\exists\, davl \bullet ci \land init)[cvl'/cvl] \Rightarrow (wp.p_2.\psi)[cvl'/cvl])'[\_/']$

[by *cvl'* are not free in *davl*, *ci*, and *init*]

$\equiv \forall\, dvl;\ dcvl \bullet (\forall\, dcvl' \bullet (\exists\, davl \bullet ci \land init)' \Rightarrow (wp.p_2.\psi)')[\_/']$

[by a property of substitution]

$\equiv wp.\ |[\ \mathbf{var}\ dvl;\ dcvl \bullet cvl : [\text{true}, (\exists\, davl \bullet ci \land init)']\ ;\ p_2\ ]|\ .\psi$ [by definition of *wp*]

□

Law *dR* Data refinement

$|[\ \mathbf{var}\ dvl;\ davl \bullet p_1\ ]|$

$\sqsubseteq\ dR$

$|[\ \mathbf{var}\ dvl;\ dcvl \bullet cvl : [\text{true}, (\exists\, davl \bullet ci)']\ ;\ p_2\ ]|$

provided $p_1 \preccurlyeq p_2$

where *davl* and *dcvl* declare the variables of *avl* (the abstract variables) and *cvl* (the concrete variables); and *ci* is the coupling invariant.

Syntactic Restrictions

- The variables of *cvl* and *cvl'* are uot free in p_1, and are not in *avl*;
- The variables of *avl* and *avl'* are not free in p_2;
- *ci* is a well-scoped and well-typed predicate.

Derivation

$wp.\ |[\ \mathbf{var}\ dvl;\ davl \bullet p_1\ ]|\ .\psi$

$\equiv \forall\, dvl;\ davl \bullet wp.p_1.\psi$ [by definition of *wp*]

$\equiv \forall\, dvl;\ dcvl;\ davl \bullet wp.p_1.\psi$ [by *cvl* and *cvl'* are not free in p_1 and ψ]

$\Rightarrow \forall\, dvl;\ dcvl;\ davl \bullet ci \Rightarrow ci \land wp.p_1.\psi$ [by predicate calculus]

$\Rightarrow \forall\, dvl;\ dcvl;\ davl \bullet ci \Rightarrow \exists\, davl \bullet ci \land wp.p_1.\psi$ [by predicate calculus]

$\equiv \forall\, dvl;\ dcvl \bullet (\exists\, davl \bullet ci) \Rightarrow \exists\, davl \bullet ci \land wp.p_1.\psi$ [by *avl* are not free in *davl*]

$\Rightarrow \forall\, dvl;\ dcvl \bullet (\exists\, davl \bullet ci) \Rightarrow wp.p_2.\exists\, davl' \bullet ci' \wedge \psi$ [by the proviso]

$\equiv \forall\, dvl;\ dcvl \bullet (\exists\, davl \bullet ci) \Rightarrow wp.p_2.(\exists\, davl' \bullet ci') \wedge \psi$ [by avl' are not free in ψ]

$\Rightarrow \forall\, dvl;\ dcvl \bullet (\exists\, davl \bullet ci) \Rightarrow wp.p_2.\psi$ [by monotonicity of wp]

$\equiv \forall\, dvl;\ dcvl \bullet (\forall\, dcvl \bullet (\exists\, davl \bullet ci) \Rightarrow wp.p_2.\psi)$ [by predicate calculus]

$\equiv \forall\, dvl;\ dcvl \bullet (\forall\, dcvl \bullet (\exists\, davl \bullet ci) \Rightarrow wp.p_2.\psi)'[\_/']$ [by a property of substitution]

$\equiv \forall\, dvl;\ dcvl \bullet (\forall\, dcvl' \bullet (\exists\, davl \bullet ci)[cvl'/cvl] \Rightarrow (wp.p_2.\psi)[cvl'/cvl])'[\_/']$
[by cvl' are not free in $davl$ and ci]

$\equiv \forall\, dvl;\ dcvl \bullet (\forall\, dcvl' \bullet (\exists\, davl \bullet ci)' \Rightarrow (wp.p_2.\psi)')[\_/']$ [by a property of substitution]

$\equiv wp.\ |[\ \mathbf{var}\ dvl;\ dcvl \bullet cvl : [\mathrm{true}, (\exists\, davl \bullet ci)']\ ;\ p_2\ ]|\ .\psi$ [by definition of wp]

□

Law *efR* Expand frame

$$\begin{array}{ll} & w : [pre, post] \\ = & efR \\ & w, x : [pre, post \wedge x' = x] \end{array}$$

Syntactic Restriction The variables of x are in scope, are not in w, and are not dashed.

Derivation

$wp.w : [pre, post].\psi$

$\equiv pre \wedge (\forall\, dw' \bullet post \Rightarrow \psi)[\_/']$ [by definition of wp]

$\equiv pre \wedge (\forall\, dw' \bullet post \Rightarrow \psi)[x/x'][\_/']$ [by a property of substitution]

$\equiv pre \wedge (\forall\, dw';\ dx' \bullet x' = x \wedge post \Rightarrow \psi)[\_/']$ [by x are not in w]

$\equiv wp.w, x : [pre, post \wedge x' = x].\psi$ [by definition of wp]

□

Law *esA* Establish assumption

$$\begin{array}{ll} & w : [pre_1, post]\ ;\ \{pre_2\} \\ = & esA \\ & w : [pre_1 \wedge (\forall\, dw' \bullet post \Rightarrow pre_2')[\_/'], post] \end{array}$$

where dw declares the variables of w.

Derivation

$wp.w : [pre_1, post]\ ;\ \{pre_2\}.\psi'$

$\equiv pre_1 \wedge (\forall\, dw' \bullet post \Rightarrow (pre_2 \wedge \psi)')[\_/']$ [by definition of wp]

$\equiv pre_1 \wedge (\forall\, dw' \bullet (post \Rightarrow pre_2') \wedge (post \Rightarrow \psi'))[\_/']$ [by predicate calculus]

$\equiv pre_1 \land ((\forall\, dw' \bullet post \Rightarrow pre_2') \land (\forall\, dw' \bullet post \Rightarrow \psi'))[\_/']$ [by predicate calculus]

$\equiv pre_1 \land (\forall\, dw' \bullet post \Rightarrow pre_2')[\_/'] \land (\forall\, dw' \bullet post \Rightarrow \psi')[\_/']$ [by a property of substitution]

$\equiv wp.w : [pre_1 \land (\forall\, dw' \bullet post \Rightarrow pre_2')[\_/'], post].\psi'$ [by definition of wp]

□

Law *fassigI* Following assignment introduction

$w, vl : [pre, post]$

$\sqsubseteq$ *fassigI*

$w, vl : [pre, post[el[w', vl'/w, vl]/vl']\,] \;;\; vl := el$

Syntactic Restrictions

- vl contains no duplicated variables;
- vl and el have the same length;
- el is well-scoped and well-typed;
- el has no free dashed variables;
- The corresponding variables of vl and expressions of el have the same type.

Derivation

$w, vl : [pre, post]$

$\sqsubseteq$ *seqcI*

$|[\,\mathbf{con}\ dcl \bullet$

$\quad w, vl : [pre, post[el[w', vl'/w, vl]/vl']\,] \;;$

$\quad w, vl : [post[el[w', vl'/w, vl]/vl'][cl/w, vl][\_/'], post[cl/w, vl]\,]$ $\triangleleft$

$]|$

$\sqsubseteq$ *assigI*

$vl := el$

The application of *assigI* generates the proof-obligation that we discharge below.

$post[el[w', vl'/w, vl]/vl'][cl/w, vl][\_/']$

$\equiv post[cl/w, vl][el[w', vl'/w, vl]/vl'][\_/']$ [by a property of substitution]

$\equiv post[cl/w, vl][el/vl'][\_/']$ [by a property of substitution]

As the constants of cl are not free in $w, vl : [pre, post[el[w', vl'/w, vl]/vl']\,]$ and $vl := el$, we can remove the constant block by applying *conR*.

□

Law *fiV* Fix initial value

$$
\begin{array}{ll}
& w : [pre, post] \\
\sqsubseteq & \mathit{fiV} \\
& [\![\, \mathbf{con}\ dcl \bullet w : [pre \wedge c_1 = e_1 \wedge \ldots \wedge c_n = e_n, post] \,]\!]
\end{array}
$$

where $c_1, \ldots, c_n$ are the constants declared by *dcl*.

Syntactic Restrictions

- The expressions of $e_1, \ldots, e_n$ are well-scoped and well-typed;
- The expressions of $e_1, \ldots, e_n$ have no free dashed variables;
- The corresponding constants of $c_1, \ldots, c_n$ and expressions of $e_1, \ldots, e_n$ have the same type;
- $c_1, \ldots, c_n$ and $c'_1, \ldots, c'_n$ are not free in $w : [pre, post]$ and in the corresponding expressions of $e_1, \ldots, c_n$.

Derivation Direct application of *conI*. The generated proof-obligation can be discharged as follows.

$$
\begin{array}{lr}
pre & \\
\equiv (\exists\, dcl \bullet c_1 = e_1 \wedge \ldots \wedge c_n = e_n) \wedge pre & \text{[by predicate calculus]} \\
\equiv \exists\, dcl \bullet c_1 = e_1 \wedge \ldots \wedge c_n = e_n \wedge pre & \text{[by } c_1, \ldots, c_n \text{ are not free in } pre\text{]}
\end{array}
$$

□

Law *itI* Iteration introduction

$$
\begin{array}{ll}
& w : [inv, inv[w'/w] \wedge \neg\, (\bigvee i \bullet g_i[w'/w])] \\
\sqsubseteq & \mathit{itI} \\
& \mathbf{do}\ [\!]\, i \bullet g_i \rightarrow w : [inv \wedge g_i, inv[w'/w] \wedge 0 \le vrt[w'/w] < vrt]\ \mathbf{od}
\end{array}
$$

Syntactic Restrictions

- *vrt* is a well-scoped and well-typed integer;
- Each g_i and *vrt* have no free dashed variables. expression.

Derivation

$$
\begin{array}{l}
\quad w : [inv, inv[w'/w] \wedge \neg\, (\bigvee i \bullet g_i[w'/w])] \\
\sqsubseteq \mathit{vrtI} \\
\quad [\![\mathbf{proc}\ it \mathrel{\widehat=} \{n = vrt\}\ w : [inv, inv[w'/w] \wedge \neg\, (\bigvee i \bullet g_i[w'/w])]\ \mathbf{variant}\ n\ \mathbf{is}\ vrt \bullet \\
\qquad w : [inv, inv[w'/w] \wedge \neg\, (\bigvee i \bullet g_i[w'/w])] \\
\quad]\!] \\
\sqsubseteq \mathit{pcallI} \\
\quad [\![\mathbf{proc}\ it \mathrel{\widehat=} \{n = vrt\}\ w : [inv, inv[w'/w] \wedge \neg\, (\bigvee i \bullet g_i[w'/w])]\ \mathbf{variant}\ n\ \mathbf{is}\ vrt \bullet it\,]\!]
\end{array}
$$

The procedure body can be refined as follows.

$\sqsubseteq abA$

$w : [n = vrt \wedge inv, inv[w'/w] \wedge \neg (\bigvee i \bullet g_i[w'/w])]$

$\sqsubseteq altI$

$$\begin{array}{lll} \mathbf{if} \; [] \; i \bullet g_i \rightarrow & w : [g_i \wedge n = vrt \wedge inv, inv[w'/w] \wedge \neg (\bigvee i \bullet g_i[w'/w])] & \lhd \\ [] \;\; \neg (\bigvee i \bullet g_i) \rightarrow & w : [\neg (\bigvee i \bullet g_i) \wedge n = vrt \wedge inv, inv[w'/w] \wedge \neg (\bigvee i \bullet g_i[w'/w])] & (i) \\ \mathbf{fi} & & \end{array}$$

$\sqsubseteq seqcI$

$|[\; \mathbf{con}\; dcl \bullet$

$$w : [g_i \wedge n = vrt \wedge inv, n = vrt \wedge inv[w'/w] \wedge 0 \leq vrt[w'/w] < vrt] \; ; \quad \lhd$$

$$w : \left[\begin{array}{l} (n = vrt \wedge inv[w'/w] \wedge 0 \leq vrt[w'/w] < vrt)[cl/w][\_/'], \\ (inv[w'/w] \wedge \neg (\bigvee i \bullet g_i[w'/w]))[cl/w] \end{array} \right] \quad (ii)$$

$]|$

$\sqsubseteq sP$

$w : [g_i \wedge n = vrt \wedge inv, inv[w'/w] \wedge 0 \leq vrt[w'/w] < vrt]$

$\sqsubseteq wP$

$w : [inv \wedge g_i, inv[w'/w] \wedge 0 \leq vrt[w'/w] < vrt]$

The refinement of the specification statement (ii) is as follows.

$(ii) \sqsubseteq sP$

$$w : \left[\begin{array}{l} (n = vrt \wedge inv[w'/w] \wedge 0 \leq vrt[w'/w] < vrt)[cl/w][\_/'], \\ inv[w'/w] \wedge \neg (\bigvee i \bullet g_i[w'/w]) \end{array} \right]$$

$\sqsubseteq wP$

$w : [0 \leq vrt < n \wedge inv, inv[w'/w] \wedge \neg (\bigvee i \bullet g_i[w'/w])]$

$= abA$

$\{0 \leq vrt < n\} \; w : [inv, inv[w'/w] \wedge \neg (\bigvee i \bullet g_i[w'/w])]$

At this point, we can apply *pcallI* to the variant block to replace the above program by a recursive call to *it* (and the variant block by a procedure block). Afterwards, we can apply *conR* to remove the constant block. The refinement of (i) proceeds as shown below.

$(i) \sqsubseteq skI$

skip

The resulting program is presented below.

```
|[ proc it ≙ if [] i • g_i → w : [inv ∧ g_i, inv[w'/w] ∧ 0 ≤ vrt[w'/w] < vrt] ; it
                [] ¬ (∨ i • g_i) → skip
             fi •
     it
]|
```

This procedure block, by definition, is equal to the iteration below.

$$\mathbf{do} \, [] \, \imath \bullet g_i \to w : [inv \wedge g_i, inv[w'/w] \wedge 0 \leq vrt[w'/w] < vrt] \, \mathbf{od}$$

□

Law *lassigI* Leading assigument introduction

$w, vl : [pre[el/vl], post[el/vl]\ ]$

$\sqsubseteq$ *lassigI*

$vl := el \ ; \ w, vl : [pre, post]$

Syntactic Restrictions

- vl contains uo duplicated variables;
- vl and el have the same length;
- el is well-scoped and well-typed;
- el has no free dashed variables;
- The corresponding variables of vl and expressions of el have the same type.

Derivation

$w, vl : [pre[el/vl], post[el/vl]\ ]$

$\sqsubseteq$ *seqcI*

$[\![\, \mathbf{con} \ dcl \bullet$

$vl : [pre[el/vl], pre' \wedge vl' = el]$ ◁

$w, vl : [(pre' \wedge vl' = el)[cl/vl][\_/'], post[el/vl][cl/vl]\ ]$ $(\imath)$

$]\!]$

$\sqsubseteq$ *assigI*

$vl := el$

The application of *assigI* generates the proof-obligation that we discharge below.

$pre[el/vl]$

$\equiv pre'[el/vl'][\_/']$ [by a property of substitution]

$\equiv pre'[el/vl'][\_/'] \wedge el = el$ [by predicate calculus]

$\equiv (pre' \wedge vl' = el)[el/vl'][\_/']$ [by a property of substitution]

We proceed with the refinement as follows.

$(\imath) \sqsubseteq sP$

$w, vl : [(pre' \wedge vl' = el)[cl/vl][\_/'], post]$

We discharge the proof-obligation that arises from the application of *sP* as follows.

$(pre' \land vl' = el)[cl/vl][\_/'] \land post$
$\equiv pre \land vl = el[cl/vl] \land post$ [by a property of substitution]
$\equiv pre \land vl = el[cl/vl] \land post[el[cl/vl]/vl]$ [by predicate calculus]
$\Rightarrow post[el/vl][cl/vl]$ [by predicate calculus]

We finish the refinement with an application of *wP*.

$\sqsubseteq wP$
$w, vl : [pre, post]$

The proof-obligation generated is trivial.

□

Law *mA* Merge annotations (assumptions)

$\{pre_1\}\{pre_2\}$
$= \quad mA$
$\{pre_1 \land pre_2\}$

Derivation

$wp.\{pre_1\}\{pre_2\}.\psi'$
$\equiv pre_1 \land pre_2 \land \psi$ [by definition of wp]
$\equiv wp.\{pre_1 \land pre_2\}.\psi'$ [by definition of wp]

□

Law *mA* Merge annotations (coercions)

$[post_1][post_2]$
$= \quad mA$
$[post_1 \land post_2]$

Derivation

$wp.[post_1][post_2].\psi'$
$\equiv post_1[\_/'] \Rightarrow (post_2[\_/'] \Rightarrow \psi)$ [by definition of wp]
$\equiv (post_1 \land post_2)[\_/'] \Rightarrow \psi$ [by predicate calculus]
$\equiv wp.[post_1 \land post_2].\psi'$ [by definition of wp]

□

Law *mpS* Merge parametrised statements

$$(\mathbf{par}_2\ dvl_2 \bullet (\mathbf{par}_1\ dvl_1 \bullet p)(el_1))(el_2)$$

$= \quad mpS$

$$(\mathbf{par}_1\ dvl_1;\ \mathbf{par}_2\ dvl_2 \bullet p)(el_1, el_2)$$

where dvl_1 and dvl_2 declare the variables of vl_1 and vl_2, respectively.

Syntactic Restrictions

- vl_1 and vl_2 are disjoint;
- The variables of vl_2 are not free in el_1;
- If $\mathbf{par}_1$ and $\mathbf{par}_2$ are either result or value-result, then el_1 and el_2 are disjoint.

Derivation By definition.

□

Law *pcallI* Call to a non-recursive procedure introduction

$$|[\ \mathbf{proc}\ pn \mathrel{\widehat=} (fpd \bullet p_1) \bullet p_2[(fpd \bullet p_1)]\ ]|$$

$= \quad pcallI$

$$|[\ \mathbf{proc}\ pn \mathrel{\widehat=} (fpd \bullet p_1) \bullet p_2[pn]\ ]|$$

Syntactic Restriction pn is not recursive.

Derivation

$|[\ \mathbf{proc}\ pn \mathrel{\widehat=} (fpd \bullet p_1) \bullet p_2[(fpd \bullet p_1)]\ ]|$

$= p_2[(fpd \bullet p_1)](\mu(fpd \bullet p_1))$ [by definition]

$= p_2[(fpd \bullet p_1)](fpd \bullet p_1)$ [by pn is not recursive]

$= p_2[pn](fpd \bullet p_1)$ [by a property of substitution]

$= p_2[pn](\mu(fpd \bullet p_1))$ [by pn is not recursive]

$= |[\ \mathbf{proc}\ pn \mathrel{\widehat=} (fpd \bullet p_1) \bullet p_2[pn]\ ]|$ [by definition]

□

Law *pcallI* Procedure call introduction in the main program of a variant block

$$|[\ \mathbf{proc}\ pn \mathrel{\widehat=} (fpd \bullet p_1)\ \mathbf{variant}\ n\ \mathbf{is}\ e \bullet p_2[(fpd \bullet p_3)]\ ]|$$

$\sqsubseteq \quad pcallI$

$$|[\ \mathbf{proc}\ pn \mathrel{\widehat=} (fpd \bullet p_1)\ \mathbf{variant}\ n\ \mathbf{is}\ e \bullet p_2[pn]\ ]|$$

provided $\{n = e\}\ p_3 \sqsubseteq p_1$

Syntactic Restrictions

- pn is not free in p_3;
- n is not free in e and p_3.

Derivation

$[\![\,\mathbf{proc}\ pn \mathrel{\widehat=} (fpd \bullet p_1)\ \mathbf{variant}\ n\ \mathbf{is}\ e \bullet p_2[(fpd \bullet p_3)]\,]\!]$

$= p_2[(fpd \bullet p_3)](\mu(fpd \bullet [\![\,\mathbf{con}\ n : \mathbb{Z} \bullet p_1\,]\!]))$ [by definition]

$= p_2[(fpd \bullet [\![\,\mathbf{con}\ n : \mathbb{Z} \bullet \{n = e\}\ p_3\,]\!])](\mu(fpd \bullet [\![\,\mathbf{con}\ n : \mathbb{Z} \bullet p_1\,]\!]))$

[by n is not free in e and p_3]

$= p_2[(fpd \bullet [\![\,\mathbf{con}\ n : \mathbb{Z} \bullet p_1\,]\!])](\mu(fpd \bullet [\![\,\mathbf{con}\ n : \mathbb{Z} \bullet p_1\,]\!]))$ [by the proviso]

$= p_2[(fpd \bullet [\![\,\mathbf{con}\ n : \mathbb{Z} \bullet p_1\,]\!])](fpd \bullet [\![\,\mathbf{con}\ n : \mathbb{Z} \bullet p_1\,]\!])$ [by pn is not free in p_1]

$= p_2[pn](fpd \bullet [\![\,\mathbf{con}\ n : \mathbb{Z} \bullet p_1\,]\!])$ [by a property of substitution]

$= p_2[pn](\mu(fpd \bullet [\![\,\mathbf{con}\ n : \mathbb{Z} \bullet p_1\,]\!]))$ [by pn is not free in p_1]

$= [\![\,\mathbf{proc}\ pn \mathrel{\widehat=} (fpd \bullet p_1)\ \mathbf{variant}\ n\ \mathbf{is}\ e \bullet p_2[pn]\,]\!]$ [by definition]

□

The derivation of the next formulation of *pcallI* relies on Lemmas D.5 and D.6, which we present below. In [35, p.73], we can find a more restricted version of Lemma D.5, where just programs (parametrised statements with empty formal parameter declarations) are considered. Since [35] outlines a proof of this special case, for the sake of brevity, we consider just parametrised statements with ordinary (non-empty) formal parameter declarations in the proof of Lemma D.5. This proof relies on the following additional lemma.

Lemma D.4 *Let a family of programs p_i be such that, for any i, $p_i \sqsubseteq c(fpd \bullet \sqcup\{j \mid j < i \bullet p_j\})$, for a non-empty formal parameter declaration fpd, and monotonic c. Then $p_i \sqsubseteq c(\mu(fpd \bullet c))$, for all i.*

Proof By induction:

(Case $i = 0$)

p_0

$\sqsubseteq c(fpd \bullet \sqcup\{j \mid j < 0 \bullet p_j\})$ [by assumption]

$= c(fpd \bullet \sqcup\varnothing)$ [by a property of numbers]

$= c(fpd \bullet \mathbf{abort})$ [by **abort** is the least refined program]

$\sqsubseteq c(\mu(fpd \bullet c))$ [by $(fpd \bullet \mathbf{abort})$ is the least refined parametrised statement]

(Case $i > 0$)

p_i

$\sqsubseteq c(fpd \bullet \sqcup\{j \mid j < i \bullet p_j\})$ [by assumption]

$\sqsubseteq c(fpd \bullet \sqcup\{j \mid j < i \bullet c(\mu(fpd \bullet c))\})$ [by induction hypothesis]

$= c(fpd \bullet c(\mu(fpd \bullet c)))$ [by a property of least upper bounds]

$= c((fpd \bullet c)(\mu(fpd \bullet c)))$ [by a property of contexts]

$= c(\mu(fpd \bullet c))$ [by a property of fixed points]

□

Lemma D.5 *If, for an integer constant n, an integer expression e, a formal parameter declaration fpd, and a program p, we have that* $\{n = e\}\ p \sqsubseteq c(fpd \bullet \{0 \le e < n\}\ p)$*, then we can deduce that* $(fpd \bullet p) \sqsubseteq \mu(fpd \bullet c)$*, provided c is a monotonic context, and n is not free in p and c.*

Proof The assumption can be written as $\{n = e\}\ p \sqsubseteq c(fpd \bullet \sqcup\{j \mid j < n \bullet \{j = e\}\ p\})$, as we show below.

$wp.\{0 \le e < n\}\ p.\psi$

$\equiv wp.\{\bigvee\{j \mid j < n \bullet j = e\}\}\ p.\psi$ [by predicate calculus]

$\equiv \bigvee\{j \mid j < n \bullet j = e\} \wedge wp.p.\psi$ [by definition of wp]

$\equiv \bigvee\{j \mid j < n \bullet j = e \wedge wp.p.\psi\}$ [by predicate calculus]

$\equiv \bigvee\{j \mid j < n \bullet wp.\{j = e\}\ p.\psi\}$ [by definition of wp]

$\equiv wp.\sqcup\{j \mid j < n \bullet \{j = e\}\ p\}.\psi$ [by a property of $\sqcup$]

So, by Lemma D.4, $\{n = e\}\ p \sqsubseteq c(\mu(fpd \bullet c))$, for all n. Consequently, as we prove below, $p \sqsubseteq c(\mu(fpd \bullet c))$.

$wp.p.\psi$

$\equiv (\exists n \bullet n = e) \wedge wp.p.\psi$ [by predicate calculus]

$\equiv \exists n \bullet n = e \wedge wp.p.\psi$ [by n is not free in $wp.p.\psi$]

$\equiv \exists n \bullet wp.\{n = e\}\ p.\psi$ [by definition of wp]

$\Rightarrow \exists n \bullet wp.c(\mu(fpd \bullet c)).\psi$ [by the conclusion above]

$\equiv wp.c(\mu(fpd \bullet c)).\psi$ [by n is not free in $wp.c(\mu(fpd \bullet c)).\psi$]

Using this result, we can get to the required conclusion as follows.

$(fpd \bullet p)$

$\sqsubseteq (fpd \bullet c(\mu(fpd \bullet c)))$ [by the result above]

$= (fpd \bullet c)(\mu(fpd \bullet c))$ [by a property of contexts]

$= \mu(fpd \bullet c)$ [by a property of fixed points]

□

Lemma D.6 *For any program context* $c[pn]$, $\mu\, c[\mu\, c[pn]] \sqsubseteq \mu\, c[pn]$.

Proof $\mu\, c[pn]$ is a fixed point of $c[\mu\, c[pn]]$, as we show below.

$$\begin{array}{ll}
c[\mu\, c[pn]](\mu\, c[pn]) & \\
= c[pn](\mu\, c[pn]) & \text{[by a property of substitution]} \\
= \mu\, c[pn] & \text{[by a property of fixed points]}
\end{array}$$

Therefore, $\mu\, c[\mu\, c[pn]] \sqsubseteq \mu\, c[pn]$, as required.

□

Law *pcallI* Recursive call introduction

$$\begin{array}{ll}
& |[\, \mathbf{proc}\ pn \mathrel{\widehat=} (fpd \bullet p_1[(fpd \bullet \{0 \le e < n\}\ p_3)])\ \mathbf{variant}\ n\ \mathbf{is}\ e \bullet p_2 \,]| \\
\sqsubseteq & \mathit{pcallI} \\
& |[\, \mathbf{proc}\ pn \mathrel{\widehat=} (fpd \bullet p_1[pn]) \bullet p_2 \,]|
\end{array}$$

provided $\{n = e\}\ p_3 \sqsubseteq p_1[(fpd \bullet \{0 \le e < n\}\ p_3)]$.

Syntactic Restriction n is not free in p_3 and $p_1[pn]$

Derivation

$$\begin{array}{ll}
|[\, \mathbf{proc}\ pn \mathrel{\widehat=} (fpd \bullet p_1[(fpd \bullet \{0 \le e < n\}\ p_3)])\ \mathbf{variant}\ n\ \mathbf{is}\ e \bullet p_2 \,]| & \\
= p_2(\mu(fpd \bullet |[\, \mathbf{con}\ n : \mathbb{Z} \bullet p_1[(fpd \bullet \{0 \le e < n\}\ p_3)] \,]|)) & \text{[by definition]} \\
\sqsubseteq p_2(\mu(fpd \bullet |[\, \mathbf{con}\ n : \mathbb{Z} \bullet p_1[(fpd \bullet p_3)] \,]|)) & \text{[by } \mathit{assumpR} \text{ and } \mathit{sIC}\text{]} \\
\sqsubseteq p_2(\mu(fpd \bullet |[\, \mathbf{con}\ n : \mathbb{Z} \bullet p_1[\mu(fpd \bullet p_1[pn])] \,]|)) & \text{[by Lemma D.5 and the proviso]} \\
\sqsubseteq p_2(\mu(fpd \bullet p_1[\mu(fpd \bullet p_1[pn])])) & \text{[by } n \text{ is not free in } p_1\text{]} \\
\sqsubseteq p_2(\mu(fpd \bullet p_1[pn])) & \text{[by Lemma D.6]} \\
= |[\, \mathbf{proc}\ pn \mathrel{\widehat=} (fpd \bullet p_1[pn]) \bullet p_2 \,]| & \text{[by definition]}
\end{array}$$

□

Law *prcI* Procedure introduction

$$\begin{array}{ll}
& p_2 \\
= & \mathit{prcI} \\
& |[\, \mathbf{proc}\ pn \mathrel{\widehat=} (fpd \bullet p_1) \bullet p_2 \,]|
\end{array}$$

Syntactic Restrictions

- pn is not free in p_2;
- $(fpd \bullet p_1)$ is well-scoped and well-typed.

Derivation

p_2

$= p_2(\mu(fpd \bullet p_1))$ [by pn is not free in p_2]

$= |[\,\textbf{proc}\ pn \mathrel{\widehat=} (fpd \bullet p_1) \bullet p_2\,]|$ [by definition]

$\square$

Law rS Result specification

$$
\begin{array}{ll}
 & w, vl_2 : [pre, post] \\
= & rS \\
 & (\textbf{res}\ dvl_1 \bullet w, vl_1 : [pre, post[vl_1'/vl_2']\,])(vl_2)
\end{array}
$$

where dvl_1 declares the variables of vl_1.

Syntactic Restrictions

- vl_1 and vl_2 have the same length and contain no duplicated variables:
- The variables of vl_1 are not in w, are not free in pre, and are not dashed;
- The variables of vl_1 and vl_1' and are not free in $post$;
- The variables of vl_2 are not in w.

Derivation

$wp.w, vl_2 : [pre, post].\psi'$

$\equiv pre \land (\forall dw';\ dvl_2' \bullet post \Rightarrow \psi')[\_/']$ [by definition of wp]

$\equiv pre \land (\forall dw';\ dl' \bullet post[l'/vl_2'] \Rightarrow \psi'[l'/vl_2'])[\_/']$ [by l' are fresh and vl_2' are not in w']

$\equiv pre \land (\forall dw';\ dl' \bullet post[vl_1'/vl_2'][l'/vl_1'] \Rightarrow \psi'[l'/vl_2'])[\_/']$ [by vl_1' are not free in $post$]

$\equiv \forall dl \bullet pre \land (\forall dw';\ dl' \bullet post[vl_1'/vl_2'][l'/vl_1'] \Rightarrow \psi'[l'/vl_2'])[\_/']$ [by l are fresh]

$\equiv \forall dl \bullet pre[l/vl_1] \land (\forall dw';\ dl' \bullet post[vl_1'/vl_2'][l, l'/vl_1, vl_1'] \Rightarrow \psi'[l'/vl_2'])[\_/']$
[by vl_1 are not free in pre and $post$]

$\equiv \forall dl \bullet pre[l/vl_1] \land (\forall dw';\ dl' \bullet post[vl_1'/vl_2'][l, l'/vl_1, vl_1'] \Rightarrow \psi[l/vl_2][l'/l]')[l/l'][\_/']$
[by a property of substitution]

$\equiv wp.|[\,\textbf{var}\ dl \bullet (w, vl_1 : [pre, post[vl_1'/vl_2']\,])[l, l'/vl_1, vl_1'];\ vl_2 := l\,]|.\psi'$ [by vl_1 are not in w]

$\equiv wp.(\textbf{res}\ dvl_1 \bullet w, vl_1 : [pre, post[vl_1'/vl_2']\,])(vl_2).\psi'$ [by definition]

$\square$

Law *rS* Result specification (function application as actual parameter)

$$w, f : [pre, \{x?\} \lhd f' = \{x?\} \lhd f \land post[f'\ x?/fp']\,]$$

$=$ *rS*

$$(\mathbf{res}\ fp : t \bullet w, fp : [pre, post])(f\ \ x?)$$

where t is the type that contains the range of f.

Syntactic Restrictions

- f is of a function type;
- f and fp are not in w;
- f and f' and are not free in $post$.

Derivation

$$w, f : [pre, \{x?\} \lhd f' = \{x?\} \lhd f \land post[f'\ x?/fp']\,]$$

$\sqsubseteq$ *vrbI*

$$[\![\ \mathbf{var}\ v : t \bullet$$
$$\quad v, w, f\ : [pre, \{x?\} \lhd f' = \{x?\} \lhd f \land post[f'\ x?/fp']\,] \qquad \lhd$$
$$]\!]$$

$\sqsubseteq$ *fassigI*

$$v, w, f : [pre, \{x?\} \lhd (f' \oplus \{x? \mapsto v'\}) = \{x?\} \lhd f \land post[f'\ x?/fp'][f' \oplus \{x? \mapsto v'\}/f']\,]\ ; \qquad \lhd$$
$$f := f \oplus \{x? \mapsto v\}$$

$\sqsubseteq$ *sP*

$$v, w, f\ : [pre, \{x?\} \lhd f' = \{x?\} \lhd f \land post[v'/fp']\,]$$

$\sqsubseteq$ *cfR*

$$v, w : [pre, post[v'/fp']\,]$$

$\sqsubseteq$ *rS*

$$(\mathbf{res}\ fp : t \bullet w, fp : [pre, post])(v)$$

Therefore, we have proved that the refinement below holds.

$$w, f : [pre, \{x?\} \lhd f' = \{x?\} \lhd f \land post[f'\ x?/fp]\,]$$
$$\sqsubseteq [\![\ \mathbf{var}\ v : t \bullet (\mathbf{res}\ fp : t \bullet w, fp : [pre, post])(v)\ ;\ f := f \oplus \{x? \mapsto v\}\,]\!]$$

By definition, this variable block is $(\mathbf{res}\ fp : t \bullet w, fp : [pre, post])(f\ \ x?)$, as required.

$\Box$

Law *seqcI* Sequential composition introduction

$w, x : [pre, post]$

$\sqsubseteq$ *seqcI*

$w : [pre, mid[w'/w]\,] \;;\; w, x : [mid, post]$

Syntactic Restrictions

- *mid* is well-scoped and well-typed;
- *mid* has no free dashed variables;
- No free variable of *post* is in w.

Derivation

$wp.w, x : [pre, post].\psi$

$\equiv$ $pre \wedge (\forall dw';\ dx' \bullet post \Rightarrow \psi)[\_/']$ [by definition of *wp*]

$\Rightarrow$ $pre \wedge (\forall dw' \bullet mid[w'/w] \Rightarrow mid[w'/w] \wedge (\forall dw';\ dx' \bullet post \Rightarrow \psi))[\_/']$

[by predicate calculus]

$\equiv$ $pre \wedge (\forall dw' \bullet mid[w'/w] \Rightarrow mid[w'/w] \wedge (\forall dw';\ dx' \bullet post \Rightarrow \psi)[\_/'])[\_/']$

[by w' are not free in dw']

$\equiv$ $pre \wedge (\forall dw' \bullet mid[w'/w] \Rightarrow mid[w'/w] \wedge (\forall dw';\ dx' \bullet post \Rightarrow \psi)[\_/']'[\_/'])[\_/']$

[by a property of substitution]

$\equiv$ $pre \wedge (\forall dw' \bullet mid[w'/w] \Rightarrow mid[w'/w] \wedge (\forall dw';\ dx' \bullet post \Rightarrow \psi)[\_/']')[\_/']$

[by w are not free in dw', dx', *post*, and ψ, and w' are not free in dw']

$\equiv$ $pre \wedge (\forall dw' \bullet mid[w'/w] \Rightarrow (mid \wedge (\forall dw';\ dx' \bullet post \Rightarrow \psi)[\_/'])')[\_/']$

[by a property of substitution]

$\equiv$ $wp.(w : [pre, mid[w'/w]\,] \;;\; w, x : [mid, post]).\psi$ [by definition of *wp*]

□

Law *seqcI* Sequential composition introduction

$w, x, y!, z! : [pre, post]$

$\sqsubseteq$ *seqcI*

$|[\, \mathbf{con}\ dcl \bullet w, y! : [pre, mid] \;;\; w, x, y!, z! : [mid[cl/w][\_/'], post[cl/w]\,] \,]|$

where *dcl* declares the constants of *cl*.

Syntactic Restrictions

- *mid* is well-scoped and well-typed;
- The names of *cl* and *cl'* are not free in *mid* and $w, x, y!, z! : [pre, post]$;
- *cl* and w have the same length;
- The constants of *cl* have the same type as the corresponding variables of w.

Derivation

$wp.w, x, y!, z! : [pre, post].\psi$

$\equiv pre \land (\forall dw';\ dx';\ dy!;\ dz! \bullet post \Rightarrow \psi)[\_/']$ [by definition of wp]

$\Rightarrow pre \land (\forall dw';\ dy! \bullet mid \Rightarrow mid \land (\forall dw';\ dx';\ dy!;\ dz! \bullet post \Rightarrow \psi))[\_/']$
[by predicate calculus]

$\equiv pre \land (\forall dw';\ dy! \bullet mid \Rightarrow mid \land (\forall dw';\ dx';\ dy!;\ dz! \bullet post \Rightarrow \psi))[cl/w][w/cl][\_/']$
[by cl are not free in dw', $dy!$, mid, dx', $dz!$, $post$, and ψ]

$\equiv pre \land (\exists\, dcl \bullet cl = w \land$ [by predicate calculus]
$(\forall dw';\ dy! \bullet mid \Rightarrow mid \land (\forall dw';\ dx';\ dy!;\ dz! \bullet post \Rightarrow \psi))[cl/w])[\_/']$

$\equiv pre \land (\exists\, dcl \bullet cl = w \land$ [by a property of substitution]
$(\forall dw';\ dy! \bullet mid \Rightarrow mid[cl/w] \land (\forall dw';\ dx';\ dy!;\ dz! \bullet post[cl/w] \Rightarrow \psi))[cl/w])[\_/']$

$\equiv pre \land (\exists\, dcl \bullet cl = w \land$ [by predicate calculus]
$(\forall dw';\ dy! \bullet mid \Rightarrow mid[cl/w] \land (\forall dw';\ dx';\ dy!;\ dz! \bullet post[cl/w] \Rightarrow \psi)))[\_/']$

$\Rightarrow pre \land (\exists\, dcl \bullet$ [by predicate calculus]
$(\forall dw';\ dy! \bullet mid \Rightarrow mid[cl/w] \land (\forall dw';\ dx';\ dy!;\ dz! \bullet post[cl/w] \Rightarrow \psi)))[\_/']$

$\equiv pre \land \exists\, dcl \bullet$ [by dashed variables are not free in dcl]
$(\forall dw';\ dy! \bullet mid \Rightarrow mid[cl/w] \land (\forall dw';\ dx';\ dy!;\ dz! \bullet post[cl/w] \Rightarrow \psi))[\_/']$

$\equiv \exists\, dcl \bullet pre \land$ [by cl are not free in pre]
$(\forall dw';\ dy! \bullet mid \Rightarrow mid[cl/w] \land (\forall dw';\ dx';\ dy!;\ dz! \bullet post[cl/w] \Rightarrow \psi))[\_/']$

$\equiv \exists\, dcl \bullet pre \land$ [by a property of substitution]
$(\forall dw';\ dy! \bullet mid \Rightarrow mid[cl/w][\_/']' \land (\forall dw';\ dx';\ dy!;\ dz! \bullet post[cl/w] \Rightarrow \psi))[\_/']$

$\equiv \exists\, dcl \bullet pre \land$ [by w' are not free in dw']
$(\forall dw';\ dy! \bullet mid \Rightarrow mid[cl/w][\_/']' \land (\forall dw';\ dx';\ dy!;\ dz! \bullet post[cl/w] \Rightarrow \psi)[\_/'])[\_/']$

$\equiv \exists\, dcl \bullet pre \land (\forall dw';\ dy! \bullet mid \Rightarrow$ [by a property of substitution]
$mid[cl/w][\_/']' \land (\forall dw';\ dx';\ dy!;\ dz! \bullet post[cl/w] \Rightarrow \psi)[\_/']'[\_/'])[\_/']$

$\equiv \exists\, dcl \bullet pre \land$
$(\forall dw';\ dy! \bullet mid \Rightarrow mid[cl/w][\_/']' \land (\forall dw';\ dx';\ dy!;\ dz! \bullet post[cl/w] \Rightarrow \psi)[\_/']')[\_/']$
[by w are not free in dw',dx', $dy!$, $dz!$, $post[cl/w]$, and ψ, and w' are not free in dw']

$\Rightarrow \exists\, dcl \bullet pre \land (\forall dw';\ dy! \bullet mid \Rightarrow$ [by a property of substitution]
$(mid[cl/w][\_/'] \land (\forall dw';\ dx';\ dy!;\ dz! \bullet post[cl/w] \Rightarrow \psi)[\_/'])')[\_/']$

$\equiv wp.[\![\, \mathbf{con}\ dcl \bullet w, y! : [pre, mid]\ ;\ w, x, y!, z! : [mid[cl/w][\_/'], post[cl/w]\,]\ ]\!].\psi$
[by definition of wp]

□

Law *skI* Skip introduction

$$\begin{array}{ll} & w : [pre, post] \\ \sqsubseteq & skI \\ & \mathbf{skip} \end{array}$$

provided $pre \Rightarrow post[\_/']$.

Derivation

$$\begin{array}{ll} & wp.w : [pre, post].\psi' \\ \equiv & pre \wedge (\forall\, dw' \bullet post \Rightarrow \psi')[\_/'] & \text{[by definition of } wp\text{]} \\ \Rightarrow & post[\_/'] \wedge (\forall\, dw' \bullet post \Rightarrow \psi')[\_/'] & \text{[by the proviso]} \\ \Rightarrow & post[\_/'] \wedge (post \Rightarrow \psi')[\_/'] & \text{[by predicate calculus]} \\ \equiv & post[\_/'] \wedge (post[\_/'] \Rightarrow \psi) & \text{[by dashed variables are not free in } \psi\text{]} \\ \Rightarrow & \psi & \text{[by predicate calculus]} \\ \equiv & wp.\mathbf{skip}.\psi' & \text{[by definition of } wp\text{]} \end{array}$$

□

Law *slC* Skip left composition

$$\begin{array}{ll} & \mathbf{skip}\ ;\ p \\ = & slC \\ & p \end{array}$$

Derivation

$$\begin{array}{ll} & wp.(\mathbf{skip}\ ;\ p).\psi \\ \equiv & wp.\mathbf{skip}.(wp.p.\psi)' & \text{[by definition of } wp\text{]} \\ \equiv & wp.p.\psi & \text{[by definition of } wp\text{]} \end{array}$$

□

Law *sP* Strengthen postcondition

$$\begin{array}{ll} & w : [pre, post] \\ \sqsubseteq & sP \\ & w : [pre, npost] \end{array}$$

provided $pre \wedge npost \Rightarrow post$

Syntactic Restriction *npost* is well-scoped and well-typed.

Derivation

$$\begin{array}{ll} & wp.w : [pre, post].\psi \\ \equiv & pre \wedge (\forall\, dw' \bullet post \Rightarrow \psi)[\_/'] & \text{[by definition of } wp\text{]} \end{array}$$

$\Rightarrow pre \wedge (\forall\, dw' \bullet pre \wedge npost \Rightarrow \psi)[\_/']$ [by the proviso]

$\equiv (pre \wedge (\forall\, dw' \bullet pre \wedge npost \Rightarrow \psi))[\_/']$ [by no dashed variable is free in pre]

$\equiv (\forall\, dw' \bullet pre \wedge (pre \wedge npost \Rightarrow \psi))[\_/']$ [by w' are not free in pre]

$\equiv (\forall\, dw' \bullet pre \wedge (npost \Rightarrow \psi))[\_/']$ [by predicate calculus]

$\equiv pre \wedge (\forall\, dw' \bullet npost \Rightarrow \psi)[\_/']$ [by no dashed variable is free in pre]

$\equiv wp.w : [pre, npost].\psi$ [by definition of wp]

□

Law srC Skip right composition

$$\begin{array}{ll} & p \;;\; \mathbf{skip} \\ = & srC \\ & p \end{array}$$

Derivation

$wp.(p \;;\; \mathbf{skip}).\psi'$

$\equiv wp.p.(wp.\mathbf{skip}.\psi')'$ [by definition of wp]

$\equiv wp.p.\psi'$ [by definition of wp]

□

Law sS Simple specification

$$\begin{array}{ll} & vl := el \\ = & sS \\ & vl : [\mathrm{true}, vl' = el] \end{array}$$

Derivation

$wp.vl := el.\psi'$

$\equiv \psi[el/vl]$ [by definition of wp]

$\equiv \psi'[el/vl'][\_/']$ [by a property of substitution]

$\equiv (\forall\, dvl' \bullet vl' = el \Rightarrow \psi')[\_/']$ [by predicate calculus]

$\equiv wp.vl : [\mathrm{true}, vl' = el].\psi'$ [by definition of wp]

□

Law *vrbI* Variable introduction

$$\begin{array}{ll} & w : [pre, post] \\ = & vrbI \\ & |[\ \mathbf{var}\ dvl \bullet vl, w : [pre, post]\ ]| \end{array}$$

where *dvl* declares the variables of *vl*.

Syntactic Restrictions

- *dvl* is well-scoped and well-typed;
- The variables of *vl* and *vl'* are not free in $w : [pre, post]$ and are not dashed.

Derivation

$wp.w : [pre, post].\psi$

$\equiv pre \land (\forall dw' \bullet post \Rightarrow \psi)[\_/']$ [by definition of *wp*]

$\equiv \forall dvl \bullet pre \land (\forall dvl';\ dw' \bullet post \Rightarrow \psi)[vl/vl'][\_/']$

[by *vl* and *vl'* are not free in *pre*, *dvl'*, *dw'*, *post*, and ψ]

$\equiv wp.\ |[\ \mathbf{var}\ dvl \bullet vl, w : [pre, post]\ ]|\ .\psi$ [by definition of *wp*]

□

Law *vrbR* Variable renaming

$$\begin{array}{ll} & |[\ \mathbf{var}\ dvl \bullet p\ ]| \\ = & vrbR \\ & |[\ \mathbf{var}\ dul \bullet p[ul, ul'/vl, vl']\ ]| \end{array}$$

where *dvl* and *dul* declare the variables of *vl* and *ul*, respectively.

Syntactic Restrictions

- The variables of *ul* and *ul'* are not free in *p*;
- The variables of *ul* are not dashed.

Derivation

$wp.\ |[\ \mathbf{var}\ dvl \bullet p\ ]|\ .\psi$

$\equiv \forall dvl \bullet wp.p.\psi$ [by definition of *wp*]

$\equiv \forall dul \bullet (wp.p.\psi)[ul/vl]$ [by *ul* and *ul'* are not free in *p* and ψ]

$\equiv \forall dul \bullet (wp.p[ul, ul'/vl, vl'].\psi[ul, ul'/vl, vl'])[vl/ul][ul/vl]$ [by Lemma 3.1]

$\equiv \forall dul \bullet wp.p[ul, ul'/vl, vl'].\psi$ [by *vl* and *vl'* are not free in $p[ul, ul'/vl, vl']$ and ψ]

$\equiv wp.\ |[\ \mathbf{var}\ dul \bullet p[ul, ul'/vl, vl']\ ]|\ .\psi$ [by definition of *wp*]

□

Law *vrS* Value-result specification

$$p[vl_2, vl_2'/vl_1, vl_1']$$
$$= \quad vrS$$
$$(\textbf{val-res}\ dvl_1 \bullet p)(vl_2)$$

where dvl_1 declares the variables of vl_1.

Syntactic Restrictions

- The variables of vl_2 and vl_2' are not free in p;
- The variables of vl_1 and vl_2 are not dashed.

Derivation

$wp.p[vl_2, vl_2'/vl_1, vl_1'].\psi'$

$\equiv wp.p[l, l'/vl_1, vl_1'][vl_2, vl_2'/l, l'].\psi'$ [by l and l' are fresh]

$\equiv wp.p[l, l'/vl_1, vl_1'][vl_2, vl_2'/l, l'].\psi'[l'/vl_2'][vl_2'/l']$ [by l' are fresh]

$\equiv (wp.p[l, l'/vl_1, vl_1'].\psi'[l'/vl_2'])[vl_2/l]$

[by vl_2 and vl_2' are not free in p and $\psi'[l'/vl_2']$, and Lemma 3.1]

$\equiv (wp.p[l, l'/vl_1, vl_1'].\psi[l/vl_2]'[l'/l])[vl_2/l]$ [by a property of substitution]

$\equiv \forall\, dl \bullet (wp.p[l, l'/vl_1, vl_1'].\psi[l/vl_2]'[l'/l])[vl_2/l]$ [by predicate calculus]

$\equiv wp.\,\llbracket\, \textbf{var}\ dl \bullet l := vl_2\ ;\ p[l, l'/vl_1, vl_1']\ ;\ vl_2 := l \,\rrbracket\,.\psi'$ [by definition of wp]

$\equiv wp.(\textbf{val-res}\ dvl_1 \bullet p)(vl_2).\psi'$ [by definition]

□

Law *vrS* Value-result specification (specification statements)

$$w, vl_2 : [pre[vl_2/vl_1], post[vl_2/vl_1]\ ]$$
$$= \quad vrS$$
$$(\textbf{val-res}\ dvl_1 \bullet w, vl_1 : [pre, post[vl_1'/vl_2']\ ])(vl_2)$$

where dvl_1 declares the variables of vl_1.

Syntactic Restrictions

- The variables of vl_1 are not in w and are not dashed;
- The variables of vl_1' are not free in $post$;
- The variables of vl_2 and vl_2' are not free in $w : [pre, post]$.

Derivation

$w, vl_2 : [pre[vl_2/vl_1], post[vl_2/vl_1]\ ]$

$=$ by vl_1' are not free in $post$

$w, vl_2 : [pre[vl_2/vl_1], post[vl_1'/vl_2'][vl_2'/vl_1'][vl_2/vl_1]\ ]$

$= $ by vl_1 are not in w

$(w, vl_1 : [pre, post[vl_1'/vl_2']])[vl_2, vl_2'/vl_1, vl_1']$

$= vrS$

$(\textbf{val-res } dvl_1 \bullet w, vl_1 : [pre, post[vl_1'/vl_2']])(vl_2)$

$\square$

Law vrS Value-result specification (function application as actual parameter)

$w, f : [pre[f\ x?/fp], \{x?\} \lhd f' = \{x?\} \lhd f \land post[f\ x?, f'\ x?/fp, fp']]$

$= \quad vrS$

$(\textbf{val-res } fp : t \bullet w, fp : [pre, post])(f\ x?)$

where t is the type that contains the range of f.

Syntactic Restrictions

- f is of a function type;
- f and fp are not in w;
- f and f' and are not free in $post$.

Derivation

$w, f : [pre[f\ x?/fp], \{x?\} \lhd f' = \{x?\} \lhd f \land post[f\ x?, f'\ x?/fp, fp']]$

$\sqsubseteq vrbI$

$[\![\textbf{var } v : t \bullet$

$v.w, f : [pre[f\ x?/fp], \{x?\} \lhd f' = \{x?\} \lhd f \land post[f\ x?, f'\ x?/fp, fp']]$ $\lhd$

$]\!]$

$\sqsubseteq seqcI$

$v : [pre[f\ x?/fp], pre[f\ x?/fp]' \land v' = f'\ x?]$; $\lhd$

$v, w, f : [pre[f\ x?/fp] \land v = f\ x?, \{x?\} \lhd f' = \{x?\} \lhd f \land post[f\ x?, f'\ x?/fp, fp']]$ (i)

$\sqsubseteq assigI$

$v := f\ x?$

The specification statement (i) can be refined as follows.

$(i) \sqsubseteq fassigI$

$$v, w, f : \left[\begin{array}{l} pre[f\ x?/fp] \land v = f\ x?, \\ \left(\begin{array}{l} \{x?\} \lhd (f' \oplus \{x? \mapsto v'\}) = \{x?\} \lhd f \\ post[f\ x?, f'\ x?/fp, fp'][f' \oplus \{x? \mapsto v'\}/f'] \end{array} \right) \end{array} \right] \ ; \qquad \lhd$$

$f := f \oplus \{x? \mapsto v\}$

$$\begin{array}{l}
\sqsubseteq sP \\
\quad v, w, f : [pre[f\ x?/fp] \land v = f\ x?, \{x?\} \lhd f' = \{x?\} \lhd f \land post[f\ x?, v'/fp, fp']\ ] \\
\sqsubseteq cfR \\
\quad v, w : [pre[f\ x?/fp] \land v = f\ x?, post[f\ x?, v'/fp, fp']\ ] \\
\sqsubseteq sP \\
\quad v, w : [pre[f\ x?/fp] \land v = f\ x?, post[v, v'/fp, fp']\ ]
\end{array}$$

Below we discharge the proof-obligation generated by this application of sP.

$$\begin{array}{ll}
pre[f\ x?/fp] \land v = f\ x? \land post[v, v'/fp, fp'] & \\
\equiv pre[f\ x?/fp] \land v = f\ x? \land post[v, v'/fp, fp'][f\ x?/v] & \text{[by predicate calculus]} \\
\equiv pre[f\ x?/fp] \land v = f\ x? \land post[f\ x?, v'/fp, fp'] & \text{[by } v \text{ is not free in } post\text{]} \\
\Rightarrow post[f\ x?, v'/fp, fp'] & \text{[by predicate calculus]}
\end{array}$$

We proceed as follows.

$$\begin{array}{l}
\sqsubseteq wP \\
\quad v, w : [pre[v/fp], post[v, v'/fp, fp']\ ]
\end{array}$$

The proof-obligation generated by wP is discharged below.

$$\begin{array}{ll}
pre[f\ x?/fp] \land v = f\ x? & \\
\equiv pre[v/fp][f\ x?/v] \land v = f\ x? & \text{[by } v \text{ is not free in } pre\text{]} \\
\equiv pre[v/fp] \land v = f\ x? & \text{[by predicate calculus]} \\
\Rightarrow pre[v/fp] & \text{[by predicate calculus]}
\end{array}$$

The refinement continues as shown below.

$$\begin{array}{l}
\sqsubseteq vrS \\
\quad (\textbf{val-res}\ fp : t \bullet w, fp : [pre, post])(v)
\end{array}$$

Therefore, we have proved that the following refinement holds.

$$\begin{array}{l}
\quad w, f : [pre[f\ x?/fp], \{x?\} \lhd f' = \{x?\} \lhd f \land post[f\ x?, f'\ x?/fp, fp']\ ] \\
\sqsubseteq |[\ \textbf{var}\ v : t \bullet v := f\ x?\ ;\ (\textbf{val-res}\ fp : t \bullet w, fp : [pre, post])(v)\ ;\ f := f \oplus \{x? \mapsto v\}\ ]|
\end{array}$$

By definition, this variable block is $(\textbf{val-res}\ fp : t \bullet w, fp : [pre, post])(f\ x?)$, as required.

□

Law *vrtI* Variant introduction

$$\begin{array}{ll}
 & p_2 \\
= & vrtI \\
 & |[\ \textbf{proc}\ pn \mathrel{\widehat=} (fpd \bullet \{n = e\}\ p_1)\ \textbf{variant}\ n\ \textbf{is}\ e \bullet p_2\ ]|
\end{array}$$

Syntactic Restrictions

- pn and n are not free in e and p_1;
- $(fpd \bullet p_1)$ and e are well-scoped and well-typed.

Derivation

$$
\begin{array}{ll}
p_1 & \\
= p_2(|[\,\mathbf{con}\ n : \mathbb{N} \bullet \mu(fpd \bullet \{n = e\}\ p_1)\,]|) & \text{[by } pn \text{ is not free in } p_2] \\
= |[\,\mathbf{proc}\ pn \mathrel{\widehat=} (fpd \bullet \{n = e\}\ p_1)\ \mathbf{variant}\ n\ \mathbf{is}\ e \bullet p_2\,]| & \text{[by definition]}
\end{array}
$$

□

Law vS Value specification

$$
\begin{array}{ll}
 & w : [pre[el/vl], post[el, el'/vl, vl']\,] \\
= & vS \\
 & (\mathbf{val}\ dvl \bullet w : [pre, post])(el)
\end{array}
$$

where *dvl* declares the variables of *vl*.

Syntactic Restrictions

- The variables of *vl* are not in *w* and are not dashed;
- The variables of *w* are not free in *el*;
- *el* has no free dashed variables.

Derivation

$$
\begin{array}{ll}
wp.w : [pre[el/vl], post[el, el'/vl, vl']\,].\psi & \\
\equiv pre[el/vl] \land (\forall\, dw' \bullet post[el, el'/vl, vl'] \Rightarrow \psi)[\_/'] & \text{[by definition of } wp] \\
\equiv pre[l/vl][el/l] \land (\forall\, dw' \bullet post[l, l'/vl, vl'][el, el'/l, l'] \Rightarrow \psi)[\_/'] & \text{[by } l \text{ and } l' \text{ are fresh]} \\
\equiv pre[l/vl][el/l] \land (\forall\, dw' \bullet post[l, l'/vl, vl'] \Rightarrow \psi)[el, el'/l, l'][\_/'] & \\
 & \text{[by } l \text{ and } l' \text{ are fresh, and } w' \text{ are not free in } el'] \\
\equiv pre[l/vl][el/l] \land (\forall\, dw' \bullet post[l, l'/vl, vl'] \Rightarrow \psi)[l/l'][\_/'][el/l] & \text{[by a property of substitution]} \\
\equiv (pre[l/vl] \land (\forall\, dw' \bullet post[l, l'/vl, vl'] \Rightarrow \psi)[l/l'][\_/'])[el/l] & \text{[by a property of substitution]} \\
\equiv \forall\, dl \bullet (pre[l/vl] \land (\forall\, dw' \bullet post[l, l'/vl, vl'] \Rightarrow \psi)[l/l'][\_/'])[el/l] & \text{[by predicate calculus]} \\
\equiv wp.\,|[\,\mathbf{var}\ dl \bullet l := el\ ;\ (w : [pre, post])[l, l'/vl, vl']\,]|\,.\psi & \text{[by } vl \text{ are not in } w] \\
\equiv wp.(\mathbf{val}\ dvl \bullet w : [pre, post])(el).\psi & \text{[by definition]}
\end{array}
$$

□

Law wG Weakening guards

$$
\begin{array}{ll}
 & \mathbf{if}\ []\ i \bullet g_i \land g \rightarrow p_i\ \mathbf{fi} \\
\sqsubseteq & wG \\
 & \mathbf{if}\ []\ i \bullet g_i \rightarrow p_i\ \mathbf{fi}
\end{array}
$$

Derivation

$$\begin{array}{l} \quad \textbf{if} \; [] \; i \bullet g_i \wedge g \rightarrow p_i \; \textbf{fi} \\ \sqsubseteq \; dimG \\ \quad \textbf{if} \; [] \; i \bullet g_i \rightarrow p_i \; \textbf{fi} \end{array}$$

The proof-obligations generated by this application of *dimG* can be discharged as follows.

$$\begin{array}{ll} (\bigvee i \bullet g_i \wedge g) & \\ \equiv (\bigvee i \bullet g_i) \wedge g & \text{[by predicate calculus]} \\ \Rightarrow (\bigvee i \bullet g_i) & \text{[by predicate calculus]} \end{array}$$

$$\begin{array}{ll} (\bigvee i \bullet g_i \wedge g) \wedge g_i & \\ \equiv (\bigvee i \bullet g_i) \wedge g \wedge g_i & \text{[by predicate calculus]} \\ \Rightarrow g_i \wedge g & \text{[by predicate calculus]} \end{array}$$

□

Law *wP* Weakening precondition

$$\begin{array}{ll} & w : [pre, post] \\ \sqsubseteq & wP \\ & w : [npre, post] \end{array}$$

provided $pre \Rightarrow npre$.

Syntactic Restrictions

- *npre* is well-scoped and well-typed;
- *npre* has no free dashed variables.

Derivation

$$\begin{array}{ll} wp.w : [pre, post].\psi & \\ \equiv pre \wedge (\forall dw' \bullet post \Rightarrow \psi)[\_/'] & \text{[by definition of } wp] \\ \Rightarrow npre \wedge (\forall dw' \bullet post \Rightarrow \psi)[\_/'] & \text{[by the proviso]} \\ \equiv wp.w : [npre, post].\psi & \text{[by definition of } wp] \end{array}$$

□

Data Refinement Laws

In what follows, we present and derive the data refinement laws of ZRC. The lists of abstract and concrete variables are *avl* and *cvl*, respectively, and the coupling invariant is *ci*. The refinement law dR (data refinement), which can actually be used to data-refine a variable block, has been presented earlier on in this appendix.

Data Refinement Law Specification statement

$$vl, w : [pre, post]$$

$$\preccurlyeq$$

$$[\![\, \mathbf{con}\ davl \bullet cvl, w : [ci \wedge pre, \exists\, davl' \bullet ci' \wedge ul' = ul \wedge post]\ ]\!]$$

where

- $davl$ declares the variables of avl;
- $avl = vl, ul$, and vl and ul are disjoint.

Syntactic Restriction The variables of avl are not in w.

Derivation

$\exists\, davl \bullet ci \wedge wp.vl, w : [pre, post].\psi$

$\equiv\ \exists\, davl \bullet ci \wedge pre \wedge (\forall\, dvl';\ dw' \bullet post \Rightarrow \psi)[\_/']$ [by definition of wp]

$\equiv\ \exists\, davl \bullet ci \wedge pre \wedge (\forall\, dvl';\ dw' \bullet post \Rightarrow \psi)[ul/ul'][\_/']$ [by a property of substitution]

$\equiv\ \exists\, davl \bullet ci \wedge pre \wedge (\forall\, dul' \bullet ul' = ul \Rightarrow (\forall\, dvl';\ dw' \bullet post \Rightarrow \psi))[\_/']$
[by predicate calculus]

$\equiv\ \exists\, davl \bullet ci \wedge pre \wedge (\forall\, davl';\ dw' \bullet ul' = ul \wedge post \Rightarrow \psi)[\_/']$
[by vl' and w' are not in ul', and $avl = vl, ul$]

$\equiv\ \exists\, davl \bullet ci \wedge pre \wedge (\forall\, dcvl';\ dw';\ davl' \bullet ul' = ul \wedge post \Rightarrow \psi)[\_/']$
[by cvl' are not in ul', and are not free in dw', $davl'$, $post$, and ψ]

$\Rightarrow \exists\, davl \bullet ci \wedge pre \wedge (\forall\, dcvl';\ dw';\ davl' \bullet ci' \wedge ul' = ul \wedge post \Rightarrow ci' \wedge \psi)[\_/']$
[by predicate calculus]

$\Rightarrow \exists\, davl \bullet ci \wedge pre \wedge (\forall\, dcvl';\ dw';\ davl' \bullet ci' \wedge ul' = ul \wedge post \Rightarrow \exists\, davl' \bullet ci' \wedge \psi)[\_/']$
[by predicate calculus]

$\equiv\ \exists\, davl \bullet ci \wedge pre \wedge (\forall\, dcvl';\ dw' \bullet (\exists\, davl' \bullet ci' \wedge ul' = ul \wedge post) \Rightarrow \exists\, davl' \bullet ci' \wedge \psi)[\_/']$
[by avl' are not free in $davl'$]

$\equiv\ wp.[\![\, \mathbf{con}\ davl \bullet cvl, w : [ci \wedge pre, \exists\, davl' \bullet ci' \wedge ul = ul' \wedge post]\,]\!]\ .\ \exists\, davl' \bullet ci' \wedge \psi$
[by definition of wp]

□

Data Refinement Law Skip

skip

$$\preccurlyeq$$

skip

Derivation This law is an application of Theorem 3.11.

□

Data Refinement Law Assumption

$\{pre\}$

$\preccurlyeq$

$[\![\,\mathbf{con}\ davl \bullet cvl : [ci \wedge pre, ci'[avl/avl']]\ ]\!]$

where *davl* declares the variables of *avl*.

Derivation

$\{pre\}$

$= assumP$

$: [pre, \mathrm{true}]$

$\preccurlyeq$

$[\![\,\mathbf{con}\ davl \bullet cvl : [ci \wedge pre, \exists\, davl' \bullet ci' \wedge avl' = avl]\ ]\!]$

$= sP$ (in both directions)

$[\![\,\mathbf{con}\ davl \bullet cvl : [ci \wedge pre, ci'[avl/avl']]\ ]\!]$

□

Data Refinement Law Coercion

$[post]$

$\preccurlyeq$

$[\![\,\mathbf{con}\ davl \bullet cvl : [ci, (ci' \wedge post)[avl/avl']\ ]\!]$

where *davl* declares the variables of *avl*.

Derivation

$[post]$

$= cO$

$: [\mathrm{true}, post]$

$\preccurlyeq$

$[\![\,\mathbf{con}\ davl \bullet cvl : [ci, \exists\, davl' \bullet ci' \wedge avl' = avl \wedge post]\ ]\!]$

$= sP$ (in both directions)

$[\![\,\mathbf{con}\ davl \bullet cvl : [ci, (ci' \wedge post)[avl/avl']\ ]\!]$

□

Data Refinement Law Sequential composition

$p_1 \ ;\ q_1$

$\preccurlyeq$

$p_2 \ ;\ q_2$

provided $p_1 \preccurlyeq p_2$ and $q_1 \preccurlyeq q_2$

Syntactic Restriction The variables of *avl* and *avl'* are not free in p_2 and q_2.

Derivation

$\exists\, davl \bullet ci \wedge wp.(p_1 \;;\; q_1).\psi$

$\equiv\; \exists\, davl \bullet ci \wedge wp.p_1.(wp.q_1.\psi)'$ [by definition of wp]

$\Rightarrow\; wp.p_2.\,\exists\, davl' \bullet ci' \wedge (wp.q_1.\psi)'$ [by a proviso]

$\equiv\; wp.p_2.\,\exists\, davl \bullet ci'[avl/avl'] \wedge (wp.q_1.\psi)'[avl/avl']$ [by predicate calculus]

$\equiv\; wp.p_2.(\exists\, davl \bullet ci \wedge wp.q_1.\psi)'$ [by a property of substitution]

$\Rightarrow\; wp.p_2.(wp.q_2.\exists\, davl' \bullet ci' \wedge \psi)'$ [by a proviso and monotonicity of wp]

$\equiv\; wp.(p_2 \;;\; q_2).\exists\, davl' \bullet ci' \wedge \psi$ [by definition of wp]

□

Data Refinement Law Alternation

$\textbf{if} \; [\!]\, i \bullet g_i \rightarrow p_i \;\textbf{fi}$

$\preccurlyeq$

$|[\, \textbf{con}\; davl \bullet \textbf{if} \; [\!]\, i \bullet ci \wedge g_i \rightarrow q_i \;\textbf{fi}\,]|$

provided $p_i \preccurlyeq q_i$

Syntactic Restriction The variables of avl and avl' are not free in q_i.

Derivation

$\exists\, davl \bullet ci \wedge wp.\textbf{if} \; [\!]\, i \bullet g_i \rightarrow p_i \;\textbf{fi}.\psi$

$\equiv\; \exists\, davl \bullet ci \wedge (\bigvee i \bullet g_i) \wedge (\bigwedge i \bullet g_i \Rightarrow wp.p_i.\psi)$ [by definition of wp]

$\Rightarrow\; \exists\, davl \bullet (\bigvee i \bullet ci \wedge g_i) \wedge (\bigwedge i \bullet (ci \wedge g_i) \Rightarrow ci \wedge wp.p_i.\psi)$ [by predicate calculus]

$\Rightarrow\; \exists\, davl \bullet (\bigvee i \bullet ci \wedge g_i) \wedge (\bigwedge i \bullet (ci \wedge g_i) \Rightarrow \exists\, davl \bullet ci \wedge wp.p_i.\psi)$ [by predicate calculus]

$\Rightarrow\; \exists\, davl \bullet (\bigvee i \bullet ci \wedge g_i) \wedge (\bigwedge i \bullet (ci \wedge g_i) \Rightarrow wp.q_i.\exists\, davl' \bullet ci' \wedge \psi)$ [by the proviso]

$\equiv\; wp.|[\, \textbf{con}\; davl \bullet \textbf{if} \; [\!]\, i \bullet ci \wedge g_i \rightarrow q_i \;\textbf{fi}\,]|\,.\exists\, davl' \bullet ci' \wedge \psi$ [by definition of wp]

□

The derivation of the next data refinement law relies on the lemma that follows. It establishes that, when variables that are not involved in the data refinement are renamed, the resulting programs are related by data refinement if they were before.

Lemma D.7 *For all programs p_1 and p_2, lists of abstract and concrete variables avl and cvl, and coupling invariant ci, if $p_1 \preccurlyeq p_2$ then $p_1[l, l'/vl, vl'] \preccurlyeq p_2[l, l'/vl, vl']$ provided the variables of vl are not in avl and are not free in ci, and the variables of l and l' are not free in p_1 and p_2, are not in avl and cvl, and are not free in ci. The variables of cvl and cvl' must not be free in p_1; the variables of avl and avl' must no be free in p_2; and avl and cvl must be disjoint.*

Proof

$\exists\, davl \bullet ci \wedge wp.p_1[l, l'/vl, vl'].\psi$

$\equiv\ \exists\, davl \bullet ci \wedge (wp.p_1[l, l'/vl, vl'][m, m'/vl, vl'].\psi[m, m'/vl, vl'])[vl/m]$ [by Lemma 3.1]

$\equiv\ \exists\, davl \bullet ci \wedge (wp.p_1[l, l'/vl, vl'].\psi[m'/vl'])[vl/m]$ [by program variables are not free in ψ]

$\equiv\ \exists\, davl \bullet ci \wedge (wp.p_1[l, l'/vl, vl'][vl, vl'/l, l'].\psi[m'/vl'][vl'/l'])[l/vl][vl/m]$ [by Lemma 3.1]

$\equiv\ \exists\, davl \bullet ci \wedge (wp.p_1.\psi[m'/vl'][vl'/l'])[l/vl][vl/m]$ [by l and l' are not free in p_1]

$\equiv\ (\exists\, davl \bullet ci \wedge wp.p_1.\psi[m'/vl'][vl'/l'])[l/vl][vl/m]$

[by vl are not free in $davl$ and ci, and l and vl are not in avl, and m are fresh]

$\Rightarrow (wp.p_2.\exists\, davl' \bullet ci' \wedge \psi[m'/vl'][vl'/l'])[l/vl][vl/m]$ [by assumption]

$\equiv\ (wp.p_2.(\exists\, davl' \bullet ci' \wedge \psi)[m'/vl'][vl'/l'])[l/vl][vl/m]$

[by vl' and l' are not free in $davl'$ and ci', and are not in avl', and m are fresh]

$\equiv\ (wp.p_2[l, l'/vl, vl'][vl, vl'/l, l'].(\exists\, davl' \bullet ci' \wedge \psi)[m'/vl'][vl'/l'])[l/vl][vl/m]$

[by l and l' are not free in p_2]

$\equiv\ (wp.p_2[l, l'/vl, vl'].(\exists\, davl' \bullet ci' \wedge \psi)[m'/vl'])[vl/m]$ [by Lemma 3.1]

$\equiv\ wp.p_2[l, l'/vl, vl'].\exists\, davl' \bullet ci' \wedge \psi$ [by Lemma 3.1]

□

Data Refinement Law Variable block

$|[\,\mathbf{var}\ dvl \bullet p_1\,]|$

$\preccurlyeq$

$|[\,\mathbf{var}\ dvl \bullet p_2\,]|$

provided $p_1 \preccurlyeq p_2$

Syntactic Restrictions

- The variables of αdvl are not in avl and are not free in ci;
- The variables of avl and avl' are not free in p_2.

Derivation

$\exists\, davl \bullet ci \wedge wp.\ |[\,\mathbf{var}\ dvl \bullet p_1\,]|\ .\psi$

$\equiv\ \exists\, davl \bullet ci \wedge \forall\, dl \bullet wp.p_1[l, l'/vl, vl'].\psi$ [by definition of wp]

$\equiv\ \exists\, davl \bullet \forall\, dl \bullet ci \wedge wp.p_1[l, l'/vl, vl'].\psi$ [by l are not free in ci]

$\Rightarrow \forall\, dl \bullet \exists\, davl \bullet ci \wedge wp.p_1[l, l'/vl, vl'].\psi$ [by predicate calculus]

$\Rightarrow \forall\, dl \bullet wp.p_2[l, l'/vl, vl'].\exists\, davl' \bullet ci' \wedge \psi$ [by Lemma D.7 and the proviso]

$\equiv\ wp.\ |[\,\mathbf{var}\ dvl \bullet p_2\,]|\ .\exists\, davl' \bullet ci' \wedge \psi$ [by definition of wp]

□

The lemma below, which is used in the derivation of the data refinement law that applies to constant blocks, is similar to Lemma D.7, but considers constants instead of variables.

Lemma D.8 *For all programs p_1 and p_2, lists of abstract and concrete variables avl and cvl, and coupling invariant ci, if $p_1 \preccurlyeq p_2$ then $p_1[l/cl] \preccurlyeq p_2[l/cl]$ provided the constants of cl are not in avl and are not free in ci, and the constants of l are not free in p_1 and p_2, are not in avl and cvl, and are not free in ci. The variables of cvl and cvl′ must not be free in p_1; the variables of avl and avl′ must not be free in p_2; and avl and cvl must be disjoint.*

Proof

$\exists\, davl \bullet ci \wedge wp.p_1[l/cl].\psi$

$\equiv \exists\, davl \bullet ci \wedge (wp.p_1[l/cl][m/cl].\psi[m/cl])[cl/m]$ [by Lemma 3.2]

$\equiv \exists\, davl \bullet ci \wedge (wp.p_1[l/cl].\psi[m/cl])[cl/m]$ [by a property of substitution]

$\equiv \exists\, davl \bullet ci \wedge (wp.p_1[l/cl][cl/l].\psi[m/cl][cl/l])[l/cl][cl/m]$ [by Lemma 3.2]

$\equiv \exists\, davl \bullet ci \wedge (wp.p_1.\psi[m/cl][cl/l])[l/cl][cl/m]$ [by l are not free in p_1]

$\equiv (\exists\, davl \bullet ci \wedge wp.p_1.\psi[m/cl][cl/l])[l/cl][cl/m]$

[by cl are not free in $davl$ and ci, l and cl are not in avl, and m are fresh]

$\Rightarrow (wp.p_2.\exists\, davl' \bullet ci' \wedge \psi[m/cl][cl/l])[l/cl][cl/m]$ [by assumption]

$\equiv (wp.p_2.(\exists\, davl' \bullet ci' \wedge \psi)[m/cl][cl/l])[l/cl][cl/m]$

[by cl and l are not free in $davl'$ and ci, cl are not in avl', and m are fresh]

$\equiv (wp.p_2[l/cl][cl/l].(\exists\, davl' \bullet ci' \wedge \psi)[m/cl][cl/l])[l/cl][cl/m]$ [by l are not free in p_2]

$\equiv (wp.p_2[l/cl].(\exists\, davl' \bullet ci' \wedge \psi)[m/cl])[cl/m]$ [by Lemma 3.2]

$\equiv wp.p_2[l/cl].\exists\, davl' \bullet ci' \wedge \psi$ [by Lemma 3.2]

□

Data Refinement Law Constant block

$|[\,\mathsf{con}\ dcl \bullet p_1\,]|$

$\preccurlyeq$

$|[\,\mathsf{con}\ dcl \bullet p_2\,]|$

provided $p_1 \preccurlyeq p_2$

Syntactic Restrictions

- The constants of αdcl are not in avl and are not free in ci;
- The variables of avl and avl' are not free in p_2.

Derivation

$\exists\, davl \bullet ci \wedge wp.|[\,\mathsf{con}\ dcl \bullet p_1\,]|.\psi$

$\equiv \exists\, davl \bullet ci \wedge \exists\, dl \bullet wp.p_1[l/cl].\psi$ [by definition of wp]

$\equiv \exists\, dl \bullet \exists\, davl \bullet ci \wedge wp.p_1[l/cl].\psi$ [by l are not free in ci and $davl$]

$\Rightarrow \exists\, dl \bullet wp.p_2[l/cl].\exists\, davl' \bullet ci' \wedge \psi$ [by Lemma D.8 and the proviso]

$\equiv wp.|[\,\mathbf{con}\ dcl \bullet p_2\,]|\,.\exists\, davl' \bullet ci' \wedge \psi$ [by definition of wp]

□

In the lemma that follows we rephrase a result presented in [18]: data refinement distributes through fixed points. This lemma is used in the derivation of the next data refinement law.

Lemma D.9 *For all program contexts pc_1 and pc_2, lists of abstract and concrete variables avl and cvl, and coupling invariant ci, if $pc_1 \preccurlyeq pc_2$, then $\mu\, pc_1 \preccurlyeq \mu\, pc_2$.*

A proof for this lemma is presented in [18].

Data Refinement Law Procedure block

$|[\,\mathbf{proc}\ pn \mathrel{\widehat=} p_1(pn) \bullet p_2(pn)\,]|$

$\preccurlyeq$

$|[\,\mathbf{proc}\ pn \mathrel{\widehat=} p_3(pn) \bullet p_4(pn)\,]|$

provided $p_1 \preccurlyeq p_3$ and $p_2 \preccurlyeq p_4$.

Derivation

$|[\,\mathbf{proc}\ pn \mathrel{\widehat=} p_1(pn) \bullet p_2(pn)\,]|$

$= p_2(\mu\, p_1)$ [by definition]

$\preccurlyeq p_4(\mu\, p_3)$ [by Lemma D.9 and the provisos]

$= |[\,\mathbf{proc}\ pn \mathrel{\widehat=} p_3(pn) \bullet p_4(pn)\,]|$ [by definition]

□

Data Refinement Law Iteration

$\mathbf{do} \,[]\, i \bullet g_i \rightarrow p_i\ \mathbf{do}$

$\preccurlyeq$

$\mathbf{do} \,[]\, i \bullet (\forall\, davl \bullet ci \Rightarrow g_i) \rightarrow q_i\ \mathbf{od}$

provided

- $(\exists\, davl \bullet ci \wedge (\bigvee i \bullet g_i)) \Rightarrow (\bigvee i \bullet \forall\, davl \bullet ci \Rightarrow g_i)$
- $p_i \preccurlyeq q_i$

Syntactic Restriction The variables of avl and avl' are not free in q_i.

Derivation

$\mathbf{do} \,[]\, i \bullet g_i \rightarrow p_i\ \mathbf{od}$

$=$ by definition

$|[\,\mathbf{proc}\ it \mathrel{\widehat=} \mathbf{if} \,[]\, i \bullet g_i \rightarrow p_i\ ;\ it \,[]\, \neg (\bigvee i \bullet g_i) \rightarrow \mathbf{skip}\ \mathbf{fi} \bullet it\,]|$

$$
\begin{array}{l}
\preccurlyeq \\
\quad |[\ \mathbf{proc}\ it \mathrel{\widehat=} |[\ \mathbf{con}\ davl \bullet \\
\qquad\qquad\qquad \mathbf{if}\ [\!]\ i \bullet ci \land g_i \rightarrow q_i\ ;\ it\ [\!]\ ci \land \neg\,(\bigvee i \bullet g_i) \rightarrow \mathbf{skip}\ \mathbf{fi} \qquad \lhd \\
\qquad\qquad]| \bullet \\
\qquad it \\
\quad]| \\
\sqsubseteq\ dimG \\
\quad \mathbf{if}\ [\!]\ i \bullet (\forall\, davl \bullet ci \Rightarrow g_i) \rightarrow q_i\ ;\ it\ [\!]\ \neg\,(\bigvee i \bullet \forall\, davl \bullet ci \Rightarrow g_i) \rightarrow \mathbf{skip}\ \mathbf{fi}
\end{array}
$$

The consequent of the first proof-obligation generated by this application of *dimG* is a tautology.

$$\neg\,(\bigvee i \bullet \forall\, davl \bullet ci \Rightarrow g_i) \lor (\bigvee i \bullet \forall\, davl \bullet ci \Rightarrow g_i)$$

The second proof-obligation can be discharged as follows.

$$
\begin{array}{ll}
((\bigvee i \bullet ci \land g_i) \lor (ci \land \neg\,(\bigvee i \bullet g_i))) \land (\forall\, davl \bullet ci \Rightarrow g_i) & \\
\equiv\ ((ci \land (\bigvee i \bullet g_i)) \lor (ci \land \neg\,(\bigvee i \bullet g_i))) \land (\forall\, davl \bullet ci \Rightarrow g_i) & \text{[by predicate calculus]} \\
\equiv\ ci \land ((\bigvee i \bullet g_i) \lor \neg\,(\bigvee i \bullet g_i)) \land (\forall\, davl \bullet ci \Rightarrow g_i) & \text{[by predicate calculus]} \\
\Rightarrow\ ci \land (ci \Rightarrow g_i) & \text{[by predicate calculus]} \\
\Rightarrow\ g_i & \text{[by predicate calculus]}
\end{array}
$$

The last proof-obligation is discharged below.

$$
\begin{array}{ll}
((\bigvee i \bullet ci \land g_i) \lor (ci \land \neg\,(\bigvee i \bullet g_i))) \land \neg\,(\bigvee i \bullet \forall\, davl \bullet ci \Rightarrow g_i) & \\
\equiv\ ci \land \neg\,(\bigvee i \bullet \forall\, davl \bullet ci \Rightarrow g_i) & \text{[by predicate calculus]} \\
\Rightarrow\ ci \land \neg\,(\exists\, davl \bullet ci \land (\bigvee i \bullet g_i)) & \text{[by the proviso]} \\
\equiv\ ci \land (\forall\, davl \bullet ci \Rightarrow \neg\,(\bigvee i \bullet g_i)) & \text{[by predicate calculus]} \\
\Rightarrow\ \neg\,(\bigvee i \bullet g_i) & \text{[by predicate calculus]}
\end{array}
$$

Since the variables of *avl* are not free in the alternation generated by the application of *dimG*, we can use *conR* to remove the constant block that declares these variables as constants. The resulting procedure block is shown below.

$$|[\ \mathbf{proc}\ it \mathrel{\widehat=} \mathbf{if}\ [\!]\ i \bullet (\forall\, davl \bullet ci \Rightarrow g_i) \rightarrow q_i\ ;\ it\ [\!]\ \neg\,(\bigvee i \bullet \forall\, davl \bullet ci \Rightarrow g_i) \rightarrow \mathbf{skip}\ \mathbf{fi} \bullet it\ ]|$$

By definition, this program is the iteration $\mathbf{do}\ [\!]\ i \bullet (\forall\, davl \bullet ci \Rightarrow g_i) \rightarrow q_i\ \mathbf{od}$, as required.

□

Bibliography

[1] R. J. R. Back. *On The Correctness of Refinement Steps in Program Development*. PhD thesis, Department of Computer Science, University of Helsinki, 1978. Report A-1978-4.

[2] R. J. R. Back. Correctness Preserving Program Refinements: Proof Theory and Applications. Technical Report Tract 131, Mathematisch Centrum, Amsterdam, 1980.

[3] R. J. R. Back. Procedural Abstraction in the Refinement Calculus. Technical report, Department of Computer Science, Åbo - Finland, 1987. Ser. A No. 55.

[4] R. J. R. Back. A Calculus of Refinements for Program Derivations. *Acta Informatica*, 25:593 - 624, 1988.

[5] R. J. R. Back. Data Refinement in the Refinement Calculus. In *Proceedings 22nd Hawai International Conference of System Sciences*, 1989.

[6] R. J. R. Back and J. Wright. Refinement Calculus, Part I: Sequential Nondeterministic Programs. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems: Models, Formalism, Correctness*, volume 430 of *Lecture Notes in Computer Science*, pages 42 – 66, Mook, The Netherlands, 1989. Springer-Verlag.

[7] R. J. R. Back and J. Wright. Refinement Concepts Formalised in Higher Order Logic. *Formal Aspects of Computing*, 2:247 – 274, 1990.

[8] S. M. Brien and J. E. Nicholls. Z Base Standard, Version 1.0. Technical Monograph TM-PRG-107, Oxford University Computing Laboratory, Oxford - UK, November 1992.

[9] D. Carrington, D. Duke, R. Duke, P. King, G. A. Rose, and G. Smith. Object-Z: An Object-oriented Extension to Z. *Formal Description Techniques, II (FORTE'89)*, pages 281 – 296, 1990.

[10] D. Carrington, D. Duke, I. Hayes, and J. Welsh. Deriving Modular Designs from Formal Specifications. *ACM Software Engineering Notes*, 18(5):89 – 98, December 1993.

[11] A. L. C. Cavalcanti, A. Sampaio, and J. C. P. Woodcock. An Inconsistency in Procedures, Parameters, and Substitution in the Refinement Calculus. *Science of Computer Programming*. To appear.

[12] V. A. O. Cordeiro, A. Sampaio, and S. R. L. Meira. From MooZ to Eiffel – A Rigorous Approach to System Development. In M. Naftalin, T. Denvir, and M. Bertran, editors, *FME'94:*

Industrial-Strength Formal Methods, volume 873 of *Lecture Notes in Computer Science*, pages 306 – 325, Barcelona, Spain, October 1994. Springer-Verlag.

[13] A. J. J. Dick, P. J. Krause, and J. Cozens. Computer Aided Transformation of Z into Prolog. In J. E. Nicholls, editor, *Z User Workshop*, Workshops in Computing, pages 71 – 85, Oxford - UK, December 1989. Springer-Verlag.

[14] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[15] E. W. Dijkstra and C. S. Scholten. *Predicate Calculus and Program Semantics*. Texts and Monographs in Computer Science. Springer-Verlag, 1989.

[16] A. Diller. *Z : An Introduction to Formal Methods*. John Wiley & Sons, 2nd edition, 1994.

[17] V. Doma and R. Nicholl. EZ : A System for Automatic Prototyping of Z Specifications. In S. Prehn and W. J. Toetenel, editors, *VDM' 91 Formal Software Development Methods*, volume 552 of *Lecture Notes in Computer Science*, pages 189 – 203. Springer-Verlag, 1991.

[18] P. H. B. Gardiner and C. C. Morgan. Data Refinement of Predicate Transformers. *Theoretical Computer Science*, 87:143 – 162, 1991.

[19] P. H. B. Gardiner and C. C. Morgan. A Single Complete Rule for Data Refinement. *Formal Aspects of Computing*, 5(4):367 – 382, 1993.

[20] L. Groves. Procedures in the Refinement Calculus: A New Approach? In H. Jifeng, editor, *7th Refinement Workshop*, Bath - UK, July 1996.

[21] L. Groves, R. Nickson, and M. Utting. A Tactic Driven Refinement Tool. In C. B. Jones, R. C. Shaw, and T. Denvir, editors, *5th Refinement Workshop*, Workshops in Computing, pages 272 – 297. Springer-Verlag, 1992.

[22] J. Grundy. A Window Inference Tool for Refinement. In C. B. Jones, R. C. Shaw, and T. Denvir, editors, *5th Refinement Workshop*, Workshops in Computing, pages 230 – 254. Springer-Verlag, 1992.

[23] U. Hamer and J. Peleska. Z Applied to the A330/340 CIDS Cabin Communication System. In M. G. Hinchey and J. P. Bowen, editors, *Applications of Formal Methods*, chapter 11, pages 253 – 284. Prentice-Hall, 1995.

[24] W. T. Harwood. Proof Rules for Balzac. Technical Report WTH/P7/001, Imperial Software Technology, Cambridge - UK, 1991.

[25] I. Hayes, editor. *Specification Case Studies*. Prentice-Hall, 2nd edition, 1993.

[26] W. H. Hesselink. *Programs, Recursion and Unbounded Choice – Predicate Transformation Semantics and Transformation Rules*. Cambridge Tracts in Theoretical Computer Science 27. Cambridge University Press, 1992.

[27] M. G. Hinchey and J. P. Bowen, editors. *Applications of Formal Methods*. Prentice-Hall, 1995.

[28] C. A. R. Hoare and Jifeng He. The Weakest Prespecification. Technical Monograph TM-PRG-44, Oxford University Computing Laboratory, Oxford - UK, June 1985.

[29] M. Johnson and P. Sanders. From Z Specifications to Functional Implementations. In J. E. Nicholls, editor, *Z User Workshop*, Workshops in Computing, pages 86 - 112, Oxford - UK, 1989. Springer-Verlag.

[30] C. B. Jones. *Software Development: A Rigorous Approach*. Prentice-Hall, 1980.

[31] R. B. Jones. ICL ProofPower. *BCS FACS FACTS*, Series III, 1(1):10 - 13, 1992.

[32] M. B. Josephs. Formal Methods for Stepwise Refinement in the Z Specification Language. Technical Monograph TR-PRG-1-86, Oxford University Computing Laboratory, Oxford - UK, 1986.

[33] M. B. Josephs. The Data Refinement Calculator for Z Specifications. *Information Processing Letters*, 27(1):29 - 33, February 1988.

[34] S. King. Z and the Refinement Calculus. In D. Bjørner and C. A. R. Hoare, editors, *VDM'90 VDM and Z - Formal Methods in Software Development*, volume 428 of *Lecture Notes in Computer Science*, pages 164 - 188, Kiel - FRG, April 1990. Springer-Verlag.

[35] S. King and C. Morgan. Exits in the Refinement Calculus. *Formal Aspects of Computing*, 7(1):54 - 76, 1995.

[36] K. Lano and H. Haughton, editors. *Object-oriented Specification Case Studies*. The Object-oriented Series. Prentice-Hall, 1994.

[37] S. R. L. Meira and A. L. C. Cavalcanti. Modular Object-Oriented Z Specifications. In J. Nicholls, editor, *Z User Workshop*, Workshops in Computing, pages 173 - 192, Oxford - UK, December 1990. Springer-Verlag.

[38] S. R. L. Meira and A. Sampaio. Modular Extensions to Z. In *VDM'90: VDM and Z - Formal Methods in Software Development*, volume 428 of *Lecture Notes in Computer Science*, pages 211 - 232, Kiel - FRG, April 1990. Springer-Verlag.

[39] C. C. Morgan. Auxiliary Variables in Data Refinement. *Information Processing Letters*, 29(6):293 - 296, 1988.

[40] C. C. Morgan. Data Refinement by Miracles. *Information Processing Letters*, 26(5), January 1988.

[41] C. C. Morgan. Procedures, parameters, and abstraction: Separate concerns. *Science of Computer Programming*, 11:17 - 27, 1988.

[42] C. C. Morgan. The Specification Statement. *ACM Transactions on Programming Languages and Systems*, 10(3):403 - 419, 1988.

[43] C. C. Morgan. Types and Invariants in the Refinement Calculus. In J. L. A. van de Snepscheut, editor, *Mathematics of Program Construction*, volume 375 of *Lecture Notes in Computer Science*, pages 363 - 378. Springer-Verlag, 1989.

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

[45] C. C. Morgan. *Programming from Specifications*. Prentice-Hall, 2nd edition, 1994.

[46] C. C. Morgan and P. H. B. Gardiner. Data Refinement by Calculation. *Acta Informatica*, 27(6):481 – 503, 1990.

[47] C. C. Morgan, K. Robinson, and P. H. B. Gardiner. On the Refinement Calculus. Technical Monograph TM-PRG-70, Oxford University Computing Laboratory, Oxford - UK, October 1988.

[48] J. M. Morris. A Theoretical Basis for Stepwise Refinement and the Programming Calculus. *Scienee of Computer Programming*, 9(3):287 – 306, 1987.

[49] J. M. Morris. Invariance Theorems for Recursive Procedures. Technical report, Department of Computer Science, University of Glasgow, 1988.

[50] J. M. Morris. Laws of Data Refinement. *Acta Informatica*, 26:287 – 308, 1989.

[51] D. S. Neilson. *From Z to C: Illustration of a Rigorous Development Method*. PhD thesis, Oxford University Computing Laboratory, Oxford - UK, 1990. Technical Monograph TM-PRG-101.

[52] B. F. Potter, J. E. Sinclair, and D. Till. *An Introduction to Formal Specification and Z*. Prentice-Hall, 2nd edition, 1996.

[53] G. B. Rafsanjani. From Object-Z to C++: A Structural Mapping. In J. P. Bowen and J. E. Nicholls, editors, *Z User Workshop*, Workshops in Computing, pages 166 – 179, London - UK, 1992. Springer-Verlag.

[54] D. Rann, J. Turner, and J. Whitworth. *Z: A Beginner's Guide*. Chapman & Hall, 1994.

[55] A. Sampaio. *An Algebraic Approach to Compiler Design*. PhD thesis, Oxford University Compnting Laboratory, Oxford - UK, 1993. Technical Monograph TM-PRG-110. Revised version to appear as volume 4 of AMAST (Algebraic Methodology and Software Technology) Series in Computing, World Scientific, 1997 (in press).

[56] C. T. Sennet. Demonstrating the Compliance of Ada Programs with Z Specifications. In C. B. Jones, R. C. Shaw, and T. Denvir, editors, *5th Refinement Workshop*, Workshops in Computing, pages 70 – 87, London - UK, 1992. Prentice-Hall.

[57] J. M. Spivey. *The fUZZ Manual*. Computing Science Consultancy, 34 Westlands Grove, Stockton Lane, York YO3 0EF, UK, 2nd edition, July 1992.

[58] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall, 2nd edition, 1992.

[59] S. Stepney, R. Barden, and D. Cooper, editors. *Object-orientation in Z*. Workshops in Computing. Springer-Verlag, 1992.

[60] A. Tarski. A Lattice Theoretical Fixed Point Theorem and its Applications. *Pacific Journal of Mathematics*, 5, 1955.

[61] S. H. Valentine. Z^{--}, an Executable Subset of Z. In J. E. Nicholls, editor, *Z User Workshop*, Workshops in Computing, pages 157 – 187, York - UK, 1991. Springer-Verlag.

[62] T. Vickers. An Overview of a Refinement Editor. In *5th Australian Software Engineering Conference*, pages 39 – 44, Sidney - Australia, May 1990.

[63] N. Ward. Adding Specification Constructors to the Refinement Calculus. In J. C. P. Woodcock and P. G. Larsen, editors, *FME'93: Industrial-Strength Formal Methods*, volume 670 of *Lecture Notes in Computer Seience*, pages 652 – 670. Springer-Verlag, 1993.

[64] J. C. P. Woodcock. Implementing Promoted Operations in Z. In C. B. Jones, R. C. Shaw, and T. Denvir, editors, *5th Refinement Workshop*, Workshops in Computing, London - UK, 1992. Prentice-Hall.

[65] J. C. P. Woodcock and J. Davies. *Using Z - Specification, Refinement, and Proof.* Prentice-Hall, 1996.

[66] J. B. Wordsworth. *Software Development with Z.* International Computer Science Series. Addison-Wesley, 1992.

[67] J. Wright. Program Refinement by Theorem Prover. In D. Till, editor, *6th Refinement Workshop*, Workshops in Computing, pages 121 – 150, London - UK, 1994. Springer-Verlag.

[68] J. Wright, J. Hekanaho, P. Luostarinen, and T. Langbacka. Mechanizing Some Advanced Refinement Concepts. *Formal Methods in System Design*, 3:49 – 81, 1993.