From Code to Requirements Specification – Reverse Engineering Using a Formal Approach

by

Nga Ting Alex Wan

A Thesis
Submitted to the Faculty of Graduate Studies
in Partial Fulfillment of the Requirements
for the Degree of

MASTER OF SCIENCE

Department of Computer Science
University of Manitoba
Winnipeg, Manitoba, Canada R3T 2N2

THE UNIVERSITY OF MANITOBA

FACULTY OF GRADUATE STUDIES

COPYRIGHT PERMISSION

FROM CODE TO REQUIREMENTS SPECIFICATION - REVERSE ENGINEERING USING A FORMAL APPROACH

by

NGA TING ALEX WAN

A Thesis submitted to the Faculty of Graduate Studies of the University of Manitoba in partial fulfillment of the requirements of the degree of

MASTER OF SCIENCE

Abstract

Services rendered by legacy systems that have evolved over decades are vital to many industries. Therefore, reengineering of these systems must ensure that the new systems provide the same functionalities as their ancestors, while exploiting new technologies. Reengineering involves reverse engineering an application from code to a higher level abstraction, and then reimplementing on a new platform. Ability to obtain the abstraction at requirements level enables maximum use of new development techniques and tools in reimplementation. Use of formal methods in specifying requirements helps eliminating unambiguity while enhancing confidence in consistency and correctness. This thesis presents a formal approach to reverse a program written in a C subset into a functional specification in Z notation, including (i) a description of the abstractions used in the reverse engineering process, (ii) a method to obtain these abstractions from a program written in a C subset, (iii) a method to derive the formal requirements specification from the abstractions, and (iv) a comparison of the logical strength of three sets of conditions to justify the correctness of the derived specification, as well as their application in reverse engineering.

Acknowledgment

First of all I have to express my greatest gratitude to Dr. Kasilingam Periyasamy, my Advisor, for his guidance, advice, and appreciation of this work.

I also need to thank Dr. Dekang Lin, and Dr. George Gratzer (Examiners) for their valuable suggestions on improving this work, as well as Dr. Dereck Meek, for chairing my thesis defense.

This work was partly supported by University of Manitoba Graduate Fellowship (1995) and University of Manitoba Faculty of Science Fellowship (1994).

I am also obligated to give thanks to the following people:

Millie Man If you didn't cook for me, I would have starved to death long time ago. Thanks for everything you do, and tireless listening to all my complains.

Deborah Loh Thanks for your prayers. You are my best friend! Thanks for listening to all my sorrow (and the bits of joy, too).

Christine Man Thanks for bringing out my inner maternal instinct by introducing me to Tamagotchi[1]. We have shared really neat jokes, haven't we?

Dad & Mom I am so proud of you. You have given me such a great home to grow up in.

Eddy Wan Take heart! Finish your thesis.

Adrian & Grace Man Thanks for your prayers. You two have given me such a warm family here.

Hon-Kee & Kathy Man Thanks for the delicious food that you have shared.

Joyce Man One fine cousin!

[1]Trademark of Bandai

Grandpa & Grandma I know. I won't get too anxious about things. Thank you for all your support and concern.

Finally, I wish to thank the following people for their support:

Capary Chow, Maggie Cheung, Tim Chan, Polly Choi, Keith Tang, Peggy So, Paddison and Mary Wong, ... (there are too many of them)

Alex Wan

August 8, 1997

e-mail: awan@cs.umanitoba.ca
Jump to 'alex secret' at http://www.cs.umanitoba.ca/~awan before it moves!

Contents

1 Introduction

For many organizations, legacy systems (systems developed more than a decade earlier) have become large and inflexible towards changes. Yet, many applications still rely on services rendered by those systems, and therefore, those systems are vital to businesses[4, p.273]. Often, changes to legacy systems are made on an ad hoc basis. However, such patching process will eventually become to complex so that organization must replace the legacy system[30, p.19]. One way to approach such replacement is *reengineering*. One of the important requirements of reengineering is to preserve the functionalities of the old code[7]. While definitions vary [2, 1], for the purpose of this thesis, software *reengineering* is defined as a two-phase process. The first phase recovers the abstract model of the application from code; this phase is called *reverse engineering*. The second phase reimplements the derived abstract model using new technology. This thesis adopts the definition for reverse-engineering from [15, p.5]; Accordingly, reverse-engineering *reverse engineering* "[is] the process of transforming or moving from one level of description of a system to a level which is regarded as more abstract or 'earlier' in terms of the standard [software development] life cycle". In the context of reengineering, a reverse engineering methodology that is able to recover higher level (*i.e.*, requirements specification) abstractions has the advantage of enabling reimplementation process to take full advantages of modern development systems and approaches at various levels of abstractions.

A good software requirements specification must correctly define all the requirements of the software product under consideration [18, p.11]. It must, however, not be inclined towards design, implementation, or project management details. The IEEE Standard for Requirements Specification[18] discusses several qualities of a requirements specification document. Among these, the qualities "unambiguity",

"consistency", "correctness", and 'traceability" can be best captured by writing the requirements using a formal notation. Recently, several software development projects have justified the claim that formal approaches used in these projects lead to uncovering many errors early in the development stage and also prove themselves to be useful in reasoning about interesting properties of the application[25]. It is for the same reason that this thesis attempts to use a formal approach to reverse engineering. In particular, if the code for an application is reversed into its formal requirements specification, the traceability, modifications and benefits of new technology during reimplementation process can all be justified.

From the above discussions, it is clear that for a reverse engineering approach to work best with reengineering, the target of the reverse engineering process should be a formal SRS. This thesis contributes to this goal by presenting a formal approach for extracting a functional specification (suitable for inclusion under the section, 'Specific Requirements'[18, p.23] in an SRS) from a program written in a C subset.

1.1 Reverse Engineering Technology

Gannod and Betty[6] advocated a two-phase approach to reverse engineering using formal methods. Detail description of the approach is presented in [12]. In order to benefit from recent advances in object-orientation and formal methods[6, p.335], their approach was aimed at producing an object-oriented design from an existing software. The design is expressed in a formal language based on predicate logic[6, p.345]. The first phase consists of obtaining a specification from a program, consisting of pre and post-conditions of each procedure. In the second phase classes and objects are identified from observing the interface (the parameter lists) between the procedures The first phase had been specified using a procedural lan-

guage with basic imperative programming constructs, assignments, alternations, iterations, and (non-recursive) procedures. A tool for assisting in phase one had been developed for a subset of Pascal. Specific guidelines[6, p.344] were also given for performing identification of classes (phase two). In particular, each structured datatype become the attribute of a certain class, and each procedure is assigned as a operation to a class based on the types of the structured variable(s) in its parameters list.

The Maintainer's Assistant[23] was a tool developed at University of Durham, U.K, that was aimed at assisting modification of an unfamiliar software given only its source code [23, p.308]. The major features[23, p.310] of the tool, apart from allowing direct editing of source code, is the inclusion of a large library of semantics preserving program transformation which may either be invoked by the user directly, or by a knowledge base. The validity of all available transformation has been established in [39]. Any editing or transformation (in case the transformation requires a condition that may not be proven automatically) that may change the semantics of the program is recorded separately. The tool supports programs written in a wide-spectrum language[23, p.309] (WSL), which was designed to have a simple semantics, and was able to seamlessly include specifications as statements. The use of such language enabled a transformation approach to be used in both translating between programming languages and obtaining a specification from code[23, p.312].

Spencer Rugaber proposed a reverse engineering methodology that was targeted towards reversing data processing applications[28]. The methodology features a top-down approach, which he claims had the advantage of providing more confidence in obtaining architectural information earlier, comparing to starting with understanding the implementation completely[28, p.2]. The methodology consists

of four phases[28, p.3]. In the first phase, a review of existing system documentation produces a textual *system description* formatted in a top-down fashion. The second phase consists of constructing a nested data flow diagram x by an analysis of the entire system's input-output behaviors. Consistency check may be performed among the nested layers of the diagram, and against the source code. The third phase involves analyzing and presenting the structure of the files that the system uses. In the forth phase, a program analysis technique invented by the author, called 'synchronized refinement', is used to obtain detailed description of certain functionalities based on incrementally annotating their functional description with *design decisions* detected from code.

In the REDO project[36], a general approach was developed for reverse engineering towards a formal specification[3]. They viewed reverse engineering as an iterative process among three phases[3, p.213]: 'clean', 'specify', and 'simplify'. The first phase involves *restructuring* of the program into a form which, in the second phase, may be translated into a sequence of equations. These equations relate the initial and final values of program variables in executing the program. A highly automated[3, p.222]normalization process[3, p.202-11] invented during the development of REDO is applied to present the equations in a normal form. The last phase is simplification of the sequence of equations. Among other mathematical simplifications[3, p.216], the use of *data equivalence*[3, p.216-22] is advocated for obtaining higher level abstraction. A *data equivalence* specifies rules for transforming a set of equations based on one data structure to another. When this general approach was applied to extracting object-oriented designs from COBOL programs[16], the three phases were specialized as follows. In the 'clean' phase, the COBOL program is translated to a semantically equivalent program in UNIFORM[5], which was designed to serve as a language-independent represen-

tation for tools development[5, p.124]. In the 'specify' phase, the mathematical description is constructed, which consists of a set of object classes identified from the program. Specifically, a class is created for each file, indexed array or report being the chief attribute. For each chief attribute, dataflow analysis is then used to collect its auxiliary variables (such as counters), and included in the respective object class as attributes. Meanwhile, *phases*[2] or slices that implement candidate operations for these object classes are identified. Sequences of normalized equations are used to describe the candidate operations. In the 'simplify' phase, the object classes specifications are translated into Z++, an object-oriented extension of the Z specification language. Finally, Lano, Breuer, and Haughton[16] also described a method to derive COBOL code back from a sequence of equations, and therefore, enables a form of reengineering[16, p.247]. Lano and Haughton[15, p.160-6] described how a similar approach may be applied to obtain an object-oriented design from a C program.

Müller *et al*[35] described an approach towards reverse engineering that was aimed at obtaining an architectural understanding of a large software system by identifying its subsystems, and studying the relationships among these subsystems. Both tasks, along with documentation of the results, were all supported by an integrated tool, Rigi. In the first step of the approach, a parser parses the subject program to create a directed weighted *resource-flow graph* (RFG) among entities of the program such as files, functions, and variables. In particular, the RFG documents the exchange of resources among these entities. In a *provision* relationship an entity is *supplied* by one entity (call the *supplier*) to another entity (called the *client*). For example, when a C structure of type `a` has a component of another structure type `i`, then `a` is a client of `i` in a provision relationship in which

[2]defined to be "maximal logically-connected piece of code which contains no [statements for opening or closing files]"

i supplies itself to i. *Requisition* relationship is simply the converse of provision relationship. In the second step, subgraphs of the original resource-flow graph are aggregated to form a hierarchy of a new kind of entities called subsystems. Each subsystem consists of another RFG in which the nodes are either entities from the original RFG, or subsystems at the next level down the hierarchy. Between two entities, we may calculate their *exact interface*, which lists separately (i) those resources exchanged between the entities, and (ii) those resources exchanged among the entities within each of these entities.

COBOL System Renovation Environment[19] (COBOL/SRE) implements an approach to reengineering based on recovery of reusable components from the subject software [19, p.64]. In a source browser, the analysis may select a segment (defined to be a set of statements, possibly non-contagious in the source text[19, p.87]) from a program which is deemed to be cooperating in performing a certain functionality. Program slicing is then used to select other relevant segments. When the analyst is convinced that all the code that implement the functionality is included, he may instruct the tool to pack the selected segments either into a subprogram , an independent program, or simply a source file containing the statements. The tool takes care of making all the necessary language-specific changes to the program necessary when packing subprograms or programs.

In summary, the essential ingredients of a reverse engineering approach are the following:

human intelligence Human intelligence remains a major ingredient to most reverse engineering approaches. For example, when using the Rigi tool, the analyst ultimately has to decide on how software entities *should be* aggregated into appropriate subsystems. The same thing happens when subprograms are extracted from COBOL programs in COBOL/SRE. However, the virtue of a reverse engineering

approach, as pointed out in [35, p.3-4], is not to be automated completely, but rather, provides support for unintelligent tasks as much as possible. For example, based on recognition of human cognitive abilities[10, p.5], Rigi, in addition to providing graph editing commands, also provides an interpretive scripting language to offer flexible end-user programming[10, p.5]. Two cognitive models of program comprehension are discussed in [34]. Mental representation of software and hints on improving program comprehension are discussed in [37].

artificial intelligence Numerous tools for reverse engineering are supported by some form of artificial intelligence. For example, when performing program transformation in Maintainer's Assistant, the analyst may ask a knowledge base for the appropriate transformation sequence[23, p.310]. There was also an attempt at University of Aberdeen, U.K.[11] in training artificial neural networks to recognize standard algorithms from COBOL programs. The problem of associating human-oriented concepts with their implementation in a program is known as *concept assignment problem*[40, p.72]. The DESIRE tool[40] attempts to alleviate this problem by having an intelligent agent to perform three related tasks[40, p.80]: search for all occurrences of any concept known by the agent, search for a concept specified by the user, or assign a concept to a given segment of code.

program slicing In general, a *slice* of a program may be defined to be a "complete program which contains a subset of the statements of the original program, and which perform a subset of the computations performed by the original program"[9, p.55][3] Two kinds of program slicing that are frequently referred to[26, 32, 19] are forward and backward slicing. The forward slice[19, p.68] of a sequence of statements with respect to a variable are those statements whose be-

[3] in other literature[32, p.2] slices need not form an executable program

havior (including flow of control and effect on values of variables) depends on the initial value of that variable. Similarly, the backward slice[19, p.68] of a sequence of statements with respect to a variable are those statements whose behavior (including flow of control and effect on values of variables) may affect the final value of that variable. In the context of program understanding, one application of forward slice is to study how the inputs are processed[19, p.68] (by forward slicing with respect to those variables that hold the inputs). Similarly, backward slicing is useful for tracing back how a particular output at a particular statement of a program is evaluated [19, p.68]. Among the reverse engineering approaches described previously, program slicing is used for identifying functionalities of COBOL code segments using COBOL/SRE[19] or the approach described in [16]. Program slicing may also occur at a different granularity than statements. An example is *interface slicing*[9], in which the slicing extracts a set of type and global variable definitions and subprograms that a given set of subprograms require in order to operate[9, p.59]. In addition, program slicing may ease analysis of a loop by enabling one to analyze the effect of executing the loop on each variable separately[17, p.60].

program restructuring Program restructuring is a valuable tool for restructuring code so as to reduce the number of control structures to ease program understanding. Breuer, Lano, and Bowen[3] described a method of transforming an unstructured program into a structured one based on its representation as a set of equations. Ward[39] provided an extensive library of transformation for a wide-spectrum language which may be used for[39, p.4] transforming unstructured programs into structured programs, transforming recursive procedures into iterations. Inclusion of general specifications[39, p.17] in the wide-spectrum language enable the same program transformation framework to describe transformation

between a program and a specification[23, p.312].

denotation semantics The denotational semantics of the programming language used for implementation also plays a crucial role in reverse engineering. The denotational semantics[21] of a language consists of a set of semantic functions from the states of the program to denotations in the semantic domain[21, p.106]. The semantic function for a given kind of construct of the language is defined as some composition of the semantic functions of its syntactic components[13, p.21]. In *function abstraction*[17][15, Ch.5],[22, Ch.5], the same technique is applied in order to specify the functionality of a segment of code in terms of the relationship between the pre and post conditions in executing the code. However, function abstraction is different from denotational semantics in handling recursive and iterative constructs. In particular, a loop may be presented in function abstraction as a recursive function (for example, [17, p.59]), while the denotational semantics of a loop must be represented non-recursively (as a *fixed point*[4] of some function[13, p.23]). Peter Baumann *et al.*[27] advocated the use of program analysis techniques based on denotational semantics for several reasons[27, p.10]. One of those reasons is that it enables *abstract interpretation*[27, p.16], a program analysis technique that supports such analysis as control and data flow analysis, which are in turn essential ingredients[26] of program slicing. The function abstraction procedure employed by REDO[15, Ch.5] also uses the notion of *translational semantics*[15, p.97], which defines the semantics of a language by first defining the semantics of a small subset of the language (for example, with denotational semantics), and then extend that subset to the entire language by defining translation rules between the rest of the language with that subset. The semantics of the wide-spectrum language[39] supported by Maintenance Assistant was also defined in a similar

[4] more discussions on fixed points is given on page 121.

fashion[23, p.308-9].

1.2 Reverse Engineering Using a Formal Approach

Wordsworth[42] described an idea of using Z schemas to specify the function of procedural program. In particular, this function may be represented by the relationship between the values of program variables before and after the execution of a construct[42, p.198]. The function of an entire program may be deduced by a process called *stepwise abstraction* [42, p.198]. In stepwise abstraction[42, p.198], the functions of individual statements in a program are obtained. From these the functions of the control structures are obtained. The function of the entire program is formed by combining the functions of its control structures. In this thesis, Wordsworth's idea is applied to a subset of the C programming language[14]. The goal is to provide a thorough theoretical account ranging from obtaining abstractions of a single construct in the program, up to extraction of an abstract functional specification. In particular, the following points summarize the findings reported in this thesis:

- Five abstractions were identified that describe various aspects of a C construct (Section 3).

- Abstraction rules for finding the abstractions of any construct from a subset of C (described in Section 5.1) from the smaller constructs that it contains. These rules are introduced in two steps: A set of basic formulas that aggregate various kinds of abstractions are identified (Section 4); The abstraction rules for each kind of C construct in the specified subset of C are derived, making extensive use of the basic formulas (Section 5).

- The steps that one may take in order to obtain a functional specification in Z notation from a program written in the specified subset of C using the five abstractions (Section 6).

- A theoretical account on how to obtain a more abstract specification from one that is created using the previous steps (Section 7).

Wordsworth himself has pointed out a major difficulty with this approach, that a specification obtained in this manner is likely to specify uninteresting behaviors of a program in addition to the interesting ones [42, p.236]. For example, since a binary search procedure is unlikely to fail even if the list to be searched is unsorted, much of the abstraction of such procedure would describe the behavior of the procedure when the list is unsorted. In fact, by considering that procedure in isolation, it is impossible to deduce the precondition that the procedure is not applicable to an unsorted list. Either human or artificial intelligence is necessary to distinguish between interesting and uninteresting behaviors. This thesis formalizes the assertion of such distinction in terms of specifying boundaries of the abstract operations within the program text, and the state invariants that the program variables must observe at these boundaries.

2 Elements of a Z Specification

This section describes the subset of Z notation[33, 42, 24] that is used in this thesis. The exposition assumes familiarity with predicate logic and set theory.

A Z specification consists of a sequence of declarations interleaving with informal descriptions. Each declaration declares a variable, or a set of variables to have certain type(s) and value(s). Each variable in a specification must be declared before use. Recursive definition is permissible in axiomatic and generic definitions,

but mutually recursive definitions are forbidden. A type in Z is defined to be equivalent to its *carrier set*, that is, the set of all values belonging to that type [33, p.24]. For the purpose of this thesis, we only consider the *value* that a variable may hold as the variable is introduced via one of the following five kinds of declarations: variable declarations, syntactic equivalences, axiomatic definitions, generic definitions, and schema declarations. We first explain the notation for syntactic equivalence, axiomatic definitions and generic definitions. Schema declarations will be introduced after notations for forming expressions and predicates are described. Finally, we discuss how a Z specification may be used to document the functional requirements of a software.

2.1 Variable Declarations

The following *variable declaration*

$$Var : Set \tag{1}$$

introduces the variable *Var* to be an element of the set *Set*, with the value of *Var* further constrained by the context in which the declaration is in effect, *i.e.*, the *scope* of the declaration. For example, the declaration

$$i : \mathbb{N} \tag{2}$$

by itself specifies i to be a natural number (*i.e.*, a non-negative integer). The predicate $i < 10$ within the scope of the declaration would additionally constrain i to be less than ten. A variable declaration of form (1) is frequently used in other declarations, expressions, and predicates.

2.2 Syntactic Equivalence

Syntactic equivalences are used in this thesis to define sets. A *syntactic equivalence* has the following notation:

$$Var == Expression \tag{3}$$

It declares *Var* to be equal to the value specified by *Expression*. *Var* becomes a *global variable*, which may be used in place of *Expression* as a free variable of any subsequent declarations.

A syntactic equivalence may also be *parameterized*. For example, in the following syntactic equivalence

$$X \leftrightarrow Y == \mathbb{P}(X \times Y) \tag{4}$$

X and Y are the *formal parameters* of the declaration. Given this declaration, the expression $\mathbb{Z} \leftrightarrow \mathbb{N}$ may be interpreted as

$$\mathbb{Z} \leftrightarrow \mathbb{N} == \mathbb{P}(\mathbb{Z} \times \mathbb{N}) \tag{5}$$

2.3 Axiomatic Definition

The notation for an axiomatic definition is:

$$\begin{array}{|l} \mathbf{D} \\ \hline \mathbf{P} \end{array}$$

D consists of a set of variable declarations. Predicate **P** further constrains the values of the variables declared in **D**. The variables declared in **D** are global, and therefore may be used as free variables in **P**, and in subsequent declarations.

2.4 Generic Definition

A generic definition is very similar to an axiomatic definition, except that the definition is *parameterized*. For example,

$$
\begin{array}{|l}
\hline
[X] \\
\hline
NonEmpty : \mathbb{P}\, X \\
\hline
\forall s : \mathbb{P}\, X \bullet s \neq \varnothing \\
\hline
\end{array}
$$

declares *NonEmpty* to be the power set of some set; X is the *formal parameter* of the generic definition. For example, if x is of type $\mathbb{Z}$ (set of integers), the predicate

$$x \in NonEmpty[\mathbb{Z}] \tag{6}$$

is valid, with formal parameter X in the above generic definition *instantiated* with *actual parameter* $\mathbb{Z}$. The expression $NonEmpty[\mathbb{Z}]$ in (6) may simply be written as *NonEmpty*. This is because, from the context in which the expression occurs, $NonEmpty[\mathbb{Z}]$ must have type $\mathbb{Z}$. Therefore the respective formal parameter must be $\mathbb{Z}$. Generic definitions are most useful for defining *polymorphic* operators (that is, operators that may take multiple types of arguments), as used frequently in this thesis. For example, the following generic definition

$$
\begin{array}{|l}
\hline
[X] \\
\hline
\_[\_] : \operatorname{seq} X \times \mathbb{N} \pfun X \\
\hline
\forall s : \operatorname{seq} X; i : \mathbb{N} \bullet s[i] = s(i+1) \\
\hline
\end{array}
$$

defines an operator ([ ]) that is a partial function with two arguments. The function may then be used in subsequent declarations with the syntax $e_1[e_2]$, where e_1 and e_2 are expressions of compatible types according the declaration above. The underscores (_) in the above declaration act as *place-holders* for the arguments. By using such place-holders, infix, postfix and prefix operators may be defined at ease. If there is no place-holder for arguments, the operator is by default prefix.

2.5 Expressions and Predicates

Z notation is based on *typed set theory* and *first order predicate logic.* In Z notation, expressions specify *atoms* (values that are not sets themselves) and sets. Predicates are used to assert statements about expressions.

2.5.1 Expressions

The kinds of *expressions* in Z notation used in this thesis may be classified into the following categories: variables, tuples, sets, definite descriptions, function applications, sequences, and schema component selection expressions. Tuples are always atoms. Even though variables, definite descriptions, and function applications may specify sets, they are considered separately from sets because they may also specify atoms. Although sequences are sets in Z notation, it is worthwhile to discuss them separately from sets because of their distinct role as specifying *ordered* lists of values in a specification. We concern ourselves only with the value of an expression rather than its type.

variables When a variable occurs in an expression, its value is obtained from its declaration. We have considered how this value is obtained from a variable declaration, syntactic equivalence, axiomatic definition, or generic definition. The remaining case is when the variable is declared using a schema, which we shall consider in Section 2.7.1.

tuples A *tuple* is an ordered collection of values, enclosed by brackets, and separated by commas. For example, (e_1, e_2) is a tuple whose first component is expression e_1, and whose second component is expression e_2. Two projection operators are available in Z for *ordered pairs* (tuples with two components): **first** p gives the

first component of an ordered pair p, and **second** p gives the second component of an ordered pair p.

sets The following notations in Z are used in this thesis for defining sets: predefined set, set operators (power set, cross product, relation, functions, union, intersection, and minus) set enumeration, and set comprehension. These notations are described as follows:

predefined sets Z notation provides a number of predefined sets, including integers ($\mathbb{Z}$), and empty set ($\varnothing$). The set of natural number ($\mathbb{N}$) is not a predefined set according to the Standard[24], but is included as part of the mathematical toolkit associated with the Z[33, p.108].

power sets The *power set* of a set S is denoted in Z notation by $\mathbb{P}\, S$.

cross products The *cross product* of expressions e_1, e_2, ..., e_n, each of them specifies a set, is the set of all tuples whose first component is an element of e_1, and whose second component is an element of e_2, and so on. This is denoted in Z by the notation $e_1 \times e_2 \times \cdots \times e_n$.

relations and functions A *relation* is a mapping from one set to another. For sets X and Y, the notation $X \leftrightarrow Y$ represents the set of all relations from X to Y. Every relation from X to Y corresponds to a subset of $X \times Y$, and *vice versa*. Therefore, the set of all relations from X to Y would be the set $\mathbb{P}(X \times Y)$. This may be captured by the following syntactic equivalence:

$$X \leftrightarrow Y == \mathbb{P}(X \times Y) \tag{7}$$

Several operators are available in Z for relations. The prefix functions dom and ran evaluates the *domain* and *range* of a relation. They may be defined by the following generic definition:

$$
\begin{array}{|l}
\hline
[X, Y] \\
\mathrm{dom} : (X \leftrightarrow Y) \rightarrow \mathbb{P}\, X \\
\mathrm{ran} : (X \leftrightarrow Y) \rightarrow \mathbb{P}\, Y \\
\hline
(\forall r : X \leftrightarrow Y \bullet \mathrm{dom}\, r = \{x : X; y : Y \mid (x, y) \in r \bullet x\} \\
\land\ (\forall r : X \leftrightarrow Y \bullet \mathrm{ran}\, r = \{x : X; y : Y \mid (x, y) \in r \bullet y\} \\
\hline
\end{array}
$$

In addition, there are infix functions available for performing operators such as *domain restriction* ($\lhd$), *domain subtraction* ($\lhd\!\!\!-$), *range restriction* ($\rhd$), and *range subtraction* ($\rhd\!\!\!-$). Each of these operators takes a pair of relations as arguments and evaluates to another relation. They may be defined by the following generic definitions:

$$
\begin{array}{|l}
\hline
[X, Y] \\
\_ \lhd \_ : (\mathbb{P}\, X \times (X \leftrightarrow Y)) \rightarrow (X \leftrightarrow Y) \\
\_ \lhd\!\!\!- \_ : (\mathbb{P}\, X \times (X \leftrightarrow Y)) \rightarrow (X \leftrightarrow Y) \\
\hline
(\forall x : \mathbb{P}\, X; r : X \leftrightarrow Y \bullet \\
\quad (x \lhd r = \{p : X; y : Y \mid p \in x \land (p, y) \in r \bullet (p, y)\}) \\
\quad \land\ (x \lhd\!\!\!- r = \{p : X; y : Y \mid p \notin x \land (p, y) \in r \bullet (p, y)\})) \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
[X, Y] \\
\_ \rhd \_ : ((X \leftrightarrow Y) \times Y) \rightarrow (X \leftrightarrow Y) \\
\_ \rhd\!\!\!- \_ : ((X \leftrightarrow Y) \times Y) \rightarrow (X \leftrightarrow Y) \\
\hline
(\forall y : \mathbb{P}\, Y; r : X \leftrightarrow Y \bullet \\
\quad (r \rhd y = \{x : X; p : Y \mid p \in y \land (x, p) \in r \bullet (x, p)\}) \\
\quad \land\ (r \rhd\!\!\!- y = \{x : X; p : Y \mid p \notin y \land (x, p) \in r \bullet (x, p)\})) \\
\hline
\end{array}
$$

Functions in Z are special forms of relations. Specifically, a *function* is a relation in which each element of its domain is mapped to a single element of its range.

The notation $X \pfun Y$ denotes the set of all *partial functions* from X to Y, that is, the set of all relations from X to Y which are functions. This may be specified in Z by

$$X \pfun Y == \{r : X \rel Y \mid (\forall x : X; y_1, y_2 : Y \mid x \in \dom r \bullet ((x, y_1) \in r \land (x, y_2) \in r) \Leftrightarrow y_1 = y_2\} \tag{8}$$

The notation $X \fun Y$ denotes the set of all *total functions* from X to Y, that is, the set of all partial functions from X to Y whose domain is X. This may be specified in Z by

$$X \pfun Y == \{r : X \fun Y \mid \dom r = X\} \tag{9}$$

There is an important infix operator known as *overriding*, denoted by $\oplus$. Given two functions of the same type as its (left and right) operands, this operator constructs another function by overriding each ordered pair in the first operand by the ordered pair in the second operand that has common first component, if there is any. The resulting function may formally be specified as

$$[X, Y]$$
$$\_ \oplus \_ : (X \pfun Y) \times (X \pfun Y) \fun (X \pfun Y)$$
$$\forall f, g : X \pfun Y \bullet f \oplus g = (\dom g \domsub f) \cup g$$

other set operators Usual set operators, *union* ($\cup$), *intersection* ($\cap$), and *minus* ($\setminus$), are available in Z as infix operators.

set enumerations A *set enumeration* defines a set by explicitly listing the elements of the set. This is denoted in Z by the following notation:

$$\{e_1, e_2, \ldots, e_n\} \tag{10}$$

which denotes a set whose elements are the values of the expressions e_1, e_2, ..., and e_n.

set comprehensions A *set comprehension* defines a set by the property that all elements of the set must satisfy. This is denoted in Z by the following notation:

$$\{\mathbf{D} \mid \mathbf{P} \bullet \mathbf{E}\} \tag{11}$$

D is a set of variable declarations. **P** is a predicate. **E** is an expression. The set (11) is constructed in the following fashion: For every sets of values of the variables declared in **D** such that **P** is satisfied, The corresponding value of **E** becomes an element of the set being constructed. A special form of set comprehension is $\{v : T \mid \mathbf{P}\}$, which is equivalent to $\{v : T \mid \mathbf{P} \bullet v\}$.

integer subranges A special kind of set comprehension is one that specifies a subrange of integers. In particular, the set of integers ranging from a to b may be denoted in Z by the following set comprehension:

$$a..b == \{n : \mathbb{Z} \mid a \leq n \leq b\} \tag{12}$$

definite descriptions A *definite description* has the following form.

$$(\mu \mathbf{D} \mid P_D(\overline{v}, \overline{x}) \bullet \mathbf{E}) \tag{13}$$

where **D** is a set of variable declarations. **E** is an expression. P_D is the *constraining predicate* that gives values to the variables declared in **D** (denoted by $\overline{v}$ in (13)), by specifying the relationship among these variables and the free variables of the description (denoted by $\overline{x}$ in (13)). The corresponding value of **E** (usually in terms

of $\overline{v}$) becomes the value of the definite description. The use of the variable names $\overline{v}$ is arbitrary because these variables are *bounded* within the definite description. Free variables $\overline{x}$ must be declared in the context in which the definition description occurs in a specification. If P_D does not constraint $\mathbf{E}$ to a unique value, the value of the definite description is taken as being one of the alternatives, but the selection of alternative is indeterminate[42, p.88]. If P_D does not constrain $\mathbf{E}$ to any value at all, the value of the definite description is still taken as being some value of its type, but is completely indeterminate. A special form of definite descriptions that is used throughout this thesis is

$$(\mu \nu_1 : T \mid P_D(\nu_1, \overline{x})) \tag{14}$$

which is equivalent to

$$(\mu \nu_1 : T \mid P_D(\nu_1, \overline{x}) \bullet \nu_1) \tag{15}$$

function application The *prefix form* of *function application* used in this thesis is $e_1(e_2)$, where expression e_1 is a function and e_2, the *argument* of the function application, is another expression By definition, the function application $e_1(e_2)$ may evaluate to at most one value (none if e_2 is not in the domain of e_1). Therefore function application may be specified using definite description, using the following generic definition:

$$\begin{array}{|l} [X, Y] \\ \hline \_(\_) : ((X \pfun Y) \times X) \fun Y \\ \hline (\forall f : X \pfun Y; x : X \bullet f(x) = (\mu \nu_1 : Y \mid (x, y) \in f)) \\ \hline \end{array}$$

sequences A *sequence* of X may be represented in Z by a partial function from natural numbers to X [33]. In particular, the set of all sequences whose elements are drawn from the set X (the formal parameter in the following declaration) may be declared as

$$\operatorname{seq} X == \{f : \mathbb{N} \pfun X \mid \operatorname{dom} f = 1..\#f\} \tag{16}$$

To index an item from a sequence, function application may be used. For example, the i^{th} element of a sequence s may be referenced by the function application $s(i)$. The following prefix operators are defined for sequences: For a sequence s, **head** s gives the first element of s, **last** s gives the last element of s, **front** s gives a sequence that is the same as s except with its last element truncated, and **tail** s gives a sequence that is the same as s except with its first element removed. These operators may be declared by the following generic definition:

$$\begin{array}{|l} [X] \\ \hline \mathbf{head} : \operatorname{seq} X \to X \\ \mathbf{last} : \operatorname{seq} X \to X \\ \mathbf{front} : \operatorname{seq} X \to \operatorname{seq} X \\ \mathbf{tail} : \operatorname{seq} X \to \operatorname{seq} X \\ \hline (\forall s : \operatorname{seq} X \bullet \mathbf{head} s = s(1) \\ \quad \land \mathbf{last} s = s(\#s) \\ \quad \land \mathbf{front} s = \{\#s\} \ndres s \\ \quad \land \mathbf{tail} s = \{1\} \ndres s \\ \hline \end{array}$$

Note that since a sequence is a function in Z, cardinality operator (#) applies equally to sequences, yielding their lengths.

A *sequence enumeration* specifies a sequence by listing its elements, as shown below:

$$\langle e_1, e_2, e_3, \ldots, e_n \rangle \tag{17}$$

2.5.2 Predicates

There are four ways of forming predicates: using relational operators among expressions, using logical connectives, quantifications, and schemas expressions. Schema expressions as predicates will be introduced in Section 2.7.3.

relational operators Z does not have an explicit boolean type. So there is no simple predicate which equates a variable or an expression to a boolean constant. Instead, all relational expressions which evaluate to truth values are treated as predicates. For example, the equality operator applied to two operands of the same type such as

$$\mathbf{X} = \mathbf{Y} \tag{18}$$

is a predicate. The truth value of such a predicate depends on the semantics of the relational expression (in the above example, the predicate evaluates to true when the expression $\mathbf{X}$ is equal to $\mathbf{Y}$). Almost all the conventional relational operators are available in Z. Some of them are: set membership ($\in$), subset operators ($\subset$, $\subseteq$), and comparative operators ($<$, $\leq$, $>$, $\geq$, $=$, $\neq$).

logical connectives Compound predicates can be created using logical connectives: negation ($\neg$), disjunction ($\vee$), conjunction ($\wedge$), implication ($\Rightarrow$), and equivalence ($\Leftrightarrow$).

quantification Both existential and universal quantifications are used in this thesis. In Z notation, an existential quantification is denoted by

$$(\exists \mathbf{D} \mid \mathbf{P}_1 \bullet \mathbf{P}_2) \tag{19}$$

where $\mathbf{D}$ is a set of variable declarations. $\mathbf{P}_1$ and $\mathbf{P}_2$ are predicates. Let $v_1, v_2, \ldots, v_n$ be the variables declared in $\mathbf{D}$. Predicate (19) asserts that 'there exists a set of values for $v_1, v_2, \ldots, v_n$, constrained by $\mathbf{P}_1$, such that $\mathbf{P}_2$ is satisfied'. Therefore, predicate (19) is equivalent to

$$(\exists \mathbf{D} \bullet \mathbf{P}_1 \wedge \mathbf{P}_2) \tag{20}$$

A universal quantification is denoted in Z by

$$(\forall \mathbf{D} \mid \mathbf{P}_1 \bullet \mathbf{P}_2) \tag{21}$$

where $\mathbf{D}$ is a set of variable declarations of form (1). $\mathbf{P}_1$ and $\mathbf{P}_2$ are predicates. Let $v_1, v_2, \ldots, v_n$ be the variables declared in $\mathbf{D}$. Predicate (21) asserts that 'for all sets of values for $v_1, v_2, \ldots, v_n$ such that $\mathbf{P}_1$ is true, $\mathbf{P}_2$ is satisfied'. Therefore, predicate (21) is equivalent to

$$(\forall \mathbf{D} \bullet \mathbf{P}_1 \Rightarrow \mathbf{P}_2) \tag{22}$$

2.6 Schemas Declarations

A *schema declaration* has the following form:

$$\mathit{SchemaVar} \mathrel{\widehat=} [\mathbf{D} \mid \mathbf{P}] \tag{23}$$

which declares *SchemaVar* to be the name of the schema whose definition is given on the right hand side of the definition symbol ('$\mathrel{\widehat=}$'). The *vertical bar* (|) divides the schema into two parts. The *signature part*, $\mathbf{D}$, is a set of declarations, each of them is either a variable declaration or a schema reference. The variables declared in $\mathbf{D}$ are known as the *components* of the schema. These variables are *local*

to *SchemaVar*, but are available to subsequent schemas declarations by schema inclusion or schema references. The *predicate part* **P** is a predicate that constrains the values of the components of the schema. Presence of a schema reference in the signature part of another schema is known as schema inclusion. Schema inclusion may be defined as follows: *Inclusion* of schema A in schema B results in another schema whose signature part is the union of the signature parts of A and B, and whose predicate part is the logical conjunction between the predicate parts of A and B.

Three kinds of schema declarations are used in this thesis: *state schemas*, *operation schemas*, and *initial state schemas.*

2.6.1 State Schemas

In a model-based specification, a system is modeled by a persistent data store, with each operation specified in terms of a change of *state*, that is, a change in the values of the data store. A schema, in this case known as a *state schema*, may be used to specify the form of the data store (that is, the *model*) for the system. Specifically, for a schema that specifies a model, its components represent the *state* of the system at some point during execution of the system. The predicate part represents the *state invariant* that must be held by the values of the persistent data at the beginning and at the end of every operation.

2.6.2 Operation Schemas

A schema, when used to specify an operation, is known as an operation schema. Syntactically, an operation schema has the same structure as that of any other schema. However, the declaration part of an operation schema must include the state space(s) on which the operation will be performed. Typically, the predicate of an operation schema describes the state changes. Following Oxford's convention

of writing Z specifications, *primed components* (components whose names are *decorated*[5] with a prime) represent the final state after the operation. The unprimed counterparts of these variables represent the initial state before the operation. The predicate part of the operation schema represents the relationship between the initial and final state of the operation. For the purposes of this thesis, we may define *precondition operator* **pre**, which gives all initial states of an operation that have corresponding final states, and *postcondition operator* **post**, which gives all final states of an operation that have corresponding initial states. Formally, for the following operation schema.

$$Op \mathrel{\widehat=} [\overline{v}, \overline{v}' \mid P_\epsilon(\overline{v}, \overline{v}')] \tag{24}$$

pre *Op* is a schema whose components are those of *Op* that are unprimed, and predicate part being

$$\mathbf{pre}\, \epsilon[\![\mathsf{C}]\!] \Leftrightarrow (\exists \overline{v}' \bullet P_\epsilon(\overline{v}, \overline{v}')) \tag{25}$$

Similarly, **post** *Op* is a schema whose components are those of *Op* that are primed, and predicate part being

$$\mathbf{post}\, \epsilon[\![\mathsf{C}]\!] \Leftrightarrow (\exists \overline{v} \bullet P_\epsilon(\overline{v}, \overline{v}')) \tag{26}$$

delta convention Consider the following schema

$$S_1 \mathrel{\widehat=} [x : \mathbb{N}; y : \mathbb{Z} \mid y > x] \tag{27}$$

An operation *Op* may be defined on S_1 by the following declaration:

$$Op \mathrel{\widehat=} [S_1; S_1' \mid x < y - 1 \land x' = x + 1 \land y' = y] \tag{28}$$

[5]for example, n', whose unprimed counterpart is n, is a variable decorated with a prime.

which, by schema inclusion, is equivalent to

$$Op \mathrel{\widehat=} [x, x' : \mathbb{N};\ y, y' : \mathbb{Z} \mid x < y - 1 \land x' = x + 1 \land y' = y] \tag{29}$$

We may use Δ-*convention* to abbreviate (28) by

$$Op \mathrel{\widehat=} [\Delta S_1 \mid x < y - 1 \land x' = x + 1 \land y' = y] \tag{30}$$

xi convention When an operation does not result in a state change, Ξ-*convention* may be used to abbreviate the operating schema. Once again, consider the schema S_1 in (27):

$$S_1 \mathrel{\widehat=} [x : \mathbb{N};\ y : \mathbb{Z} \mid y > x] \tag{31}$$

An operation $Op2$ may be defined on S_1 by the following declaration:

$$Op2 \mathrel{\widehat=} [S_1;\ S_1' \mid x < y - 1 \land x' = x \land y' = y] \tag{32}$$

which, by schema inclusion, is equivalent to

$$Op2 \mathrel{\widehat=} [x, x' : \mathbb{N};\ y, y' : \mathbb{Z} \mid x < y - 1 \land x' = x \land y' = y] \tag{33}$$

We may use Ξ-*convention* to abbreviate (32) by

$$Op2 \mathrel{\widehat=} [\Xi S_1 \mid x < y - 1] \tag{34}$$

Note that the predicate $x' = x \land y' = y$, which indicates that there is no state change, is implied by Ξ-convention.

2.6.3 Initial State Schemas

An operation schema specifies the change of state during execution. However, we also need to specify the conditions during initialization process. This is specified by an initial state schema which is considered to be a special operation on the state. In particular, the initial state of a system may be represented by an operation that specifies only its final state.

Suppose S specifies the model of a system. The initial state of the system may then be specified by the following *initial state schema*:

$$Init\_S \mathrel{\widehat=} [S' \mid \mathbf{P}] \tag{35}$$

This schema looks very similar to an operation schema that specifies an operation on S, with the exception that S is absent from the declaration part. This is because the state before initialization is irrelevant to the result of initialization.

2.7 Schema References

A *schema reference* is either a single schema name, or an 'expression' formed by applying schema operators to one or more schemas. A schema reference may either specify a set, a predicate, or an atom.

2.7.1 Schemas as Sets

We shall describe how schemas specify sets by an example. Consider the schema in (27):

$$S_1 \mathrel{\widehat=} [x : \mathbb{N}; y : \mathbb{Z} \mid y > x] \tag{36}$$

The *expression* S_1 specifies the set of all values for x and y such that $y > x$:

$$\{(x \mapsto 0, y \mapsto 1), (x \mapsto 0, y \mapsto 2),$$

$$\ldots, (x \mapsto 1, y \mapsto 2), (x \mapsto 1, y \mapsto 3), \ldots\} \tag{37}$$

The notation $(name_1 \mapsto value_1, name_2 \mapsto value_2, \ldots)$ denotes a value with multiple components, similar to a tuple, except that (i) each component is named, and (ii) the ordering of the components is insignificant. For example, $(x \mapsto 1, y \mapsto 2)$ is exactly the same as $(y \mapsto 2, x \mapsto 1)$, while $(a \mapsto 1)$ is different from $(b \mapsto 1)$.

If S_1 specifies the model of a system, then the expression S_1 would specify the set of *all* states that the system can possibly attain before and after executing any operation. Note that specific operations may further constrain the set of possible states before and/or after their execution.

Since the expression S_1 specifies a set, the variable declaration

$$s : S_1 \tag{38}$$

is valid, which declares s to be an element of the set specified by S_1. Then s would be a value with two components named x and y. To refer to the values of these components, we use the *schema component selection expressions* $s.x$ and $s.y$.

2.7.2 Schema Component Selection Expressions

Let s be a variable of a *schema type*. Then, the value of a component of s, say c, is referenced by the *schema component selection expression* $s.c$.

2.7.3 Schemas as Predicates

When a schema occurs in a specification where a predicate is required, the predicate part of the schema is substituted in place of the schema. The free variables introduced must have been declared in the context in which the schema occurs.

A number of schema operators are used in this thesis. We have seen two of them, namely **pre** and **post** operators. Except for theta convention, which specifies

an atom, all of these operators yield schemas as results. Now, we shall include the definitions for other schema operators that are used in this thesis: decoration, conjunction, disjunction, theta convention, renaming, systematic renaming, hiding, and composition.

2.7.4 Schema Decoration

For a schema A, A' is a schema which has the same declaration and predicate parts, except that the components of the schema, as well as their free occurrences in the predicate part, are decorated with a prime.

2.7.5 Logical Connectives Applied to Schemas

All the five logical operators can be applied to schemas as operands. In all cases, the declarations of the operands are merged and their respective predicate parts are connected using the logical connectives. For example, if $S_1 \mathrel{\widehat=} [D_1 \mid P_1]$ and $S_2 \mathrel{\widehat=} [D_2 \mid P_2]$, then $S_1 \Rightarrow S_2 \mathrel{\widehat=} [D_1; D_2 \mid P_1 \Rightarrow P_2]$.

2.7.6 Theta Convention

For any schema A, if $c_1, c_2, \ldots, c_n$ are the components of A, then θA denotes the following:

$$(c_1 \mapsto c_1, c_2 \mapsto c_2, \cdots c_n \mapsto c_n) \tag{39}$$

with the constraint that $c_1, c_2, \ldots, c_n$ must satisfy the predicate part of A. The notation $\theta A'$ denotes the following quantity:

$$(c_1 \mapsto c_1', c_2 \mapsto c_2', \cdots c_n \mapsto c_n') \tag{40}$$

with the constraint that $c_1', c_2', \ldots, c_n'$ must satisfy the predicate part of A. The application of theta operator to a schema results in an unnamed instance of the

schema whose bindings satisfy the constraints of the predicate part. More about bindings can be found in [33, p.26].

2.7.7 schema components renaming

For a schema A, renaming of components n_1 to m_1, n_2 to m_2, ... n_k to m_k is denoted by

$$A[m_1/n_1, m_2/n_2, \ldots m_k/n_k] \tag{41}$$

2.7.8 systematic renaming

In another form of renaming, rather than supplying the names of all the components to be renamed, only the common decorator of them are supplied. In effect, all components that have a certain decorator, are renamed to components of the same name except having another decorator. For a schema A, the notation

$$A[\_'/\__0] \tag{42}$$

denotes renaming of all components of A that has subscript '$_0$' to their primed counterparts.

2.7.9 hiding

The *hiding* of a list of components n_1, n_2, ..., n_k from a schema A is denoted by

$$A \setminus (n_1, n_2, \ldots, n_k) \tag{43}$$

The result is another schema whose components are those of A, with n_1, n_2, ..., n_k removed, and declaration part becomes the following predicate

$$(\exists \mathbf{D} \bullet P \wedge I) \tag{44}$$

where $\mathbf{D}$ represents those variable declarations in A that correspond to the hidden components, and P is the predicate part of A.

2.7.10 composition

For any two schemas A and B such that the declarations for primed variables in A match exactly with the declarations for their unprimed counterparts in B, *schema composition* of A and B results in another schema whose components are the unprimed components of A and primed components of B, and predicate part is

$$(\exists \mathbf{D}_0 \bullet A[\__0/\_'] \land B[\__0/\_]) \tag{45}$$

where $\mathbf{D}_0$ are the matching declarations between A and B, with variables decorated with another decorator '0'.

2.8 Partial Systematic Renaming

Certain abstraction rules in this thesis require a renaming operator somewhat different from the systematic renaming that has just been introduced. In that case, we use a *partial systematic renaming* operator. This operator may be applied to any predicate and any expression within the predicate part of a schema. Specifically, for each free variable in the predicate, the renaming is performed if and only if both its old name and its corresponding new name are declared identically in the signature part of the schema in which the renaming occurs. The result of partial systematic renaming is a schema with the same signature. For example,

$$[i, i', j, j' : \tau[\mathbf{int}] \mid (i' = i; j' = k(i))_{[\_'/\_]}] \tag{46}$$

would be the schema

$$[i, i', j, j' : \tau[\mathbf{int}] \mid (i' = i'; j' = k(i'))_{[\_'/\_]}] \tag{47}$$

Note that k, not declared in the signature of (46), is not renamed.

2.9 Writing Functional Specifications in Z

A *model-based functional specification* of a software system describes an abstract model of the software and the set of functionalities that the software is required to implement. In general, the functional specification should exist as part of the software requirements specification (SRS) of the system. In [42, p.148-9] Wordsworth provided a good introduction on how to use Z notation to write a functional specifications.

The specification should begin with an informal description of its purpose, explanation on the functionalities of the system being specified, assumptions made and so on. This section should establish meaningful terminologies relating to those functionalities, which is to be made precise in the formal specification.

The second part of the specification should include declarations (syntactic equivalences, axiomatic or generic definitions) of global variables and operators that are used throughout the remainder of the specification. Each variable or operator should be accompanied by informal text explaining its rationale (such as the application domain element the variable or operator corresponds to) and its use in subsequent declarations.

The third part of the specification would consist of a sequence of declarations that specifies the model as well as the operations. Each declaration in the third part should be accompanied with informal text explaining the aspect(s) of the system modeled by the declaration, and its relationship to the system as a whole.

In our approach for deriving a specification from code (Section 6), the resulting specification would have a hierarchically organized model. The following table shows an example format of such specification.

PartOne:
Problem Description
Part Two:
Global variables, operators
Part Three:
Partial Model A
Initial State A
Operations on A

Partial Model B
Initial State B
Operations on B

PartialModel C
(includes A and B)
Initial State C
Operations on C

Partial Model D
Initial State D
Operations on D

Partial Model E
Initial State E
Operations on E

Partial Model F
(includes D and E)
Initial State F
Operations on F

Complete Model G
(includes C and F)
Initial State G
Operations on G

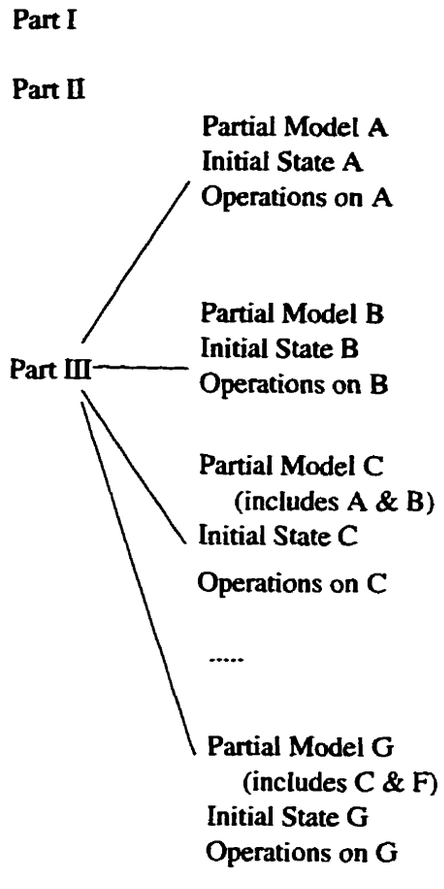

Figure 1: A functional specification with hierarchically organized model

For a hierarchically organized model, it is a good idea to justify the organization

of the model at the beginning of the third part of the specification.

3 Overview of the Abstractions

This thesis describes a reverse engineering process in which a functional requirements specification, including a model and a set of functionalities, is recovered from the code of a software. This section introduces a set of abstractions which are useful for such recovery from a C program. These abstractions capture the pre and postconditions in executing a program, or the constructs that the program contains.

We shall define C *constructs* to be certain *syntactic elements* in the C language[6]. One construct may syntactically enclose another. In this case we say that a *primitive construct* is *enclosed* by an *enclosing construct*. Two constructs may also *interleave* with one another, when they share common code but neither one encloses the other. In this case these constructs are referred to as *interleaving constructs*.

Five abstractions, defined below, are used in the reverse engineering process described in this thesis:

3.1 Abstract Type

The *abstract type* of a construct C, denoted by $\tau[\![C]\!]$, is an abstraction of the *type* that may be associated with the construct in a program. Let us consider the following four kinds of association between a construct and a type:

type expressions A *type expression* is a construct that specifies a type in C language. We shall define the abstract type of a type expression in C to be the set, in Z notation, that corresponds to all the values represented by that type. For

[6]For the list of constructs in the subset of C considered in this thesis please refer to Section 5.1.

example, the type **int** in C may be represented by the set *Min..Max* in Z, with *Min* and *Max* being the minimum and maximum integers represented by **int**. We specify this as

$$\tau[\![\mathbf{int}]\!] == Min..Max \tag{48}$$

which is an abbreviation for

$$\tau[\![\mathbf{int}]\!] == \{n : \mathbb{Z} \mid Min \leq n \leq Max\} \tag{49}$$

variable definition A *variable definition* in C is a construct in which a variable is associated with a type. We shall define the abstract type of a variable definition to be the abstract type of the type expression in the variable definition that specifies the type of the variable being defined.

object designated by an expression An expression may designate an *object*, defined to be a memory storage that contains a value of a certain type . In this case, we shall define the abstract type of the expression to be the abstract type of the type associated with the object that it designates.

result value of an expression We shall define the *result type* of an expression to be the type of the value obtained from evaluating the expression as if it is on the right hand side of an assignment expression. The result type of an expression will be captured by its *abstract value* (to be defined in Section 3.4). Therefore we shall not define the abstract type of an expression that does not designate an object. It should be noted that the type of the object designated by an expression needs not be the same as its result type. For example, if variable a is defined to be an array

of `int`, then the result type of the expression `a` would be pointer to `int`, while the expression designates an object of type array of `int`.

Following the above analysis, abstract types are defined only for types expressions, variable definitions, and expressions that designate objects. The purpose of this abstraction is to assign types to all variables throughout the specification that is being recovered.

3.2 Abstract State

The *abstract state* of a construct C, denoted by $\sigma[\![C]\!]$, is an abstraction of the collection of variables on which the construct operates. These variables are referred in this thesis to as '*the variables of the construct*'. For example, to evaluate the expression `a[i]`, we first retrieve the address of `a` and then index this address by the result value of evaluating `i`. We say that the construct `a[i]` *operates* on variables `a` and `i`.

Abstract state is applicable to any executable construct (that is, any construct other than type expressions, since they do not operate on variables). Variables and function definitions may be classified as executable because they both imply initialization of variables. For uniformity, we shall define the abstract state of a constant in an expression to be an empty schema, that is, a schema whose signature part is empty and predicate part the predicate *true*. Let $v_1, v_2, \ldots, v_n$ be the variables of a construct C, with their types defined in the program to be $T_{v_1}, T_{v_2}, \ldots,$ and T_{v_n} respectively. In other words, each variable v_i, where $1 \leq i \leq n$, represents an object of type T_{v_i}. These variables may be represented in a Z specification by the variables[7] $v_1, v_2, \ldots, v_n$. Each variable v_i, for $1 \leq i \leq n$, must be a member of the set $\tau[\![T_{v_i}]\!]$. The abstract state of C may then be represented

[7] variable names that are reused in a C program at different scopes are required to be represented by unique names in the specification

in Z by a state schema of the following form:

$$\sigma[\![\mathsf{C}]\!] \mathrel{\widehat=} [v_1 : \tau[\![\mathsf{T}_{v_1}]\!]; v_2 : \tau[\![\mathsf{T}_{v_2}]\!]; \ldots; v_n : \tau[\![\mathsf{T}_{v_n}]\!] \mid Inv(v_1, v_2, \ldots, v_n)] \quad (50)$$

The signature part of the schema represents the variables of the construct and their respective types. The predicate part of the schema represents the invariant that must be true among the variables of the construct *throughout the program*. Since one may not define a state invariant for any construct in C (other than type constraints on program variables), we may assign the predicate *true* to the state invariant of any construct. Definition (50) may be abbreviated by

$$\sigma[\![\mathsf{C}]\!] \mathrel{\widehat=} [\overline{v} \mid Inv(\overline{v})] \quad (51)$$

Note here that we have overloaded the abbreviation $\overline{v}$ to denote both a set of declarations and a list of variables in a schema. Abstract states are useful for building an abstract model of a software from its code.

3.3 Abstract Effect

The *effect* of executing a construct may be defined in terms of the pre and post-conditions of its variables under normal execution. This precisely captures the functionality of the construct. We shall define the *initial state* and *final state* of the construct respectively to be consist of the values of the variables of the construct at the beginning and the end of its execution. The *precondition* of a construct is defined to be the condition that must hold in its initial state for the construct to execute normally (*i.e.*, the construct terminates without error). The *postcondition* of a construct is defined to be the relationship between the initial and final states of the construct, provided that the precondition is satisfied.

The *abstract effect* of a construct C, denoted by $\epsilon[\![\mathsf{C}]\!]$, is an abstraction of the effect of executing the construct. The abstract effect of C may be represented in

Z by an *operation schema* of the form:

$$\epsilon[\![\mathsf{C}]\!] \mathrel{\widehat=} [\Delta\sigma[\![\mathsf{C}]\!] \mid P_\epsilon(\overline{v}, \overline{v}')] \tag{52}$$

which is equivalent to

$$\epsilon[\![\mathsf{C}]\!] \mathrel{\widehat=} [\overline{v}; \overline{v}' \mid Inv(\overline{v}) \wedge Inv(\overline{v}') \wedge P_\epsilon(\overline{v}, \overline{v}')] \tag{53}$$

In the signature part of (53), $\overline{v}$ and $\overline{v}'$ respectively represents the initial and final states of the construct. The predicate part represents the pre and postconditions on the state of the construct. The predicate $Inv(\overline{v}) \wedge Inv(\overline{v}')$ asserts that both the initial and final states must satisfy the state invariant. We may disregard its presence because both $Inv(\overline{v})$ and $Inv(\overline{v}')$ are the predicate *true*. Abstract effects are useful for building specifications for the functionalities (or *abstract operations*) implemented by the program.

3.4 Abstract Value

The *abstract value* of a construct is an abstraction of the *result value* of the construct. This abstraction is applicable only to expressions. Since, in C, the values of the variables of an expression may change as the expression is being evaluated, we may distinguish between the abstract value of C expressed in terms of its initial state (denoted by $\nu[\![\mathsf{C}]\!]$), and that expressed in terms of its final state (denoted by $\nu'[\![\mathsf{C}]\!]$). If the result of a construct is a constant, it may be represented in Z as it is, though in any case, a definite description may be used.

The form of $\nu[\![\mathsf{C}]\!]$ is

$$\nu[\![\mathsf{C}]\!] = (\mu\, \nu_1 : \tau[\![\mathsf{T}]\!] \mid \mathbf{pre}\, \epsilon[\![\mathsf{C}]\!] \wedge P_\nu(\nu_1, \overline{v})) \tag{54}$$

where $\overline{v}$ represents the initial state of C. $\tau[\![\mathsf{T}]\!]$ is the abstract type of the result type of C. $\mathbf{pre}\, \epsilon[\![\mathsf{C}]\!]$ represents the precondition of evaluation that must be satisfied in

order for the result value of the construct to be meaningful. When this condition is satisfied, the predicate P_ν represents the relationship between the initial state of C and the abstract value of C by specifying the relationship between $\overline{v}$ and ν_1. As stated in Section 2, if P_{Value} does not constrain ν_1 to a unique value, the value of (54) is taken as being one of the alternatives, but the selection of alternative is indeterminate. This corresponds to the case when evaluation of the C is non-deterministic. Recall that when P_{Value} does not constrain ν_1 to any value, the value of (54) is indeterminate. This represents the case in which either an error has occurred or the abstract value of C cannot be determined from its initial state.

The form of $\nu'[\![\mathsf{C}]\!]$ is

$$\nu'[\![\mathsf{C}]\!] = (\mu\,\nu_1 : \tau[\![\mathsf{C}]\!] \mid \mathbf{post}\,\epsilon[\![\mathsf{C}]\!] \wedge P_\nu(\nu_1, \overline{v}')) \tag{55}$$

where $\overline{v}'$ represents the final state of C. $\mathbf{post}\,\epsilon[\![\mathsf{C}]\!]$ represents the condition that must hold in the final state of C given normal execution of C. When this condition is true, the predicate P_ν represents the relationship between the final state and the result value of C by the relationship between $\overline{v}'$ and ν_1. Once again, $\tau[\![\mathsf{T}]\!]$ is the abstract type of the result type of C. Interpretation of (55) when P_ν does not constrain ν_1 uniquely is similar to that for (54).

The abstract value of a construct always occurs as parts of the predicates in the abstractions of its enclosing constructs.

3.5 Abstract Object

The *abstract object* of a construct is an abstraction of the object designated by the construct. This abstraction is applicable only to those expressions that designate objects. We shall refer to such expressions as '*object designation expression*'. For example, the construct `a[i]` designates the object which is the `(i+1)`$^{\mathsf{th}}$ element of the array `a`. The object designated by any construct, if there is one, must either

be a variable, or part of a variable (in this case the variable must be of *composite type* such as array or structure). In both cases, we say that the variable is the *parent* of the object.

The object designated by an object designation expression depends not only on the kind of construct it is, but also on the values of its variables. Recall that, as an expression is evaluated, the values of its variables may change. For any construct C that designates an object, $\omega[\![\mathsf{C}]\!]$ shall denote the abstract object of C expressed in terms of its initial state, while $\omega'[\![\mathsf{C}]\!]$ shall denote the abstract object of C expressed in terms of its final state.

In this thesis, we shall only consider the subset of C for which the parent of the object designated by an object designation expression C is fixed regardless of the values of its variables. Let v be that variable, represented by v in the specification. Similar to abstract value, $\omega[\![\mathsf{C}]\!]$ and $\omega'[\![\mathsf{C}]\!]$ may also be stated as definite descriptions, as follows.

$$\omega[\![\mathsf{C}]\!] = (\mu\, \nu_1 : \tau[\![\mathsf{T}]\!] \mid \mathbf{pre}\, \epsilon[\![\mathsf{C}]\!] \wedge P_\omega(\nu_1, \overline{v}, v_o)) \tag{56}$$

$$\omega'[\![\mathsf{C}]\!] = (\mu\, \nu_1 : \tau[\![\mathsf{T}]\!] \mid \mathbf{post}\, \epsilon[\![\mathsf{C}]\!] \wedge P_\omega(\nu_1, \overline{v}', v_o)) \tag{57}$$

where $\overline{v}$ and $\overline{v}'$ represent the initial and final state of C respectively. The difference between the general form of abstract object and abstract value is the presence of v_o in both (56) and (57). In both (56) and (57), v_o represents the *current* value held by the storage for v, which may be renamed to either v' or v, depending on context, in order to abstract the retrieval of the value of the variable v from its storage at the beginning or at the end of executing C. A distinct subscript ('o') is used on v_o to indicate its different meaning from both v, which also occurs in (56), and v', which also occurs in (57).

Similar to abstract value, the abstract object of a construct always occurs as parts of the predicates in the abstractions of its enclosing constructs.

3.6 Remarks

Among the five abstractions introduced in this section the abstractions that are of utmost importance are abstract state, which is useful for building an abstract model, and abstract effect, which is useful for specifying abstract operations. These two abstractions form the elements of a model-based functional specification. In this thesis, we shall derive a set of *abstraction rules* by which the abstract state and effect of any construct, may be obtained recursively from the five abstractions of its primitive constructs.

4 Aggregation of Abstractions

In this thesis we demonstrate that, using the abstractions introduced in the previous section, the abstractions for any construct in the subset of C described in Section 5 may be obtained by aggregating the abstractions of its primitive constructs according to a set of abstraction rules, excepts for the following cases:

- The abstract type and abstract state of an identifier in an expression must be retrieved from the corresponding abstractions of the definition of the identifier.

- When substituting the body of a function in place of its invocation, the function body must be retrieved from the definition of the function.

This section identifies and defines a set of basic aggregation operations in Z that will be used throughout subsequent derivation of the abstraction rules.

4.1 Conjunction of Abstract States

For a construct C that consists of constructs C_1 and C_2, the variables of C must be the variables of either C_1 and C_2, or both. Therefore, we may form the abstract state of C by conjoining the abstract states of C_1 and C_2:

$$\sigma[\![C]\!] \mathrel{\widehat=} \sigma[\![C_1]\!] \wedge \sigma[\![C_2]\!] \tag{58}$$

The conjunction implies that the state invariant of $\sigma[\![C]\!]$ is the conjunction of those of C_1 and C_2. However, since the state invariant for both C_1 and C_2 is the predicate *true*, the state invariant of C is also the predicate *true*, as one would expect.

4.2 Sequential Composition between Operation Schemas

Whenever two constructs are executed sequentially, an abstraction of the combined effect may be sought by sequentially composing their abstract effects. Since we are representing abstract effect of a construct by an operation schema in Z, such composition may be represented using a method very similar to schema composition.

Recall that schema composition requires the primed variables declared in the first schema to match the unprimed variables declared in the second, We may not simply use schema composition because such matching is not satisfied in general. For example, if a sequence of two statements have different sets of variables (which is generally the case), then the operation schemas which represent the abstract effect of these statements will certainly have incompatible signatures.

Therefore, we need to define a slightly different form of schema composition that overrides the requirement of signature compatibility.

Consider these two operation schemas

$$S_1 \mathrel{\widehat=} [\overline{x}; \overline{y}; \overline{x}'; \overline{y}' \mid P_1(\overline{x}, \overline{y}, \overline{x}', \overline{y}')] \tag{59}$$

$$S_2 \mathrel{\widehat=} [\overline{y}; \overline{z}; \overline{y}'; \overline{z}' \mid P_2(\overline{y}, \overline{z}, \overline{y}', \overline{z}')] \tag{60}$$

where $\overline{y}$ and $\overline{y}'$ are the variables declared in both schemas. S_1 and S_2 are not signature compatible due to the absence of $\overline{x}$ in the signature part of S_2. If we intend use schema composition, we must force these schemas to be signature compatible by modifying S_2 such that all primed variables declared in S_1 have their unprimed counterpart declared in S_2. However, if we simply add the necessary declarations to S_2, *i.e.*, rewriting S_2 as

$$S_2'' \mathrel{\widehat=} [\overline{x}; \overline{y}; \overline{z}; \overline{x}'; \overline{y}'; \overline{z}' \mid P_2(\overline{y}, \overline{z}, \overline{y}', \overline{z}')] \tag{61}$$

the result would be unacceptable because, according to the semantics of Z, since P_2 makes no provisions on the postcondition on $\overline{x}$ (note the absence of $\overline{x}'$ in P_2), they may be changed by S_2'' in any way (subject only to type constraints). Rather, we want the rewritten schema to specify that $\overline{x}$ remains constant after the operation. This may be achieved by rewriting S_2 as

$$S_2''' \mathrel{\widehat=} [\overline{x}; \overline{y}; \overline{z}; \overline{x}'; \overline{y}'; \overline{z}' \mid P_2(\overline{y}, \overline{z}, \overline{y}', \overline{z}') \wedge \overline{x} = \overline{x}'] \tag{62}$$

which may be abbreviated using *schema conjunction* in Z as

$$S_2''' \mathrel{\widehat=} S_2 \wedge [\overline{x}; \overline{x}' \mid \overline{x}' = \overline{x}] \tag{63}$$

The declarations for $\overline{x}$ and $\overline{x}'$ in S_2''' come from the signature of S_1. Based on these discussions, we may define *sequential composition* between two schemas by *extending* schema composition with following definition:

$$S_1 \mathbin{;} S_2 \mathrel{\widehat=} S_1 \mathbin{;} (S_2 \wedge [\overline{x}; \overline{x}' \mid \overline{x} = \overline{x}']) \tag{64}$$

where $\overline{x}$ are those variables that are declared only in S_1.

We may obtain a simpler definition for sequential composition between two operation schemas by expanding (64) as follows:

$$\begin{aligned} S_1 \fatsemi S_2 &\widehat{=} [\overline{x}; \overline{y}; \overline{z}; \overline{x}'; \overline{z}' \mid (\exists \overline{x}''; \overline{y}'' \bullet \\ &\qquad P_1(\overline{x}, \overline{y}, \overline{x}'', \overline{y}'') \wedge P_2(\overline{y}'', \overline{z}, \overline{y}', \overline{z}') \wedge \overline{x}' = \overline{x}'')] \\ &\widehat{=} [\overline{x}; \overline{y}; \overline{z}; \overline{x}'; \overline{y}'; \overline{z}' \mid (\exists \overline{y}'' \bullet \\ &\qquad P_1(\overline{x}, \overline{y}, \overline{x}', \overline{y}'') \wedge P_2(\overline{y}'', \overline{z}, \overline{y}', \overline{z}'))] \end{aligned} \tag{65}$$

If all variables declared in S_1 are also declared in S_2, *i.e.*, both $\overline{x}$ and $\overline{z}$ are empty, and in that case sequential composition reduces to ordinary schema composition.

Now, let S_1 be the abstract effect of a construct C_1, S_2 be the abstract effect of another construct C_2. The abstract effect of sequential execution of these constructs is $S_1 \fatsemi S_2$. The term $\overline{y}$ in (65) represents those variables that are common between C_1 and C_2. The case when sequential composition reduces to ordinary schema composition corresponds to a situation where both constructs have the same set of variables.

associativity of sequential composition among operation schemas Sequential composition among operation schemas is associative, that is, for any operation schemas S_1, S_2, S_3,

$$(S_1 \fatsemi S_2) \fatsemi S_3 \equiv S_1 \fatsemi (S_1 \fatsemi S_3) \tag{66}$$

Proof The identity is obvious from expanding both sides by (65).

4.3 Sequential Composition between an Operation Schema and a Definite Description

We shall introduce another kind of sequential composition by an example.

Example 1 Let E be the expression $\mathsf{E}_1\text{-}\mathsf{E}_2$ [8], where E_1 is `i++` and E_2 is `i--`, `i` is a variable of type `int`. Following our principle of deriving abstractions for a construct by aggregating the abstractions of its primitive constructs, one may attempt to find the abstract value of E expressed in terms of its initial state, *i.e.*, $\nu[\![\mathsf{E}]\!]$, by first finding the abstract values of E's *subexpressions* in terms of their respective initial states, *i.e.*, $\nu[\![\mathsf{E}_1]\!]$ and $\nu[\![\mathsf{E}_2]\!]$, and then subtract $\nu[\![\mathsf{E}_2]\!]$ from $\nu[\![\mathsf{E}_1]\!]$.

By observation we may write

$$\nu[\![\mathsf{E}_1]\!] = \nu[\![\texttt{i++}]\!] = (\mu\, \nu_1 : \mathit{Min}..\mathit{Max} \mid \nu_1 = i \wedge i < \mathit{Max}) \tag{67}$$

$$\nu[\![\mathsf{E}_2]\!] = \nu[\![\texttt{i--}]\!] = (\mu\, \nu_1 : \mathit{Min}..\mathit{Max} \mid \nu_1 = i \wedge \mathit{Min} < i) \tag{68}$$

Both descriptions looks very similar when they are observed independently, *i.e.*, they both assert that the abstract value of the expression is i, except that while $\nu[\![\mathsf{E}_1]\!]$ is undefined at $i \geq \mathit{Max}$ (where increment operation is invalid), $\nu[\![\mathsf{E}_2]\!]$ is defined everywhere except at $i = \mathit{Min}$ (where the decrement operation is invalid). Since both descriptions have value i where they are defined, subtracting $\nu[\![\mathsf{E}_2]\!]$ from $\nu[\![\mathsf{E}_2]\!]$ would result in zero wherever both descriptions are defined. Therefore, we have

$$\nu[\![\mathsf{E}_1]\!] - \nu[\![\mathsf{E}_2]\!] = (\mu\, \nu_1 : \mathit{Min}..\mathit{Max} \mid \nu_1 = i \wedge \mathit{Min} < i < \mathit{Max}) \tag{69}$$

whose value is zero for $\mathit{Min} < i < \mathit{Max}$ and is undefined for both $i = \mathit{Min}$ and $i = \mathit{Max}$. One may notice that, in this particular case, the value of the definite description does not seem to be related to i. However, the presence of the inequality $\mathit{Min} < i < \mathit{Max}$ is important here because it specifies the condition under which the definite description is applicable. Before going ahead to conclude that $\nu[\![\mathsf{E}]\!]$ is

[8] Although the results of evaluating `(i++)-(i--)` is undefined according to the semantics of C, for its result value depends on the order of evaluation, the example serves to illustrate the same issue for expressions whose order of evaluation among their operands is fixed (e.g., conditional and logical expressions).

the definite description in (69), we shall validate our conclusion in (69) against the results that we may obtain from manually executing the expression. Assuming a left to right order of evaluation, the following table shows the steps in executing E, along with initial and final values of i in each step, and the result values of E and its subexpressions.

step	instruction	initial value of i	final value of i	result value of E and its expressions	
1	evaluate E_1 (i++)	i_i	$i_i + 1$	E_1	i_i
2	evaluate E_2 (i--)	$i_i + 1$	i_i	E_2	$i_i + 1$
3	subtract 2^{nd} value from 1^{st}	i_i	i_i	E	-1

Table 1: Manual Execution of (i++)-(i--)

where i_i denotes the initial value of i in evaluating E.

We may observe from the table that the result value of E is -1, and is valid for all $i_i < Max$, because the value of i alternates between i_i and $i_i + 1$ throughout evaluation of E. Therefore we may conclude that the abstract value of E expressed in terms of its initial state,

$$\nu[\![E]\!] = (\mu\, \nu_1 : Min..Max \mid \nu_1 = -1 \wedge i < Max) \tag{70}$$

The conclusion in (69) is wrong for two reasons: $\nu[\![E]\!]$ should be -1, rather than zero wherever it is defined, and should be undefined only for $i = Max$, rather than for both $i = Min$ and $i = Max$. Such apparent inconsistency in this case is due to the fact that by subtracting $\nu[\![E_2]\!]$ from $\nu[\![E_1]\!]$, one implicitly assumes that the initial states of both E_1 and E_2 are identical, which in this case is not true.

A remedy for this situation is described as follows: When we write the abstract value of E in terms of its initial state, by definition, we are writing it in terms of i_i, because i is the only variable of E. Since E_1 is evaluated first in a left to right order of evaluation, i_i serves as the initial value of i for E_1 as well. However, for E_2, we

also need to consider the effect of evaluating E_1 in order to enforce sequentiality between E_1 and E_2. In particular, the initial value of i in $\nu[\![\mathsf{E}_2]\!]$ must be i_i+1 (*i.e.*, initial value of i in step two in Table 1), rather than i_i. Substitute i_i+1 into (68) we obtain

$$\nu[\![\mathsf{E}_2]\!] = (\mu\,\nu_1 : Min..Max \mid \nu = (i_i+1) \wedge Min < (i_i+1)) \tag{71}$$

We may rewrite $\nu[\![\mathsf{E}_2]\!]$ such that i denotes i_i by substituting i in place of i_i in (71). This produces

$$\begin{aligned} \textbf{corrected}\,\nu[\![\mathsf{E}_2]\!] &= (\mu\,\nu_1 : Min..Max \mid \nu_1 = (i+1) \wedge Min < (i+1)) \\ &= (\mu\,\nu_1 : Min..Max \mid \nu_1 = i+1 \wedge i < Max) \end{aligned} \tag{72}$$

whose value is $i-1$ except for $i = Max$ where the definite description is undefined.

Now we may subtract **corrected** $\nu[\![\mathsf{E}_2]\!]$ from $\nu[\![\mathsf{E}_1]\!]$ to obtain the proper abstract value of E. Since both definite descriptions in (72) and (67) are defined for $i < Max$, the subtraction would give $i-(i+1) = -1$, for $i < Max$. Therefore,

$$\begin{aligned} \nu[\![\mathsf{E}]\!] &= \nu[\![\mathsf{E}_1]\!] - \textbf{corrected}\,\nu[\![\mathsf{E}_2]\!] \\ &= (\mu\,\nu_1 : Min..Max \mid \nu_1 = i \wedge i < Max) \\ &\quad - (\mu\,\nu_1 : Min..Max \mid \nu_1 = i+1 \wedge i < Max) \\ &= (\mu\,\nu_1 : Min..Max \mid \nu_1 = -1 \wedge i < Max) \end{aligned} \tag{73}$$

which agrees with (70) exactly.

In Example 1, the need for correcting $\nu[\![\mathsf{E}_2]\!]$ comes from the fact that, before E_2 is evaluated, i has been incremented in evaluating E_1. Therefore, the correction we have just performed may be thought as somehow *appending* the result value of E_2 to the effect of E_1. This may be formalized by defining composition between an

operation schema and a definite description. Consider an operation schema S and a definite description D:

$$S \widehat{=} [\overline{x}; \overline{y}; \overline{x}'; \overline{y}' \mid P_S(\overline{x}, \overline{y}, \overline{x}', \overline{y}')] \tag{74}$$

$$D = (\mu \nu_1 : T \mid P_D(\nu_1, \overline{y}, \overline{z})) \tag{75}$$

where $\overline{y}$ are the variables common to both S and D, P_D is the constraining predicate of D and P_S is the predicate part of S.

If $\overline{y}$ is not empty, we may define *sequential composition between an operation schema and a definite description* as the following:

$$S \fatsemi D = (\mu \nu_1 : T \mid (\exists \overline{x}'; \overline{y}' \bullet P_S(\overline{x}, \overline{y}, \overline{x}', \overline{y}') \land P_D(\nu_1, \boxed{\overline{y}'}, \overline{z}))) \tag{76}$$

Equation (76), in effect, substitutes the final values of the variables that are common between S and D, in performing S, as the values of these variables for D (The substituted quantity is denoted by $\boxed{\overline{y}'}$ in (76)). The result of the composition is still a definite description of the same type (as the declaration for ν_1 is the same between (75) and (76) Note here that we have overloaded the operator $\fatsemi$ that is used for sequential composition between two operation schemas.

If $\overline{y}$ is empty, (76) reduces to

$$\begin{aligned} S \fatsemi D &= (\mu \nu_1 : T \mid (\exists \overline{x}' \bullet P_S(\overline{x}, \overline{x}') \land P_D(\nu_1, \overline{z}))) \\ &= (\mu \nu_1 : T \mid (\exists \overline{x}' \bullet P_S(\overline{x}, \overline{x}')) \land P_D(\nu_1, \overline{z})) \\ &= (\mu \nu_1 : T \mid \mathbf{pre}\, S \land P_D(\nu_1, \overline{z})) \end{aligned} \tag{77}$$

which is the same as D in (75) except that it is undefined where $\mathbf{pre}\, S$ is not satisfied.

Let us return to Example 1. We may calculate the abstract value of E, expressed in terms of its initial state, using the abstract values of E_1 and E_2, both expressed

also in terms of the same initial state, using the sequential composition we have just defined. E_1 increments $\mathtt{i}$, provided that $\mathtt{i}$ is not already equal to Max. Therefore its abstract effect is

$$\epsilon[\![\mathsf{E}_1]\!] \mathrel{\widehat{=}} [i, i' \mid i < Max \land i' = i + 1] \tag{78}$$

Now, perform sequential composition between $\epsilon[\![\mathsf{E}_1]\!]$ and $\nu[\![\mathsf{E}_2]\!]$ using equation (76):

$$\begin{aligned}
\epsilon[\![\mathsf{E}_1]\!] \mathbin{;} \nu[\![\mathsf{E}_2]\!] &= [i, i' \mid i < Max \land i' = i + 1] \mathbin{;} (\mu\, \nu_1 : Min..Max \mid \nu_1 = i \land Min < i) \\
&= (\mu\, \nu_1 : Min..Max \mid (\exists\, i' : Min..Max \mid i < Max \land i' = i + 1 \\
&\qquad\qquad \land\ \nu_1 = i' \land Min < i')) \\
&= (\mu\, \nu_1 : Min..Max \mid i < Max \land \nu_1 = i + 1 \land Min < i + 1) \\
&= (\mu\, \nu_1 : Min..Max \mid i < Max \land \nu_1 = i + 1)
\end{aligned} \tag{79}$$

Note that (79) is exactly the same as (72), which means that, $\epsilon[\![\mathsf{E}_1]\!] \mathbin{;} \nu[\![\mathsf{E}_2]\!]$ in fact calculates **corrected** $\nu[\![\mathsf{E}_2]\!]$. Finally, we may properly conclude that

$$\nu[\![\mathsf{E}]\!] = \nu[\![\mathsf{E}_1]\!] - (\epsilon[\![\mathsf{E}_1]\!] \mathbin{;} \nu[\![\mathsf{E}_2]\!]) \tag{80}$$

associativity of sequential composition of a sequence of operation schemas and a definite description Sequential composition between a sequence of operation schemas and a definite description is associative in the sense that, for any operation schemas S_1, S_2, and definite description D,

$$(S_1 \mathbin{;} S_2) \mathbin{;} D \equiv S_1 \mathbin{;} (S_2 \mathbin{;} D) \tag{81}$$

Proof The identity is obvious by expanding both sides with (76) and (65).

Conjecture 1 (*distributivity over sequential execution*) In general, if an expression E contains subexpressions E_1, E_2, ..., E_n such that the effect of evaluating E is that of sequential evaluation of the subexpressions E_1, E_2, ..., E_n, in that order, then the abstract value of E in terms of its initial state may be obtained from those of its subexpressions, if and only if, the abstract value of each subexpression E_j, where $2 \leq j \leq n$, expressed in terms of its initial state, is appended to the abstract effects of E_1, E_2, ..., and E_{j-1}. This may be formally stated as

$$\mathbf{corrected}\, \nu[\![\mathsf{E}_j]\!] \cong (\epsilon[\![\mathsf{E}_1]\!] \fatsemi \epsilon[\![\mathsf{E}_2]\!] \fatsemi \cdots \fatsemi \epsilon[\![\mathsf{E}_{j-1}]\!] \fatsemi \nu[\![\mathsf{E}_j]\!]) \tag{82}$$

4.4 Sequential Composition between a Definite Description and an Operation Schema

Let us now consider appending an abstract effect to an abstract value.

Example 2 Consider the previous example (Example 1). Let E be E_1-E_2, where E_1 is `i++` and E_2 is `i--`. Again, we want to find the abstract value of E, but this time we want to express it in terms of the final state of E. We first express the abstract values of E_1 and E_2 in terms of their respective final states. By observation we have

$$\nu'[\![\mathsf{E}_1]\!] = \nu'[\![\texttt{i++}]\!] = (\mu\, \nu_1 : \mathit{Min}..\mathit{Max} \mid \nu_1 = i' - 1 \wedge \mathit{Min} < i') \tag{83}$$

$$\nu'[\![\mathsf{E}_2]\!] = \nu'[\![\texttt{i--}]\!] = (\mu\, \nu_1 : \mathit{Min}..\mathit{Max} \mid \nu_1 = i' + 1 \wedge i' < \mathit{Max}) \tag{84}$$

Note that $\nu'[\![\mathsf{E}_1]\!]$ is defined for $\mathit{Min} < i'$, while $\nu'[\![\mathsf{E}_2]\!]$ is defined for $i' < \mathit{Max}$. Then we subtract $\nu'[\![\mathsf{E}_2]\!]$ from $\nu'[\![\mathsf{E}_1]\!]$ to obtain $\nu'[\![\mathsf{E}]\!]$:

$$\nu'[\![\mathsf{E}]\!] = \nu'[\![\mathsf{E}_1]\!] - \nu'[\![\mathsf{E}_2]\!]$$

$$
\begin{aligned}
&= (\mu\, \nu_1 : Min..Max \mid \nu_1 = i' - 1 \wedge Min < i') \\
&- (\mu\, \nu_1 : Min..Max \mid \nu_1 = i' + 1 \wedge i' < Max) \\
&= (\mu\, \nu_1 : Min..Max \mid \nu_1 = (i' - 1) - (i' + 1) \wedge Min < i' < Max) \\
&= (\mu\, \nu_1 : Min..Max \mid \nu_1 = -2 \wedge Min < i' < Max) \qquad (85)
\end{aligned}
$$

which asserts that the abstract value of E is -2 and must result in a final value of $\mathtt{i}$ satisfying $Min < \mathtt{i} < Max$.

In this particular example, we know that when order of evaluation to left to right, the result value of E is -1 (from Table 1), which is a constant. Therefore, the abstract value of E must be -1 no matter whether we express it in terms of its initial or final state. Additionally, we may observe from Table 1 that the initial and final values of $\mathtt{i}$ in evaluating E happen to be both i_i. From this observation the constraint on the final value of $\mathtt{i}$ for the evaluation to be valid must be exactly the same as before, *i.e.*, the final value of $\mathtt{i}$ must be less than Max. Hence, we may conclude that the abstract value of E expressed in terms of its final state must have exactly the same form as that expressed in terms of its initial state (70), that is:

$$\nu'[\![\mathsf{E}]\!] = (\mu\, \nu_1 : Min..Max \mid \nu_1 = -1 \wedge i' < Max) \qquad (86)$$

The conclusion we have in (85) is wrong for two reasons: Firstly, $\nu'[\![\mathsf{E}]\!]$ should be -1 rather than -2. Secondly, the constraint on the final value of $\mathtt{i}$ should be $\mathtt{i} < Max$ rather than $Min < \mathtt{i} < Max$. Similar to Example 1, such apparent inconsistency is due to the implicit assumption that the final states of E_1 and E_2 are identical, which is not true.

A remedy for this situation is described as follows: Since E_2 is evaluated last in a left to right order of evaluation, the final value of $\mathtt{i}$ as seen by E_2 must be the same as that for E. However, for E_1, we also need to consider the effect of

subsequent evaluation of E_2 in order to enforce sequentiality between E_1 and E_2. In particular, let i_f be the final value of i as seen by E; then the final value of i as seen by E_1 must be $i_f + 1$ to offset the effect of subsequent decrementation of i performed in evaluating E_2. Therefore, i' in (83) may be thought as denoting $i_f + 1$, rather than i_f. In other words,

$$\begin{aligned} \nu'[\![\mathsf{E}_1]\!] &= (\mu\, \nu_1 : Min..Max \mid \nu_1 = (i_f + 1) - 1 \wedge Min < (i_f + 1)) \\ &= (\mu\, \nu_1 : Min..Max \mid \nu_1 = i_f \wedge i_f < Max) \end{aligned} \tag{87}$$

We may rewrite $\nu'[\![\mathsf{E}_1]\!]$ such that i denotes i_f by substituting i in place of i_f in (87). This results in

$$\mathbf{corrected}\, \nu'[\![\mathsf{E}_1]\!] = (\mu\, \nu_1 : Min..Max \mid \nu_1 = i' \wedge i' < Max) \tag{88}$$

The value of $\mathbf{corrected}\, \nu'[\![\mathsf{E}_1]\!]$ is i' except for $i' = Max$ where the definite description is undefined.

Now we may subtract $\nu'[\![\mathsf{E}_2]\!]$ from $\mathbf{corrected}\, \nu'[\![\mathsf{E}_2]\!]$ to obtain the proper abstract value of E. Since the definite descriptions in both (84) and (88) are defined for $i' < Max$, the subtraction would give $i' - (i' + 1) = -1$, defined for $i' < Max$. Therefore,

$$\begin{aligned} \nu'[\![\mathsf{E}]\!] &= \mathbf{corrected}\, \nu'[\![\mathsf{E}_1]\!] - \nu'[\![\mathsf{E}_2]\!] \\ &= (\mu\, \nu_1 : Min..Max \mid \nu_1 = i' \wedge i' < Max) \\ &\quad - (\mu\, \nu_1 : Min..Max \mid \nu_1 = i' + 1 \wedge i' < Max) \\ &= (\mu\, \nu_1 : Min..Max \mid \nu_1 = -1 \wedge i' < Max) \end{aligned} \tag{89}$$

which agrees with (86) exactly.

The correction process done in Example 2 may be thought as appending the effect of E_2 to the result value of E_1. This may be formalized by defining sequential composition between a definite description and an operation schema.

Consider the following operation schema S and definite description D:

$$S \triangleq [\overline{x}; \overline{y}; \overline{x}'; \overline{y}' \mid P_S(\overline{x}, \overline{y}, \overline{x}', \overline{y}')] \tag{90}$$

$$D = (\mu \nu_1 : T \mid P_D(\nu_1, \overline{y}', \overline{z}')) \tag{91}$$

where $\overline{y}'$ are the only variables that are common between S and D. If $\overline{y}'$ is not empty, we may define the sequential composition between a definite description and an operation schema as follows.

$$D \fatsemi S = (\mu \nu_1 : T \mid (\exists \overline{x}, \overline{y} \bullet P_D(\nu_1, \boxed{\overline{y}}, \overline{z}') \wedge P_S(\overline{x}, \overline{y}, \overline{x}', \overline{y}'))) \tag{92}$$

Equation (92), in effect, substitutes the initial values of the variables that are common between S and D in performing S as the values of these variables for D (The substituted quantity is denoted by $\boxed{\overline{y}}$ in (92)). The result of the composition is still a definite description of the same type, as indicated by identical declarations for ν_1 between (92) and (91). If $\overline{y}'$ is empty, (92) reduces to

$$D \fatsemi S = (\mu \nu_1 : T \mid P_D(\nu_1, \overline{z}') \wedge \mathbf{post}\, S) \tag{93}$$

which is the same as D (91) except that it is undefined where $\mathbf{post}\, S$ is violated.

For Example 2, the abstract value of E expressed terms of its final state may be rewritten as

$$\nu'[\![\mathsf{E}]\!] = \nu'[\![\mathsf{E}_1]\!] \fatsemi \epsilon[\![\mathsf{E}_2]\!] - \nu'[\![\mathsf{E}_3]\!] \tag{94}$$

associativity of sequential composition of a definite description and a sequence of operation schemas Sequential composition between a definite description and a sequence of operation schemas is associative in the sense that, for any operation schemas S_1, S_2, and definite description D,

$$(D \fatsemi S_1) \fatsemi S_2 \equiv D \fatsemi (S_1 \fatsemi S_2) \tag{95}$$

The identity is obvious when both sides are expanded with (65) and (92).

Conjecture 2 (*distributivity over sequential evaluation*) In general, if an expression E contains subexpressions E_1, E_2, ..., E_n such that effect of evaluating E is that of sequential evaluation of the subexpressions E_1, E_2, ..., E_n, in that order, then the abstract value of E in terms of its final state may be obtained from those of its subexpressions, if and only if, for each subexpression E_j, where $1 \leq j \leq n-1$, the abstract effects of the subexpressions E_{j+1}, E_2, ..., and E_n are appended to its abstract value, expressed in terms of its final state. This may be formally stated as

$$\mathbf{corrected}\,\nu'[\![\mathsf{E}_j]\!] = (\nu'[\![\mathsf{E}_j]\!] \,;\, \epsilon[\![\mathsf{E}_{j+1}]\!] \,;\, \epsilon[\![\mathsf{E}_{j+2}]\!] \,;\, \cdots \,;\, \epsilon[\![\mathsf{E}_n]\!]) \tag{96}$$

4.5 Sequential Composition between an Abstract Object and an Operation Schema

Let us now consider appending an operation schema to an abstract object. The following example illustrates such necessity.

Example 3 Consider the following assignment expression:

```
a[i]=i++
```
(97)

Let i_f be the final value of `i` after the assignment expression is evaluated. Assuming a left to right order of evaluation, the object that is being assigned in the assignment expression, expressed in terms of the final value of `i`, is `a[`$i_f - 1$`]`. The '-1' here is needed to offset the effect of the increment operation on the right side of the assignment. The augmentation performed in the above example may be formalized by defining sequential composition between an abstract object and an operation schema.

Since abstract object is represented by a definite description, sequential composition between the abstract object of an object designation expression and another operation schema, may be evaluated using (92), the same formula for composition between a definite description and an operation schema. However, for the limited varieties of object designation expression considered in this thesis, namely, variables, array subscripting expressions, and structure component selection expressions, we may also develop formulas for each of these cases.

Let us consider sequential composition between the abstract object of an object designation expression E and an operation schema S.

variable If E is a variable v, from (rule 24), we have $\omega'[\![\mathsf{v}]\!] = \overline{v}$. Since the address of a variable may not change throughout its lifetime,

$$\omega'[\![\mathsf{v}]\!] \; ; \; S = \omega'[\![\mathsf{v}]\!] \tag{98}$$

array subscripting expression If E is an array subscripting expression of form $\mathsf{E}_1\,[\mathsf{E}_2]$, then from (rule 31), $\omega'[\![\mathsf{E}_1\,[\mathsf{E}_2]]\!]$ is $(\omega'[\![\mathsf{E}_1]\!] \; ; \; \epsilon[\![\mathsf{E}_2]\!])[\nu'[\![\mathsf{E}_2]\!]]$. To evaluate the the sequential composition $\omega'[\![\mathsf{E}]\!] \; ; \; S$, we may distribute the composition (proof below) between $(\omega'[\![\mathsf{E}_1]\!] \; ; \; \epsilon[\![\mathsf{E}_2]\!])$ and $\nu'[\![\mathsf{E}_2]\!]$ to obtain the following definition:

$$\omega'[\![\mathsf{E}_1\,[\mathsf{E}_2]]\!] \; ; \; S = (\omega'[\![\mathsf{E}_1]\!] \; ; \; \epsilon[\![\mathsf{E}_2]\!] \; ; \; S)[\nu'[\![\mathsf{E}_2]\!] \; ; \; S] \tag{99}$$

Formula (99) may be justified by the following identity: For any definite descriptions D_1 and D_2, and operation schema S_3,

$$(D_1[D_2]\,) \; ; \; S_3 \equiv (D_1 \; ; \; S_3)[D_2 \; ; \; S_3] \tag{100}$$

Proof Expansion of both sides of the identity would show that a sufficient condition for the identity to hold is that the operation schema S_3 specifies a deterministic operation.

structure component selection expression Let E be a structure component selection expression of form $\mathsf{E}_1.\mathsf{f}_i$, where f_i is a component of the type of the object designated by E_1. From (rule 38), $\omega'[\![\mathsf{E}_1.\mathsf{f}_i]\!]$ is $\omega'[\![\mathsf{E}_1]\!].f_i$. Since the composition between $\omega'[\![\mathsf{E}_1]\!].f_i$ and S may not change a schema component name (f_i in this case),

$$\omega'[\![\mathsf{E}_1.\mathsf{f}_i]\!] \; ; \epsilon[\![\mathsf{C}]\!] = (\omega'[\![\mathsf{E}_1]\!] \; ; \epsilon[\![\mathsf{C}]\!]).f_i \tag{101}$$

associativity of sequential composition of a definite description and a sequence of operation schemas The compositions between an abstract object and a sequence of abstract effects are associative, just as an abstract value and a sequence of abstract effects do.

4.6 Sequential Composition between an Operation Schema and an Abstract Object

Appending an abstract object to an operation schema may be defined similarly as appending an operation schema to an abstract object.

Consider the sequential composition between the operation schema S and the abstract object of an object designation expression E. We may derive formulas for each kind of object designation expressions.

variable If E is a variable v, then from (rule 24), $\omega[\![\mathsf{v}]\!] = \overline{v}$. Since the address of a variable may not change,

$$S \,;\, \omega[\![\mathsf{v}]\!] = \omega[\![\mathsf{v}]\!] \tag{102}$$

array subscripting expression If E is an array subscripting expression of form $\mathsf{E}_1\,[\mathsf{E}_2]$, then from (rule 30), $\omega[\![\mathsf{E}_1\,[\mathsf{E}_2]]\!]$ is $\omega[\![\mathsf{E}_1]\!][\epsilon[\![\mathsf{E}_1]\!] \,;\, \nu[\![\mathsf{E}_2]\!]]$. If S specifies a deterministic operation, we may distributive the sequential composition in $S \,;\, \omega[\![\mathsf{E}]\!]$ between $\omega'[\![\mathsf{E}_1]\!]$ and $\epsilon[\![\mathsf{E}_1]\!] \,;\, \nu[\![\mathsf{E}_2]\!]$ to obtain the following definition:

$$S \,;\, \omega[\![\mathsf{E}_1\,[\mathsf{E}_2]]\!] = (S \,;\, \omega[\![\mathsf{E}_1]\!])[S \,;\, \epsilon[\![\mathsf{E}_1]\!] \,;\, \nu[\![\mathsf{E}_2]\!]] \tag{103}$$

structure component selection expression If E is a structure component selection expression of form $\mathsf{E}_1\,.\,\mathsf{f}_i$, where f_i is a component of the type of the object designated by E_1, then from (rule 37), $\omega[\![\mathsf{E}_1\,.\,\mathsf{f}_i]\!]$ is $\omega[\![\mathsf{E}_1]\!].f_i$. The composition $\omega[\![\mathsf{E}_1\,.\,\mathsf{f}_i]\!] \,;\, \epsilon[\![\mathsf{C}]\!]$ is then

$$\epsilon[\![\mathsf{C}]\!] \,;\, \omega[\![\mathsf{E}_1\,.\,\mathsf{f}_i]\!] = (\epsilon[\![\mathsf{C}]\!] \,;\, \omega[\![\mathsf{E}_1]\!]).f_i \tag{104}$$

associativity of sequential composition of a definite description and a sequence of operation schemas The compositions between an abstract object and a sequence of abstract effects are associative, just as an abstract value and a sequence of abstract effects do.

5 Abstractions for a C subset

For the purposes of this thesis, we consider only programs written using a subset of the C language that includes the following features:

- Every program must contain a parameterless non-recursive function, call '`main`',
- The only data types supported are `int`, arrays (any dimension) of `int` and structures, and structures whose components have type as one of the above three types or structures by themselves.
- All functions are parameterless functions and do not return any value (*i.e.*, the return type is `void`).
- A function invocation may occur as an expression statement only.
- Pointers are not included.
- Every program is assumed to be preprocessed, that is, it does not contain any preprocessor directives[14, p.39].

The main simplifying assumption that gives rise to the chosen subset is that the parent of the object designated by an object designation expression is fixed .

5.1 Extended BNF Grammar for the C Subset

The grammar presented below is ambiguous in the sense that operator precedence and associativity rules are not implied by the grammar. Therefore, a tool must use this grammar in conjunction with Table 2, which presents operator precedence and associativity rules. The grammar has been written so that the classification of certain syntactic elements as constructs, as well as the enclosure relationship

among these construct, is evident. Specifically, each non-terminal on the left hand side of each production represents a category of constructs. The construct 'type expression', which may be enclosed by a variable definition, is the exception (note its absence in the grammar). This is because the type expression in the definition a array variable is lexically split into two parts (element type precedes the variable name, while the size specification proceeds the variable name). In the grammar, 'aop' stands for an arithmetic operator, and 'rop' stands for a relational operator. There is one grammatical rule that is invisible from the grammar: A 'variable definition' enclosed within another 'variable definition' may neither have the keyword '`static`' at its beginning nor an '=' 'initializer' at its end.

Here is the extended BNF grammar for our C subset:

program ← { (variable-definition | function-definition) }

variable-definition ← ['`static`'] ('`int`' | '`struct`' '{' { variable-definition } '}')
identifier [{ '[' integer-constant ']' }]
['=' 'initializer'] ';'

function-definition ← '`void`' 'function-name' '()' block

block ← '{' [{ variable-definition }] [sequence-of-statements] '}'

sequence-of-statements ← { statement }

statement ← expression-statement
| block
| if-statement
| if-then-else-statement
| while-statement

```
                | do-while-statement
                | for-statement

expression-statement ← expression ';'
                     | 'function-name' '();'

expression ← integer-constant
           | parenthesized-expression
           | object-designation-expression
           | unary-arithmetic-expression
           | binary-arithmetic-expression
           | relational-expression
           | conditional-expression
           | logical-expression
           | simple-assignment-expression
           | compound-assignment-expression
           | post-increment-expression
           | post-decrement-expression
           | pre-increment-expression
           | pre-decrement-expression

parenthesized-expression ← '(' expression ')'

object-designation-expression ← 'variable-name'
                              | array-subscripting-expression
                              | structure-component-selection-expression

integer-constant ← 'integer-constant'
```

identifier ← **'variable-name'**
| **'structure-component-name'**

structure-component-selection-expression ←
object-designation-expression '.' **'structure-component-name'**

array-subscription-expression ← object-designation-expression '[' expression ']'

parenthesized-expression ← '(' expression ')'

unary-arithmetic-expression ← '+' expression
| '-' expression

binary-arithmetic-expression ← expression **'aop'** expression

relational-expression ← expression **'rop'** expression

conditional-expression ← expression '?' expression ':' expression

logical-expression ← expression '&&' expression
| expression '||' expression
| '!' expression

simple-assignment-expression ← expression '=' expression

compound-assignment-expression ← expression **'aop='** expression

post-increment-expression ← expression '++'

post-decrement-expression ← expression '--'

pre-increment-expression ← '++' expression

pre-decrement-expression ← '--' expression

if-statement ← 'if' '(' expression ')' statement

if-then-else-statement ← 'if' '(' expression ')' statement 'else' statement

while-statement ← 'while' '(' expression ')' statement

do-while-statement ← 'do' statement 'while' '(' expression ')' ';'

for-statement ← 'for' '(' expression ';' expression ';' expression ')' statement

5.2 Precedence and Associativity of Operators

The following table shows the associativity of the operators available in the C subset described in the above grammar in descending order of their precedence:

Operator	Description	Associativity
[] . ++ --	array subscripting, structure component selection, postincrement and postdecrement	left
++ -- ! + -	preincrement and predecrement, logical negation, unary arithmetic operators	right
* / %	multiplication, division (aop)	left
+ -	addition, subtraction (aop)	left
> < <= >=	relational (rop)	left
== !=	equal, not equal (rop)	left
&&	conjunction	left
||	disjunction	left
? :	conditional expression	right
= += -= *= /= %=	assignment expression	right

Table 2: Precedence and Associativity Rules for Operators in a C subset

We now consider the abstractions for each category of constructs available in the C subset that we have just presented.

5.3 Variable Definitions

In our C subset, we consider a *variable* as a named storage of a certain type. The *storage class*[14, p.75] of a variable may be either static or local. *Static variables*[14, p.75] are those variables for which storage is allocated before the program starts, and persists throughout the execution of the program. Static variables may be defined outside any function (in which case they are *global variables*) or at the beginning of a block. *Local variables*[14, p.75] may be defined only at the beginning of a block. The storage for local variables are allocated every time the block in which they are defined is entered, and is deallocated when exiting the block.

With some exceptions[14, p.93], each variable definition may be accompanied by an optional *initializer*, which assigns a value to the variable when the storage for the variable is allocated[14, p.92]. Hence, for static variables, initializations may be assumed to be done only once, before the `main` function is invoked. For local variables, initializations are assumed to be done every time the block in which these variables are defined is entered. Our C subset assumes that initializers must be *literal constants*.

Default initializations are permitted, which is assumed to obey the following rule [9]: Every local variable is initialized to an arbitrary value of its type (for example, *Min* and *Max* are both possible initial value of an integer local variables defined without an initializer). For a static variable, the default initialization when the variable has integer type is to assign it with zero. For an array, it would be to

[9]This follows the same rule as stated in [p.93]C, except that a local variable without initializer is assumed to holds a valid value at the beginning, rather than a possibly invalid value

apply default initialization recursively to all its elements. For a structure, it would be to apply default initialization recursively to all its components.

For each variable definition, we may define its abstract state, which abstracts the storage for the variable, and the abstract effect, which abstracts the initialization of the variable. Abstract value and abstract object are left undefined because a variable definition may not appear in an expression.

We shall first define the abstractions for a general variable definition, identify the parameters, and then derive the values for these parameters for the various types available in C.

5.3.1 General Form

We shall use the notation $\mathcal{D}(\mathsf{v})$ to denote the definition of an identifier (either a variable name, or a structure component name) v in a program.

abstract type Suppose v is defined by $\mathcal{D}(\mathsf{v})$ to have type T. By definition, its abstract type would be[10]

$$\tau[\![\mathcal{D}(\mathsf{v})]\!] == \tau[\![\mathsf{T}]\!] \qquad \text{(rule 1)}$$

abstract state $\mathcal{D}(\mathsf{v})$ specifies initialization of v when its storage is allocate. Therefore it operates on the variable v. The abstract state of $\mathcal{D}(\mathsf{v})$ is derived as

$$\sigma[\![\mathcal{D}(\mathsf{v})]\!] \mathrel{\widehat=} [v : \tau[\![\mathsf{T}]\!]] \qquad \text{(rule 2)}$$

where v represents the variable name in the Z specification that correspond to v. is the specification.

[10]each equation with equation number of form (rule *number*) shall indicate that it is an abstraction rule

abstract effect The effect of 'executing' a variable definition is taken as that of the initialization of the variable. When an optional initializer I is present, the effect would be the same as assigning the initializer to the variable. In which case, the abstract effect of $\mathcal{D}(\mathsf{v})$ is

$$\epsilon[\![\mathcal{D}(\mathsf{v})]\!] \mathrel{\widehat=} [\sigma[\![\mathcal{D}(\mathsf{v})]\!]' \mid v' = I] \qquad \text{(rule 3)}$$

where I is the specification of initializer I. In (rule 3), $\epsilon[\![\mathcal{D}(\mathsf{v})]\!]$ is an initial state schema. This is because we do not concern ourselves with the value of v *before* it is initialized. When I is not present, we follow the default initialization rules described earlier. The particular rule to be applied depends on the storage class of the variable. If v is a local variable, the default initialization asserts that v has an arbitrary value of type T, the abstract effect of $\mathcal{D}(\mathsf{v})$ would be,

$$\epsilon[\![\mathsf{D}]\!] \mathrel{\widehat=} [\sigma[\![\mathcal{D}(\mathsf{v})]\!]'] \qquad \text{(rule 4)}$$

If v is a static variable, then In this case the abstract effect of $\mathcal{D}(\mathsf{v})$ would be

$$\epsilon[\![\mathsf{D}]\!] \mathrel{\widehat=} [\sigma[\![\mathcal{D}(\mathsf{v})]\!]' \mid v' = \mathcal{Z}[\![\mathsf{T}]\!]] \qquad \text{(rule 5)}$$

where $\mathcal{Z}[\![\mathsf{T}]\!]$ represents a value of type T that a static variable of type T would hold when default initialization is applied.

We may summarize the abstract effect for various combinations of storage classes and the presence or absence of initializer in the following table:

	storage class	
	static	local
initializer is present	$[\sigma[\![\mathcal{D}(\mathsf{v})]\!]' \mid v' = I]$	$[\sigma[\![\mathcal{D}(\mathsf{v})]\!]' \mid v' = I]$
initializer is absent	$[\sigma[\![\mathcal{D}(\mathsf{v})]\!]' \mid v' = \mathcal{Z}[\![\mathsf{T}]\!]]$	$[\sigma[\![\mathcal{D}(\mathsf{v})]\!]']$

Table 3: Abstract Effect of a Variable Definition

identify the parameters For a variable definition, the general form of its abstractions may be characterized by three parameters:

1. $\tau[\![\mathsf{T}]\!]$, a representation of the type T in Z
2. I, representation of an initializer of type T
3. $\mathcal{Z}[\![\mathsf{T}]\!]$, representing the value of a static variable of type T when default initialization is applied to it.

Since both the second and third parameters in turn depend on T, the type of the variable in the program, we may instantiate these parameters for each types available in our C subset.

5.3.2 Integer

In this case, the parameter T is `int`.

abstract type We may represent T in Z by a subrange of integers. Specifically, we shall define the abstract type of `int` to be

$$\tau[\![\texttt{int}]\!] == Min..Max \qquad \text{(rule 6)}$$

where Min and Max are the minimum and maximum integers representable by the type `int`. The quantities Max and Min may be declared in Z by the following axiomatic definition:

$$\begin{array}{|l} Min : \mathbb{Z} \\ Max : \mathbb{Z} \\ \hline Min < Max \land Min < 0 \land Max > 0 \end{array}$$

initialization An initializer for a variable of type `int` is an integer constant, which may be represented in Z as it is. $\mathcal{Z}[\mathsf{T}]$ in this case is the integer zero. The axiomatic definition for *Min* and *Max* that we have just presented ensures that zero is a valid value of type `int`.

5.3.3 Arrays

In this case, the parameter T is

$$\mathsf{T}_1 \ [\mathsf{B}] \tag{105}$$

where B is an integer constant that specifies the size of the array, and *element type* T_1 is type of the elements in the array.

sequence representations and operations An array may be represented in Z by a sequence. Recall that A sequence of type X in Z may be represented a function from natural number to X:

$$\operatorname{seq} X == \{f : \mathsf{N} \pfun X \mid \operatorname{dom} f = 1..\#f\} \tag{106}$$

Definition (106) implies that the indices of any sequence start from one. In contrast, all arrays in C have indices start from zero. To account for this difference, we need to declare a couple of operators: one for indexing an element of a sequence; the other is a special case of the overriding operator ($\oplus$) that is useful for fixed length sequences. Here are the declarations for these operators:

$$\begin{array}{|l} [X] \\ \hline \_[\_] : \operatorname{seq} X \times \mathsf{N} \pfun X \\ \hline \forall s : \operatorname{seq} X; i : \mathsf{N} \bullet s[i] = s(i+1) \\ \hline \end{array}$$

$$
\begin{array}{|l}
\hline
[X] \\
\_ \boxplus \_ : (\mathbb{N} \pfun X) \times (\mathbb{N} \pfun X) \fun (\mathbb{N} \pfun X) \\
\hline
\forall f, g, h : \mathbb{N} \pfun X \bullet f \boxplus g = h \Leftrightarrow \dom g \subseteq (\dom f \cup \{0\} \setminus \{\# f\})) \\
\quad \land h = \{n : \mathbb{N} \mid n \in \dom g \bullet n + 1\} \rhd f \cup \\
\qquad \{n : \mathbb{N} \mid n \in \dom g \bullet (n + 1, g[n])\} \\
\hline
\end{array}
$$

Here explains the normal use of the $\boxplus$ operator in this thesis: The left operand of $\boxplus$ is a sequence. The right operand is a set of pairs (*index*, *element*), with *index* of each pair unique over the set. The difference between the use of $\boxplus$ and $\oplus$ on a sequence is that the indices in the right operand are specified as if the sequence indices begin at zero, rather than one. For example, the pair $(0, a)$ in the right operand would make a the first element of the sequence resulted from the $\boxplus$ operator (provided that the left operand is not an empty sequence). The constraint $\dom g \subseteq (\dom f \cup \{0\} \setminus \{\# f\}))$ ensures that the sequence resulted from $\boxplus$ operator must have the same length as the left operand. Note that the declaration of $\boxplus$, as given above, does not forbids its use in other contexts, in which the left operand is not a sequence. However, we shall not encounter such use in this thesis.

abstract type We may represent an array type T by a sequence of its element type $\tau[\![\mathsf{T}_1]\!]$ (*i.e.*, $\seq \tau[\![\mathsf{T}_1]\!]$), with the constraint that the length of the sequence is B, the size of the array declared in the program. Therefore the abstract type of T is given by

$$\tau[\![\mathsf{T}]\!] == \{s : \seq \tau[\![\mathsf{T}_1]\!] \mid \# s = \mathsf{B}\} \qquad \text{(rule 7)}$$

Due to such representation, we have the following identities:

$$\ran \tau[\![\mathsf{T}]\!] \equiv \tau[\![\mathsf{T}_1]\!] \qquad (107)$$

$$\dom \tau[\![\mathsf{T}]\!] \equiv 1..\mathsf{B} \qquad (108)$$

Identity (107) gives the element type of an array type, and identity (108) gives the *bounds* of the array as a set of integers. We may abbreviate $\operatorname{ran} \tau[\![\mathsf{T}]\!]$ by $\tau_e[\![\mathsf{T}]\!]$ and $\operatorname{dom} \tau[\![\mathsf{T}]\!]$ by $\mathcal{B}[\![\mathsf{T}]\!]$.

initialization An initializer for a one dimensional array has the following form:

$$\{\ \mathsf{I}_1,\ \mathsf{I}_2,\ \mathsf{I}_3,\ \ldots,\ \mathsf{I}_\mathsf{B}\ \} \tag{109}$$

where I_1 is the initializer for the first element, I_2 is the initializer for the second element and so on. All of I_1, I_2, ..., and I_B are of type T_1. An initializer for an array must initialize all elements of the array. The initializer may be represented by a *sequence enumeration* in Z as

$$\langle I_1, I_2, I_3, \ldots, I_n \rangle \tag{110}$$

where I_1, I_2, ..., I_n, are representations of initializers I_1, I_2, ..., I_n. We may apply the rules for representing initializers recursively.

Recall that default initialization of a static array is to apply default initialization to all of its elements[14, p.93]. $\mathcal{Z}[\![\mathsf{T}]\!]$, therefore, should be a value of type $\tau[\![\mathsf{T}]\!]$ with the values of all elements being $\mathcal{Z}[\![\mathsf{T}_1]\!]$. This may be represented in Z by a definite description:

$$\mathcal{Z}[\![\mathsf{T}]\!] = (\mu\, \nu_1 : \tau[\![\mathsf{T}]\!] \mid (\forall j : 1..\mathsf{B} \bullet \nu_1(j) = \mathcal{Z}[\![\mathsf{T}_1]\!])) \tag{111}$$

which may alternatively be written as

$$\mathcal{Z}[\![\mathsf{T}]\!] = (\mu\, \nu_1 : \tau[\![\mathsf{T}]\!] \mid (\forall j : \mathcal{B}[\![\mathsf{T}]\!] \bullet \nu_1(j) = \mathcal{Z}[\![\mathsf{T}]\!]) \tag{112}$$

multi-dimensional arrays Consider the following type expression:

$$\mathsf{T_1\ [B_1]\,[B_2]} \tag{113}$$

where $\mathsf{B_1}$ and $\mathsf{B_2}$ are integer constants specifying the *bounds* of the array. Due to left-associativity of subscript expressions, the type expression may be interpreted as a one-dimensional array of length $\mathsf{B_1}$ that has element type $\mathsf{T_1\,[B_2]}$. We may then apply the same results as for one-dimension arrays. Multi-dimensional array types of any dimension may be analyzed similarly.

5.3.4 Structure

When the variable definition define a structure variable, the parameter T is

$$\texttt{struct \{ } \mathsf{T_1\ f_1;\ T_2\ f_2;\ T_3\ f_3;\ \ldots T_n\ f_n;} \texttt{ \}} \tag{114}$$

where $\mathsf{f_1}$, $\mathsf{f_2}$, $\mathsf{f_3}$, ..., $\mathsf{f_n}$ are the components of the structure, and $\mathsf{T_1}$, $\mathsf{T_1}$, $\mathsf{T_1}$, ..., $\mathsf{T_n}$ are their respective types. The type of each component may either be an integer, an array, or another structure.

abstract type We may represent T in Z by a *schema type*. In particular, the abstract type of T is

$$\tau[\![\mathsf{T}]\!] \mathrel{\widehat=} [f_1 : \tau[\![\mathsf{T_1}]\!]; f_2 : \tau[\![\mathsf{T_2}]\!];\ \cdots; f_n : \tau[\![\mathsf{T_n}]\!]] \qquad \text{(rule 8)}$$

The schema components f_1, f_2, ..., f_n represent the structure components $\mathsf{f_1}$, $\mathsf{f_2}$, $\mathsf{f_3}$, ..., $\mathsf{f_n}$, and therefore must respectively be members of the sets have types $\tau[\![\mathsf{T_1}]\!]$, $\tau[\![\mathsf{T_2}]\!]$, ..., and $\tau[\![\mathsf{T_n}]\!]$. The predicate part of the schema is empty because all the type constraints have been included within the declarations of the components.

initialization The initializer for a structure variable must initialize all components of the structure. Let the initializer I consists of $I_1, I_2, \ldots, I_n$, which initialize the structure components $f_1, f_1, \ldots, f_n$ respectively. Then I may be represented in Z by the following definite description.

$$I = (\mu \nu_1 : \tau[\![\mathsf{T}]\!] \mid \nu_1.f_1 = I_1 \wedge \nu_1.f_2 = I_2 \wedge \cdots \wedge \nu_1.f_n = I_n) \qquad (115)$$

where $I_1, I_2, \ldots, I_n$ are corresponding specifications of $I_1, I_2, \ldots, I_n$.

Recall that the type of each structure component is either an integer, an array, or another structure. Each of $I_1, I_2, \ldots, I_n$ must therefore have its form either as an initializer for integer, an array, or another structure. Therefore we may apply the rules for representing initializers recursively until we have the desired representation.

Recall that the default initialization for a static structure variable is to apply default initialization to all its components. We have

$$\mathcal{Z}[\![\mathsf{T}]\!] = (\mu \nu_1 : \tau[\![\mathsf{T}]\!] \mid \nu_1.f_1 = \mathcal{Z}[\![\mathsf{T}_1]\!] \wedge \nu_1.f_2 = \mathcal{Z}[\![\mathsf{T}_2]\!] \wedge \cdots \wedge \nu_1.f_n = \mathcal{Z}[\![\mathsf{T}_n]\!]) \qquad (116)$$

Once again, the identity is recursive.

5.4 Integer Constants

In our C subset, the only constants that may appear in an expression are *integer constants*[14, p.25] of type `int`.

abstract state Since no variable occurs in C, its abstract state is

$$\sigma[\![\mathsf{C}]\!] \mathrel{\widehat=} \emptyset \qquad \text{(rule 9)}$$

where $\emptyset$ represents the *empty schema*, which has no components and predicate part being the predicate *true*.

abstract effect Since evaluating C has no effect, the abstract effect of C is

$$\epsilon[\![\mathsf{C}]\!] \mathrel{\widehat{=}} \not\emptyset \qquad \text{(rule 10)}$$

abstract value Since the result value of C is the constant C, we may either express the abstract value by the constant C itself:

$$\nu[\![\mathsf{C}]\!] = \nu'[\![\mathsf{C}]\!] = \mathsf{C} \qquad \text{(rule 11)}$$

or we may use a definite description

$$\nu[\![\mathsf{C}]\!] = \nu'[\![\mathsf{C}]\!] = (\mu\, \nu_1 : \tau[\![\mathbf{int}]\!] \mid \nu_1 = \mathsf{C}) \qquad \text{(rule 12)}$$

abstract object The abstract object of C is left undefined because a constant does not designate any object.

5.5 Parenthesized Expressions

The general form of a *parenthesized expression*[14, p.185] is (E). Since the only function of the parentheses is to override operator precedence in C, we simply drop them when deriving abstractions. This leads to the following rules.

$$\tau[\![(\mathsf{E})]\!] == \tau[\![\mathsf{E}]\!] \qquad \text{(rule 13)}$$

$$\sigma[\![(\mathsf{E})]\!] \mathrel{\widehat{=}} \sigma[\![\mathsf{E}]\!] \qquad \text{(rule 14)}$$

$$\epsilon[\![(\mathsf{E})]\!] \mathrel{\widehat{=}} \epsilon[\![\mathsf{E}]\!] \qquad \text{(rule 15)}$$

$$\nu[\![(\mathsf{E})]\!] = \nu[\![\mathsf{E}]\!] \qquad \text{(rule 16)}$$

$$\nu'[\![(\mathsf{E})]\!] = \nu'[\![\mathsf{E}]\!] \qquad \text{(rule 17)}$$

$$\omega[\![(\mathsf{E})]\!] = \omega[\![\mathsf{E}]\!] \qquad \text{(rule 18)}$$

$$\omega'[\![(\mathsf{E})]\!] = \omega'[\![\mathsf{E}]\!] \qquad \text{(rule 19)}$$

Each of these rules is valid if and only if the respective abstraction is defined for E.

5.6 Object Designation Expression

An object designation expression designates an object. For our C subset there are three kinds of object designation expressions: variables, array subscripting expressions, and structure component selection expressions.

5.6.1 Variable Name

An expression may consist of a single identifier v.

abstract type The type of the object designated by the expression v is the type of the variable as defined in its definition. Therefore, the abstract type of the expression v is

$$\tau[\![\mathsf{v}]\!] = \tau[\![\mathcal{D}(\mathsf{v})]\!] \qquad \text{(rule 20)}$$

abstract state The expression v operates on the variable v. Therefore, we may define the abstract state of the expression v to be

$$\sigma[\![\mathsf{v}]\!] \mathrel{\hat{=}} \sigma[\![\mathcal{D}(\mathsf{v})]\!] \qquad \text{(rule 21)}$$

abstract effect Evaluating v has no effect. Therefore, its abstract effect is

$$\epsilon[\![\mathsf{v}]\!] \mathrel{\widehat{=}} [\Delta\sigma[\![\mathsf{v}]\!] \mid v' = v] \qquad \text{(rule 22)}$$

This abstraction rule may be abbreviated by xi convention as

$$\epsilon[\![\mathsf{v}]\!] \mathrel{\widehat{=}} [\Xi\sigma[\![\mathsf{v}]\!]] \qquad \text{(rule 23)}$$

abstract object The object that the expression designates is always the variable v regardless of the value of v. Recall from the definition of abstract object that, in the abstract object of an expression, the parent of the object designated by an object designation expression is represented by a variable subscripted with 'o'. Let v_o be that variable. The abstract object of v is then the following:

$$\omega[\![\mathsf{v}]\!] = \omega'[\![\mathsf{v}]\!] = (\mu\,\omega_1 : \tau[\![\mathsf{v}]\!] \mid \omega_1 = v_o) \qquad \text{(rule 24)}$$

abstract value The result value of the expression v when the variable v has a scalar type has the same type as the object that it designates. Therefore, in this case, the abstract value of v, expressed in term of its initial and final states, are, respectively,

$$\nu[\![\mathsf{v}]\!] = (\mu\,\nu_1 : \tau[\![\mathcal{D}(\mathsf{v})]\!] \mid \nu_1 = v) \qquad \text{(rule 25)}$$

$$\nu'[\![\mathsf{v}]\!] = (\mu\,\nu_1 : \tau[\![\mathcal{D}(\mathsf{v})]\!] \mid \nu_1 = v') \qquad \text{(rule 26)}$$

The abstract value of v is left undefined when v designates an object of a composite type.

5.6.2 Array Subscripting Expressions

The general form of an *array subscripting expression* [14, p.186] for our C subset is

$$\mathsf{E}[\mathsf{E}_1] \tag{117}$$

E must designate an object of an array type. E_1 is a *subscript expression* whose result type is `int`.

abstract type $\mathsf{E}[\mathsf{E}_1]$ designates an object of its element type. Therefore, its abstract type is

$$\tau[\![\mathsf{E}[\mathsf{E}_1]]\!] == \tau_e[\![\mathsf{E}]\!] \qquad \text{(rule 27)}$$

abstract state The variables of $\mathsf{E}[\mathsf{E}_1]$ are the combination of those of E and those of E_1. Therefore, the abstract state of $\mathsf{E}[\mathsf{E}_1]$ is

$$\sigma[\![\mathsf{E}[\mathsf{E}_1]]\!] \mathrel{\widehat{=}} \sigma[\![\mathsf{E}]\!] \wedge \sigma[\![\mathsf{E}_1]\!] \qquad \text{(rule 28)}$$

abstract effect The effect of evaluating $\mathsf{E}[\mathsf{E}_1]$ is that of sequential evaluation of E and E_1 in some order. Therefore, assuming a left to right order of evaluation, the abstract effect of $\mathsf{E}[\mathsf{E}_1]$ is

$$\epsilon[\![\mathsf{E}[\mathsf{E}_1]]\!] \mathrel{\widehat{=}} [\Delta\sigma[\![\mathsf{E}[\mathsf{E}_1]]\!] \mid \epsilon[\![\mathsf{E}]\!] \mathbin{;} \epsilon[\![\mathsf{E}_1]\!]] \qquad \text{(rule 29)}$$

abstract object The object designated by $\mathsf{E}[\mathsf{E}_1]$ is the object designated by E indexed according to the subscript expression E_1. Since $\tau[\![\mathsf{E}]\!]$ is a sequence, the object designated by $\mathsf{E}[\mathsf{E}_1]$ corresponds to an element in a sequence the specification. In particular, assuming a left to right order of evaluation, the abstract object of $\mathsf{E}[\mathsf{E}_1]$ expressed in terms of its initial and final states are respectively

$$\omega[\![\mathsf{E}[\mathsf{E}_1]]\!] = \omega[\![\mathsf{E}]\!][\epsilon[\![\mathsf{E}]\!] \,;\, \nu[\![\mathsf{E}_1]\!]] \qquad \text{(rule 30)}$$

$$\omega'[\![\mathsf{E}[\mathsf{E}_1]]\!] = (\omega'[\![\mathsf{E}]\!] \,;\, \epsilon[\![\mathsf{E}_1]\!])[\nu'[\![\mathsf{E}_1]\!]] \qquad \text{(rule 31)}$$

abstract value The type of the result value of the expression $\mathsf{E}[\mathsf{E}_1]$ is the same as that of the object that it designates whenever the object that it designates has a scalar type. In this case, the abstract value of v expressed in term of its initial and final states are respectively:

$$\nu[\![\mathsf{E}[\mathsf{E}_1]]\!] = (\mu\, \nu_1 : \tau[\![\mathsf{E}[\mathsf{E}_1]]\!] \mid \nu_1 = \omega[\![\mathsf{E}]\!]_{[\_/\_o]}[\epsilon[\![\mathsf{E}]\!] \,;\, \nu[\![\mathsf{E}_1]\!]]\ ) \qquad \text{(rule 32)}$$

$$\nu'[\![\mathsf{E}[\mathsf{E}_1]]\!] = (\mu\, \nu_1 : \tau[\![\mathsf{E}[\mathsf{E}_1]]\!] \mid \nu_1 = (\omega'[\![\mathsf{E}]\!]_{[\_'/\_o]} \,;\, \epsilon[\![\mathsf{E}_1]\!])[\nu'[\![\mathsf{E}_1]\!]]\ ) \text{(rule 33)}$$

The partial systematic renamings $[\_/\_o]$ and $[\_'/\_o]$ respectively abstract the retrieval of the value stored at the parent of the object that $\mathsf{E}[\mathsf{E}_1]$ designates at the beginning and at the end of evaluating the expression.

5.6.3 Structure Component Selection Expressions

The general form of a structure component selection expression may be stated as

$$\mathsf{E}.\mathsf{f}_i \qquad (118)$$

where the object designated by the object designation expression E must be of a structure type, with f_i being one of its components.

abstract type The type of $\mathsf{E}.\mathsf{f_i}$ is retrieved from the definition for the structure component $\mathsf{f_i}$ in the program. Therefore, the abstract type of $\mathsf{E}.\mathsf{f_i}$ is

$$\tau[\![\mathsf{E}.\mathsf{f_i}]\!] == \tau[\![\mathcal{D}(\mathsf{f_i})]\!] \qquad \text{(rule 34)}$$

abstract state The variables of $\mathsf{E}.\mathsf{f_i}$ are the same as those of E. Therefore the abstract state of $\mathsf{E}.\mathsf{f_i}$ is

$$\sigma[\![\mathsf{E}.\mathsf{f_i}]\!] \mathrel{\widehat{=}} \sigma[\![\mathsf{E}]\!] \qquad \text{(rule 35)}$$

abstract effect The effect of evaluating $\mathsf{E}.\mathsf{f_i}$ is that of evaluating E Therefore, the abstract effect $\mathsf{E}.\mathsf{f_i}$ is

$$\epsilon[\![\mathsf{E}.\mathsf{f_i}]\!] \mathrel{\widehat{=}} \epsilon[\![\mathsf{E}]\!] \qquad \text{(rule 36)}$$

abstract object The object designated by the expression $\mathsf{E}.\mathsf{f_i}$ is the structure component $\mathsf{f_i}$ within the object (which is a structure) designated by E. Recall that a structure is represented by a variable of schema type. The abstract object of $\mathsf{E}.\mathsf{f_i}$ would be the following schema component selection expressions:

$$\omega[\![\mathsf{E}.\mathsf{f_i}]\!] == \omega[\![\mathsf{E}]\!].f_i \qquad \text{(rule 37)}$$

$$\omega'[\![\mathsf{E}.\mathsf{f_i}]\!] == \omega'[\![\mathsf{E}]\!].f_i \qquad \text{(rule 38)}$$

where f_i is the schema component of $\tau[\![\mathsf{E}]\!]$ that represents the structure component $\mathsf{f_i}$.

abstract value When the structure component $\mathsf{f_i}$ has a scalar type, the result value of the expression $\mathsf{E.f_i}$ will the value stored at the object designated by the expression. Therefore

$$\nu[\![\mathsf{E.f_i}]\!] = (\mu\,\nu_1 : \tau[\![\mathsf{E.f_i}]\!] \mid \nu_1 = \omega[\![\mathsf{E}]\!]_{[\_/\_o]}) \qquad \text{(rule 39)}$$

$$\nu'[\![\mathsf{E.f_i}]\!] = (\mu\,\nu_1 : \tau[\![\mathsf{E.f_i}]\!] \mid \nu_1 = \omega'[\![\mathsf{E}]\!]_{[\_/\_o]}) \qquad \text{(rule 40)}$$

The partial systematic renamings $[\_/\_o]$ and $[\_'/\_o]$ abstract the retrieval of the value stored at the parent of the object that E designates, at the beginning and at the end of evaluating the expression respectively.

5.7 Unary Arithmetic Expressions

The unary arithmetic operators available in our C subset are plus (+) and minus (-).

5.7.1 Unary Plus

The form of a unary plus expression is

$$\mathsf{+E} \qquad (119)$$

For our C subset E must be an expression of type `int`. $\mathsf{+E}$ is defined to be equivalent to `(0)`$\mathsf{+E}$ [14, p.196]. Therefore, we have the following abstractions.

$$\sigma[\![\mathsf{+E}]\!] \mathrel{\widehat{=}} \sigma[\![\mathsf{E}]\!] \qquad \text{(rule 41)}$$

$$\epsilon[\![\mathsf{+E}]\!] \mathrel{\widehat{=}} \epsilon[\![\mathsf{E}]\!] \qquad \text{(rule 42)}$$

$$\nu[\![\mathsf{+E}]\!] = \nu[\![\mathsf{E}]\!] \qquad \text{(rule 43)}$$

$$\nu'[\![\mathsf{+E}]\!] = \nu'[\![\mathsf{E}]\!] \qquad \text{(rule 44)}$$

Abstract object is not defined for $\mathsf{+E}$ because the expression $\mathsf{+E}$ does not designate an object.

5.7.2 Unary Minus

The form of a unary minus expression is

$$\texttt{-}\mathsf{E} \tag{120}$$

For our C subset E must be an expression of type `int`. As one would expect, this will have very similar abstractions as E.

abstract state The abstract state of $\texttt{-}\mathsf{E}$ is the same as that of E. Therefore,

$$\sigma[\![\texttt{-}\mathsf{E}]\!] \mathrel{\widehat{=}} \sigma[\![\mathsf{E}]\!] \qquad \text{(rule 45)}$$

abstract effect The effect of $\texttt{-}\mathsf{E}$ is the same as that of E, with an additional precondition that the unary minus operation must not result in overflow. Therefore, the abstract effect of $\texttt{-}\mathsf{E}$,

$$\epsilon[\![\texttt{-}\mathsf{E}]\!] \mathrel{\widehat{=}} [\Delta\sigma[\![\texttt{-}\mathsf{E}]\!] \mid \epsilon[\![\mathsf{E}]\!] \wedge \mathit{Min} \leq -\nu[\![\mathsf{E}]\!] \leq \mathit{Max}] \qquad \text{(rule 46)}$$

$\nu[\![\mathsf{E}]\!]$ is used here rather than $\nu'[\![\mathsf{E}]\!]$ because we are specifying a precondition.

abstract value The result type of $\texttt{-}\mathsf{E}$ is the same as that of E, which is `int`. The result of evaluating $\texttt{-}\mathsf{E}$ is the arithmetic negation of the result of evaluating E, provided that arithmetic overflow does not occur. Therefore, the abstract value of $\texttt{-}\mathsf{E}$, expressed in terms of its initial state,

$$\nu[\![\texttt{-}\mathsf{E}]\!] = (\mu\, \nu_1 : \tau[\![\texttt{int}]\!] \mid \nu_1 = -\nu[\![\mathsf{E}]\!] \wedge -\nu[\![\mathsf{E}]\!] \in \tau[\![\texttt{int}]\!]) \qquad \text{(rule 47)}$$

We may omit the constraint $-\nu[\![\mathsf{E}]\!] \in \tau[\![\texttt{int}]\!]$ because it is implied by $\nu_1 = -\nu[\![\mathsf{E}]\!]$, in which $-\nu[\![\mathsf{E}]\!]$ is equated with ν_1, which has been declared to be a member of

the set $\tau[\![\texttt{int}]\!]$. Therefore,

$$\nu[\![\texttt{-E}]\!] = (\mu\, \nu_1 : \tau[\![\texttt{int}]\!] \mid \nu_1 = -\nu[\![\texttt{E}]\!]) \qquad \text{(rule 48)}$$

Similarly, the abstract value of -E expressed in terms of its final state,

$$\nu'[\![\texttt{-E}]\!] = (\mu\, \nu_1 : \tau[\![\texttt{int}]\!] \mid \nu_1 = -\nu'[\![\texttt{E}]\!]) \qquad \text{(rule 49)}$$

in which the constraint $-\nu'[\![\texttt{E}]\!] \in \tau[\![\texttt{int}]\!]$ is implicit.

abstract object As with +E, abstract object is not defined for -E,

5.8 Binary Arithmetic Expressions

The general form of a binary arithmetic expression E in our C subset is

$$\texttt{E}_1 \ \texttt{aop} \ \texttt{E}_2 \qquad (121)$$

where both operands $\texttt{E}_1$ and $\texttt{E}_2$ are expressions of type `int`. aop denotes one of the following arithmetic operators: addition(+), subtraction(-), multiplication(*), division(/), remainder(%).

representing arithmetic operators in Z First, we need to represent each arithmetic operator in C by a corresponding arithmetic operator in Z. Here is a mapping between the arithmetic operators in C and those in Z.

C (aop)	+	−	*	/	%
Z (*aop*)	+	−	*	div	mod

Table 4: Mapping between Arithmetic Operators of C and Z

abstract state The variables of E are the those of E_1 and E_2. Therefore, the abstract state of E is

$$\sigma[\![\mathsf{E}]\!] \mathrel{\widehat{=}} \sigma[\![\mathsf{E}_1]\!] \wedge \sigma[\![\mathsf{E}_2]\!] \qquad \text{(rule 50)}$$

abstract effect The effect of E is equivalent to sequential evaluation of its operands in some order, in conjunction with the precondition that the arithmetic operation does not result in overflow. Therefore, assuming left to right order of evaluation, the abstract effect of E is

$$\begin{aligned}\epsilon[\![\mathsf{E}]\!] \mathrel{\widehat{=}} [\Delta\sigma[\![\mathsf{E}]\!] \mid (\epsilon[\![\mathsf{E}_1]\!] \fatsemi \epsilon[\![\mathsf{E}_2]\!]) \wedge \\ \nu[\![\mathsf{E}_1]\!] \mathit{aop} (\epsilon[\![\mathsf{E}_1]\!] \fatsemi \nu[\![\mathsf{E}_2]\!]) \in \tau[\![\mathsf{E}]\!]] \end{aligned} \qquad \text{(rule 51)}$$

where *aop* is the corresponding representation of the arithmetic operator aop in Z. Note that $\nu[\![\mathsf{E}_2]\!]$ is appended to $\epsilon[\![\mathsf{E}_1]\!]$ to ensure that both operands of *aop* are expressed in terms of the initial state of E.

abstract value Since both operands are of type `int`, the result type would also be `int`. The result value of E is the value obtained from applying aop to the result values of evaluating E_1 and E_2, provided that the arithmetic operation does not result in overflow. Therefore, assuming left to right order of evaluation, the abstract value of E expressed in terms of its initial state is

$$\begin{aligned}\nu[\![\mathsf{E}]\!] = (\mu\, \nu_1 : \tau[\![\mathtt{int}]\!] \mid \nu_1 = \nu[\![\mathsf{E}_1]\!] \mathit{aop} (\epsilon[\![\mathsf{E}_1]\!] \fatsemi \nu[\![\mathsf{E}_2]\!]) \\ \wedge\, \nu[\![\mathsf{E}_1]\!] \mathit{aop} (\epsilon[\![\mathsf{E}_1]\!] \fatsemi \nu[\![\mathsf{E}_2]\!])) \in \tau[\![\mathtt{int}]\!] \end{aligned} \qquad \text{(rule 52)}$$

In (rule 52), $\nu[\![\mathsf{E}_2]\!]$ is appended to $\epsilon[\![\mathsf{E}_1]\!]$ to ensure that both operands of *aop* are expressed in terms of the initial state of E. Since $\nu[\![\mathsf{E}_1]\!]$ *aop* $(\epsilon[\![\mathsf{E}_1]\!] \fatsemi \nu[\![\mathsf{E}_2]\!])$ is

equated with ν_1, which is of type `int`, We may remove the constraint $\nu[\![\mathsf{E}_1]\!]$ *aop* $(\epsilon[\![\mathsf{E}_1]\!] \,;\, \nu[\![\mathsf{E}_2]\!]) \in \tau[\![\texttt{int}]\!]$ from (rule 52) and write

$$\nu[\![\mathsf{E}]\!] = (\mu\, \nu_1 : \tau[\![\texttt{int}]\!] \mid \nu_1 = \nu[\![\mathsf{E}_1]\!]\ aop\ (\epsilon[\![\mathsf{E}_1]\!] \,;\, \nu[\![\mathsf{E}_2]\!])) \qquad \text{(rule 53)}$$

Similarly, the abstract value of E expressed in terms of its the final state is

$$\nu'[\![\mathsf{E}]\!] = (\mu\, \nu_1 : \tau[\![\texttt{int}]\!] \mid \nu_1 = (\nu'[\![\mathsf{E}_1]\!] \,;\, \epsilon[\![\mathsf{E}_2]\!])\ aop\ \nu'[\![\mathsf{E}_2]\!]) \qquad \text{(rule 54)}$$

Here, $\epsilon[\![\mathsf{E}_2]\!]$ is appended to $\nu'[\![\mathsf{E}_1]\!]$ to ensure that both operands of *aop* are expressed in terms of the final state of E. The constraint $(\nu'[\![\mathsf{E}_1]\!] \,;\, \epsilon[\![\mathsf{E}_2]\!])\ aop\ \nu'[\![\mathsf{E}_2]\!] \in \tau[\![\texttt{int}]\!]$ is implicit.

5.9 Relational Expressions

The form of a relational expression E is

$$\mathsf{E}_1\ \mathsf{rop}\ \mathsf{E}_2 \qquad (122)$$

where E_1 and E_2 are expressions. In our C subset both expressions are assumed to be of type `int`. The symbol rop represents one of the following relational operators: less than (<), less than or equal to (<=), greater than(>), greater than or equal to (>=), equal (==), and not equal(!=). A relational expression is a boolean expression returning either *true* or *false*. However, C does not define a boolean type and the results of evaluating such expressions are expressed as an integer of type `int` instead. Under normal execution, the result value of a relational expression is either zero (stands for *false*) or one (stands for *true*).

representing relational operators in Z Similar to binary arithmetic expressions, we need to represent the relational operators in C by corresponding relational operators in Z. We also define the *complement operator* $\overline{rop}$ of a relational operator

rop as follows: For any two integers a and b, if a *rop* b is true, then $a\ \overline{rop}\ b$ is false, and *vice versa*. The mapping between relational operators in C and those in Z is given in the following table.

C (rop)	<	<=	>	>=	==	!=
Z (*rop*)	$<$	$\leq$	$>$	$\geq$	$=$	$\neq$
Complement in Z ($\overline{rop}$)	$\geq$	$>$	$\leq$	$<$	$\neq$	$=$

Table 5: Mapping between Relational Operators of C and Z

abstract state The variables of E are those of E_1 and E_2. Therefore, the abstract state of E is

$$\sigma[\![\mathsf{E}]\!] \mathrel{\widehat{=}} \sigma[\![\mathsf{E}_1]\!] \wedge \sigma[\![\mathsf{E}_2]\!] \qquad \text{(rule 55)}$$

abstract effect Unlike the case for arithmetic expressions in which arithmetic overflow may occur, a relational expression executes normally if and only if both operands evaluates without errors. Therefore, the effect of evaluating a relational expression is simply the effect of evaluating its operands in some order. Therefore, assuming left to right order of evaluation, the abstract effect of E is given by

$$\epsilon[\![\mathsf{E}]\!] \mathrel{\widehat{=}} [\Delta\sigma[\![\mathsf{E}]\!] \mid \epsilon[\![\mathsf{E}_1]\!] \fatsemi \epsilon[\![\mathsf{E}_2]\!]] \qquad \text{(rule 56)}$$

abstract value The result type of a relational expression is `int`. The abstract value of E expressed in terms of its initial state is

$$\nu[\![\mathsf{E}]\!] = (\mu\, \nu_1 : \tau[\![\mathtt{int}]\!] \mid \nu[\![\mathsf{E}_1]\!]\ rop\ (\epsilon[\![\mathsf{E}_1]\!] \fatsemi \nu[\![\mathsf{E}_2]\!]) \wedge \nu_1 = 1$$

$$\vee\ \nu[\![\mathsf{E}_1]\!]\ \overline{rop}\ (\epsilon[\![\mathsf{E}_1]\!]\ ;\ \nu[\![\mathsf{E}_2]\!]) \wedge \nu_1 = 0) \qquad \text{(rule 57)}$$

where *rop* is the Z relational operator that represents rop, $\overline{rop}$ represents the complement operator of *rop*. The first disjunct in (rule 57) asserts that the result value of E is zero whenever the relation between the result values of the operands holds. The second disjunct indicates that the result value of E is one whenever the complement of the relation between the result values of the operands holds (*i.e.*, the relation between them does not hold). Note that for any two expressions E_1 and E_2, due to the definition of complement operator, at most one of the two disjuncts, $\nu[\![\mathsf{E}_1]\!]\ rop\ (\epsilon[\![\mathsf{E}_1]\!]\ ;\ \nu[\![\mathsf{E}_2]\!])$, and $\nu[\![\mathsf{E}_1]\!]\ \overline{rop}\ (\epsilon[\![\mathsf{E}_1]\!]\ ;\ \nu[\![\mathsf{E}_2]\!])$, may be true (none is true if at least one of the operands does not execute normally). Similarly, the abstract value of E expressed in terms of its final state is

$$\nu'[\![\mathsf{E}]\!] = (\mu\, \nu_1 : \tau[\![\texttt{int}]\!] \mid (\nu'[\![\mathsf{E}_1]\!]\ ;\ \epsilon[\![\mathsf{E}_2]\!])\ rop\ \nu'[\![\mathsf{E}_2]\!]) \wedge \nu_1 = 1$$
$$\vee\ (\nu'[\![\mathsf{E}_1]\!]\ ;\ \epsilon[\![\mathsf{E}_2]\!])\ \overline{rop}\ \nu'[\![\mathsf{E}_2]\!]) \wedge \nu_1 = 0) \qquad \text{(rule 58)}$$

Once again, $(\nu'[\![\mathsf{E}_1]\!]\ ;\ \epsilon[\![\mathsf{E}_2]\!])\ rop\ \nu'[\![\mathsf{E}_2]\!]$ and $(\nu'[\![\mathsf{E}_1]\!]\ ;\ \epsilon[\![\mathsf{E}_2]\!])\ \overline{rop}\ \nu'[\![\mathsf{E}_2]\!]$ are mutually exclusive.

5.10 Conditional Expressions

A conditional expression E in our C subset has this form:

$$\mathsf{E}_1 ? \mathsf{E}_2 : \mathsf{E}_3 \qquad (123)$$

where the subexpressions E_1, E_2, and E_3 are expressions of type `int`.

abstract state The abstract state of E is the conjunction of the abstract states of its subexpressions. In other words,

$$\sigma[\![\mathsf{E}]\!] \mathrel{\widehat{=}} \sigma[\![\mathsf{E}_1]\!] \wedge \sigma[\![\mathsf{E}_2]\!] \wedge \sigma[\![\mathsf{E}_3]\!] \qquad \text{(rule 59)}$$

abstract effect In executing E, E_1 is first evaluated. If the result is non-zero, then E_2 is evaluated and its result value becomes the result value of E. Otherwise, E_3 is evaluated and its result value becomes the result value of E. Therefore, the abstract effect of E,

$$\begin{aligned}\epsilon[\![\mathsf{E}]\!] \mathrel{\widehat{=}} & [\Delta\sigma[\![\mathsf{E}]\!] \mid (\epsilon[\![\mathsf{E}_1]\!] \fatsemi \epsilon[\![\mathsf{E}_2]\!]) \wedge \nu[\![\mathsf{E}_1]\!] \neq 0 \wedge \overline{z}' = \overline{z}] \\ & \vee [\Delta\sigma[\![\mathsf{E}]\!] \mid (\epsilon[\![\mathsf{E}_1]\!] \fatsemi \epsilon[\![\mathsf{E}_3]\!]) \wedge \nu[\![\mathsf{E}_1]\!] = 0 \wedge \overline{x}' = \overline{x}] \qquad \text{(rule 60)}\end{aligned}$$

where $\overline{x}$ represent those variables that occur only in E_2, $\overline{z}$ represent those variables that occur only in E_3. The first disjunct corresponds to the case when E_1 evaluates to true, and the second disjunct corresponds to the other case. The predicates $\overline{x}' = \overline{x}$ and $\overline{z}' = \overline{z}$ asserts that the variables that appear only in one of E_2 and E_3 should remain constant whenever the corresponding subexpression in which they occur is not evaluated.

abstract value The result type of E depends on the type of E_2 and E_3 [14, p.218]. since both of them has type `int` in our C subset, The result type of E is `int`, too. The abstract value of E in terms of its initial state is given by

$$\begin{aligned}\nu[\![\mathsf{E}]\!] = (\mu\, \nu_1 : \tau[\![\texttt{int}]\!] \mid & \nu_1 = (\epsilon[\![\mathsf{E}_1]\!] \fatsemi \nu[\![\mathsf{E}_2]\!]) \wedge \nu[\![\mathsf{E}_1]\!] \neq 0 \\ & \vee \nu_1 = (\epsilon[\![\mathsf{E}_1]\!] \fatsemi \nu[\![\mathsf{E}_3]\!]) \wedge \nu[\![\mathsf{E}_1]\!] = 0) \qquad \text{(rule 61)}\end{aligned}$$

The first disjunct assert that the result of E should be that of E_2 if E_1 is evaluated to true, and be that of E_3 otherwise. The compositions $\epsilon[\![\mathsf{E}_1]\!] \fatsemi \nu[\![\mathsf{E}_2]\!]$ and $\epsilon[\![\mathsf{E}_1]\!] \fatsemi \nu[\![\mathsf{E}_3]\!]$ ensure that all variables in the definite description represent the initial state of E.

Similarly, The abstract value of E expressed in terms of its final state is given by

$$\nu'[\![\mathsf{E}]\!] = (\mu\, \nu_1 : \tau[\![\texttt{int}]\!] \mid \nu_1 = \nu'[\![\mathsf{E}_2]\!] \wedge (\nu'[\![\mathsf{E}_1]\!] \fatsemi \epsilon[\![\mathsf{E}_2]\!]) \neq 0$$

$$\vee\, \nu_1 = \nu'[\![\mathsf{E}_3]\!] \wedge (\nu'[\![\mathsf{E}_1]\!] \,;\, \epsilon[\![\mathsf{E}_3]\!]) = 0) \qquad \text{(rule 62)}$$

The compositions $\nu'[\![\mathsf{E}_1]\!] \,;\, \epsilon[\![\mathsf{E}_2]\!]$ and $\nu'[\![\mathsf{E}_1]\!] \,;\, \epsilon[\![\mathsf{E}_3]\!]$ ensure that all variables in the definite description represent the final state of E.

The difference between $\nu[\![\mathsf{E}]\!]$ (rule 61) and $\nu'[\![\mathsf{E}]\!]$ (rule 62) is that, while $\nu[\![\mathsf{E}]\!]$ is always defined under normal execution, $\nu'[\![\mathsf{E}]\!]$ is undefined when the following conditions occur simultaneously:

$$\nu'[\![\mathsf{E}_1]\!] \,;\, \epsilon[\![\mathsf{E}_2]\!] \neq 0 \tag{124}$$

$$\nu'[\![\mathsf{E}_1]\!] \,;\, \epsilon[\![\mathsf{E}_3]\!] = 0 \tag{125}$$

$$\nu'[\![\mathsf{E}_2]\!] \neq \nu'[\![\mathsf{E}_3]\!] \tag{126}$$

The following example would illustrate the situation:

Example 4 Let E be a conditional expression

$$\mathsf{E}_1 ? \mathsf{E}_2 : \mathsf{E}_3 \tag{127}$$

where E_1 is `i`, E_2 is `i` and E_3 is `i++`. Variable `i` is of type `int`. This expression is well-behaved in the sense that *lint* (a C program checker) does not complain. The abstractions for the subexpressions E_1, E_2, and E_3 are

$$\nu[\![\mathsf{E}_1]\!] = \nu[\![\mathsf{E}_2]\!] = (\mu\, \nu_1 : \tau[\![\mathbf{int}]\!] \mid \nu_1 = i) \tag{128}$$

$$\nu'[\![\mathsf{E}_1]\!] = \nu'[\![\mathsf{E}_2]\!] = (\mu\, \nu_1 : \tau[\![\mathbf{int}]\!] \mid \nu_1 = i') \tag{129}$$

$$\nu[\![\mathsf{E}_3]\!] = (\mu\, \nu_1 : \tau[\![\mathbf{int}]\!] \mid i < \mathit{Max} \wedge \nu_1 = i) \tag{130}$$

$$\nu'[\![\mathsf{E}_3]\!] = (\mu\, \nu_1 : \tau[\![\mathbf{int}]\!] \mid \mathit{Min} < i' \wedge \nu_1 = i' - 1) \tag{131}$$

$$\epsilon[\![\mathsf{E}_1]\!] \mathrel{\widehat{=}} [\Delta\sigma[\![\mathsf{E}_1]\!] \mid i' = i] \tag{132}$$

$$\epsilon[\![\mathsf{E}_2]\!] \mathrel{\widehat{=}} [\Delta\sigma[\![\mathsf{E}_2]\!] \mid i' = i] \tag{133}$$

$$\epsilon[\![\mathsf{E}_3]\!] \mathrel{\widehat{=}} [\Delta\sigma[\![\mathsf{E}_3]\!] \mid i' = i + 1 \wedge i < \mathit{Max}] \tag{134}$$

To find $\nu[\![\mathsf{E}]\!]$ using (rule 61), we need to perform these compositions:

$$\epsilon[\![\mathsf{E}_1]\!] \mathbin{;} \nu[\![\mathsf{E}_2]\!]$$
$$\epsilon[\![\mathsf{E}_1]\!] \mathbin{;} \nu[\![\mathsf{E}_3]\!] \tag{135}$$

Applying (76),

$$\epsilon[\![\mathsf{E}_1]\!] \mathbin{;} \nu[\![\mathsf{E}_2]\!] = (\mu\, \nu_1 : \tau[\![\mathbf{int}]\!] \mid (\exists\, i' : \tau[\![\mathbf{int}]\!] \bullet P_\epsilon(i, i') \wedge P_\nu(\nu_1, i'))) \tag{136}$$

where P_ϵ is the predicate part of $\epsilon[\![\mathsf{E}_1]\!]$ (132), P_ν is the constraining predicate of $\nu[\![\mathsf{E}_2]\!]$ (128) with i substituted by i'. Therefore,

$$\begin{aligned}\epsilon[\![\mathsf{E}_1]\!] \mathbin{;} \nu[\![\mathsf{E}_2]\!] &= (\mu\, \nu_1 : \tau[\![\mathbf{int}]\!] \mid (\exists\, i' : \tau[\![\mathbf{int}]\!] \bullet i' = i \wedge \nu_1 = i')) \\ &= (\mu\, \nu_1 : \tau[\![\mathbf{int}]\!] \mid \nu_1 = i)\end{aligned} \tag{137}$$

Applying (76),

$$\epsilon[\![\mathsf{E}_1]\!] \mathbin{;} \nu[\![\mathsf{E}_3]\!] = (\mu\, \nu_1 : \tau[\![\mathbf{int}]\!] \mid (\exists\, i' : \tau[\![\mathbf{int}]\!] \bullet P_\epsilon(i, i') \wedge P_\nu(\nu_1, i'))) \tag{138}$$

where P_ϵ is the predicate part of $\epsilon[\![\mathsf{E}_1]\!]$ (132), P_ν is the constraining predicate of $\nu[\![\mathsf{E}_3]\!]$ (130) with i substituted by i'. Therefore,

$$\begin{aligned}\epsilon[\![\mathsf{E}_1]\!] \mathbin{;} \nu[\![\mathsf{E}_3]\!] &= (\mu\, \nu_1 : \tau[\![\mathbf{int}]\!] \mid (\exists\, i' : \tau[\![\mathbf{int}]\!] \bullet i' = i \wedge i < \mathit{Max} \wedge \nu_1 = i')) \\ &= (\mu\, \nu_1 : \tau[\![\mathbf{int}]\!] \mid i < \mathit{Max} \wedge \nu_1 = i)\end{aligned} \tag{139}$$

Then the abstract value of E in terms of initial value of $\mathtt{i}$, from (rule 61), is

$$\begin{aligned}\nu[\![\mathsf{E}]\!] &= (\mu\, \nu_1 : \tau[\![\mathbf{int}]\!] \mid \nu_1 = (\epsilon[\![\mathsf{E}_1]\!] \mathbin{;} \nu[\![\mathsf{E}_2]\!]) \wedge \nu[\![\mathsf{E}_1]\!] \neq 0 \\ &\qquad \vee\, \nu_1 = (\epsilon[\![\mathsf{E}_1]\!] \mathbin{;} \nu[\![\mathsf{E}_3]\!]) \wedge \nu[\![\mathsf{E}_1]\!] = 0) \\ &= (\mu\, \nu_1 : \tau[\![\mathbf{int}]\!] \mid \nu_1 = (\mu\, \nu_1 : \tau[\![\mathbf{int}]\!] \mid \nu_1 = i) \wedge (\mu\, \nu_1 : \tau[\![\mathbf{int}]\!] \mid \nu_1 = i) \neq 0\end{aligned}$$

$$\begin{aligned}
&\vee \nu_1 = (\mu \nu_1 : \tau[\![\mathbf{int}]\!] \mid i < Max \wedge \nu_1 = i) \wedge (\mu \nu_1 : \tau[\![\mathbf{int}]\!] \mid \nu_1 = i) = 0) \\
&= (\mu \nu_1 : \tau[\![\mathbf{int}]\!] \mid \nu_1 = i \wedge i \neq 0 \vee \nu_1 = i \wedge i < Max \wedge i = 0) \\
&= (\mu \nu_1 : \tau[\![\mathbf{int}]\!] \mid \nu_1 = i \wedge i \neq 0 \vee \nu_1 = 0 \wedge i = 0) \qquad (140)
\end{aligned}$$

To find $\nu'[\![\mathsf{E}]\!]$ using (rule 62), we need to perform these compositions:

$$\begin{aligned}
&\nu'[\![\mathsf{E}_1]\!] \; ; \epsilon[\![\mathsf{E}_2]\!] \\
&\nu'[\![\mathsf{E}_1]\!] \; ; \epsilon[\![\mathsf{E}_3]\!] \qquad (141)
\end{aligned}$$

Using (92) we obtain

$$\nu'[\![\mathsf{E}_1]\!] \; ; \epsilon[\![\mathsf{E}_2]\!] = (\mu \nu_1 : \tau[\![\mathbf{int}]\!] \mid (\exists i : \tau[\![\mathbf{int}]\!] \bullet P_\nu(\nu_1, i) \wedge P_\epsilon(i, i'))) \qquad (142)$$

where P_ν is the constraining predicate of $\nu'[\![\mathsf{E}_1]\!]$ (129), with i' substituted by i, P_ϵ is the predicate part of $\epsilon[\![\mathsf{E}_2]\!]$ (133). Therefore,

$$\begin{aligned}
\nu'[\![\mathsf{E}_1]\!] \; ; \epsilon[\![\mathsf{E}_2]\!] &= (\mu \nu_1 : \tau[\![\mathbf{int}]\!] \mid (\exists i : \tau[\![\mathbf{int}]\!] \bullet \nu_1 = i \wedge i' = i)) \\
&= (\mu \nu_1 : \tau[\![\mathbf{int}]\!] \mid \nu_1 = i') \qquad (143)
\end{aligned}$$

Using (92) we obtain

$$\nu'[\![\mathsf{E}_1]\!] \; ; \epsilon[\![\mathsf{E}_3]\!] = (\mu \nu_1 : \tau[\![\mathbf{int}]\!] \mid (\exists i : \tau[\![\mathbf{int}]\!] \bullet P_\nu(\nu_1, i) \wedge P_\epsilon(i, i'))) \qquad (144)$$

where P_ν is the constraining predicate of $\nu'[\![\mathsf{E}_1]\!]$ (129), with i' substituted by i, P_ϵ is the predicate part of $\epsilon[\![\mathsf{E}_3]\!]$ (134). Therefore,

$$\begin{aligned}
\nu'[\![\mathsf{E}_1]\!] \; ; \epsilon[\![\mathsf{E}_3]\!] &= (\mu \nu_1 : \tau[\![\mathbf{int}]\!] \mid (\exists i : \tau[\![\mathbf{int}]\!] \bullet \nu_1 = i \wedge i' = i + 1 \wedge i < Max)) \\
&= (\mu \nu_1 : \tau[\![\mathbf{int}]\!] \mid \nu_1 = i' - 1 \wedge i' - 1 < Max) \\
&= (\mu \nu_1 : \tau[\![\mathbf{int}]\!] \mid \nu_1 = i' - 1 \wedge Min < i') \qquad (145)
\end{aligned}$$

Using (rule 62), the abstract value of E expressed in terms of its final state,

$$\begin{aligned}
\nu'[\![\mathsf{E}]\!] &= (\mu\,\nu_1 : \tau[\![\mathbf{int}]\!] \mid \nu_1 = \nu'[\![\mathsf{E}_2]\!] \wedge (\nu'[\![\mathsf{E}_1]\!] \,;\, \epsilon[\![\mathsf{E}_2]\!]) \neq 0 \\
&\qquad \vee\, \nu_1 = \nu'[\![\mathsf{E}_3]\!] \wedge (\nu'[\![\mathsf{E}_1]\!] \,;\, \epsilon[\![\mathsf{E}_3]\!]) = 0) \\
&= (\mu\,\nu_1 : \tau[\![\mathbf{int}]\!] \mid \\
&\qquad\qquad \nu_1 = (\mu\,\nu_1 : \tau[\![\mathbf{int}]\!] \mid \nu_1 = i') \wedge (\mu\,\nu_1 : \tau[\![\mathbf{int}]\!] \mid \nu_1 = i') \neq 0 \\
&\qquad\qquad \vee\, \nu_1 = (\mu\,\nu_1 : \tau[\![\mathbf{int}]\!] \mid \mathit{Min} < i' \wedge \nu_1 = i' - 1) \\
&\qquad\qquad\qquad \wedge\, (\mu\,\nu_1 : \tau[\![\mathbf{int}]\!] \mid \nu_1 = i' - 1 \wedge \mathit{Min} < i') = 0) \\
&= (\mu\,\nu_1 : \tau[\![\mathbf{int}]\!] \mid \nu_1 = i' \wedge i' \neq 0 \vee \nu_1 = i' - 1 \wedge \mathit{Min} < i' \wedge i' - 1 = 0) \\
&= (\mu\,\nu_1 : \tau[\![\mathbf{int}]\!] \mid \nu_1 = i' \wedge i' \neq 0 \vee \nu_1 = 0 \wedge i' = 1) \qquad (146)
\end{aligned}$$

For $i' = 1$,

$$\nu'[\![\mathsf{E}]\!] = (\mu\,\nu_1 : \tau[\![\mathbf{int}]\!] \mid \nu_1 = 1 \wedge 1 \neq 0 \vee \nu_1 = 0 \wedge 1 = 1) \qquad (147)$$

Eliminating the redundant inequalities $1 \neq 0$ and $1 = 1$ reduces (147) to

$$(\mu\,\nu_1 : \tau[\![\mathbf{int}]\!] \mid \nu_1 = 0 \vee \nu_1 = 1) \qquad (148)$$

The definite description in this case is undefined. This represents a case in which the result value of an expression may not be determined from its final state alone. In this example, given the final value of `i` is one, it is uncertain whether the result of the expression is zero or one. The following table shows the results of executing E for all possible initial values of `i`.

initial value of i	final value of i	result value of E
Min	*Min*	*Min*
...	...	...
−1	−1	−1
0	**1**	**0**
1	**1**	**1**
2	2	2
...	...	...
Max	*Max*	*Max*

Table 6: Results of Evaluating `i?i:i++`

We may observe from the table that for initial value of zero or one for `i`, the final value of `i` in both cases is one, but the result values of E are different between these cases. This observation is accurately recorded in our expression for $\nu'[\![\mathsf{E}]\!]$.

Recall the conditions (124, 125, 126) that causes $\nu'[\![\mathsf{E}]\!]$ to be undefined despite of normal execution. In this case

$$\begin{aligned}\nu'[\![\mathsf{E}_1]\!] \,;\, \epsilon[\![\mathsf{E}_2]\!] \neq 0 &\Leftrightarrow (\mu\, \nu_1 : \tau[\![\mathtt{int}]\!] \mid \nu_1 = i') \neq 0 \\ &\Leftrightarrow i' \neq 0 \qquad (149)\\ \nu'[\![\mathsf{E}_1]\!] \,;\, \epsilon[\![\mathsf{E}_3]\!] = 0 &\Leftrightarrow (\mu\, \nu_1 : \tau[\![\mathtt{int}]\!] \mid \nu_1 = i' - 1 \wedge \mathit{Min} < i') = 0 \\ &\Leftrightarrow \mathit{Min} < i' \wedge i' - 1 = 0 \Leftrightarrow i = 1 \qquad (150)\end{aligned}$$

$$\begin{aligned}\nu'[\![\mathsf{E}_2]\!] \neq \nu'[\![\mathsf{E}_3]\!] &\Leftrightarrow (\mu\, \nu_1 : \tau[\![\mathtt{int}]\!] \mid \nu_1 = i') \\ &\qquad \neq (\mu\, \nu_1 : \tau[\![\mathtt{int}]\!] \mid \mathit{Min} < i' \wedge \nu_1 = i' - 1) \\ &\Leftrightarrow \mathit{Min} < i \qquad (151)\end{aligned}$$

The three conditions are simultaneously true if and only if $i' = 1$. Therefore we may conclude that given that the final value of `i` is one, the result value of E may not be determined.

5.11 Logical Expressions

We shall consider these binary logical operators in C: conjunction (&&) and disjunction (||). They must be dealt with separately from binary arithmetic operators or relational operators due to difference in their semantics. Unlike a binary arithmetic or relational expression, the order of evaluation of a logical expression is always from left to right. In addition, the right operand of a logical operator is evaluated based on the result of evaluating the left operand. Under normal execution, the result of a logical expression is either zero (representing *false*) or one (representing *true*).

5.11.1 Conjunction

An expression with conjunction operator has the following form

$$\mathsf{E} = \mathsf{E}_1 \ \&\& \ \mathsf{E}_2 \tag{152}$$

Our C subset assumes that both expressions E_1 and E_2 are of type `int`. E_1 is evaluated first. E_2 would be executed only if the result value is non-zero (*true*). This is because, if one of the operands of a conjunction evaluates to false, the result of conjunction must be false.

abstract state The abstract state of a conjunction expression is the conjunction of the abstract states of both operands:

$$\sigma[\![\mathsf{E}]\!] \mathrel{\widehat{=}} \sigma[\![\mathsf{E}_1]\!] \wedge \sigma[\![\mathsf{E}_2]\!] \tag{rule 63}$$

abstract effect The abstract effect of E is

$$\epsilon[\![\mathsf{E}]\!] \mathrel{\widehat{=}} [\Delta\sigma[\![\mathsf{E}]\!] \mid (\epsilon[\![\mathsf{E}_1]\!] \fatsemi \epsilon[\![\mathsf{E}_2]\!]) \wedge \nu[\![\mathsf{E}_1]\!] \neq 0]$$

$$\vee\ [\Delta\sigma[\![\mathsf{E}]\!] \mid \epsilon[\![\mathsf{E}_1]\!] \wedge \nu[\![\mathsf{E}_1]\!] = 0 \wedge \overline{x}' = \overline{x}] \qquad \text{(rule 64)}$$

where $\overline{x}$ represents those variables of E that appear only in E_2. The first disjunct in rule (64) asserts that the effect of E_1**&&**E_2 is equivalent to sequential execution of E_1 followed by E_2 whenever E_1 evaluates to non-zero (*true*). The other disjunct asserts that in case E_1 evaluates to zero (*false*), those variables that appear only in E_2 must conserve their values, while the effect is equivalent to executing E_1 alone.

abstract value A conjunction expression evaluates to either zero (for *false*) or one (for *true*). Only one of the following three cases may occur under normal execution:

- E_1 evaluates to zero. In this case the result value of E is zero.
- E_1 evaluates to non-zero but E_2 evaluates to zero. In this case the result value of E is also zero.
- both E_1 and E_2 evaluates to non-zero. In this case the result value of E is one.

In addition, the type of the result value of a conjunction expression is always `int`. Therefore, the abstract value of E, in terms of its initial state,

$$\begin{aligned}\nu[\![\mathsf{E}]\!] = (\mu\,\nu_1 : \tau[\![\mathtt{int}]\!] \mid\ & \nu[\![\mathsf{E}_1]\!] = 0 \wedge \nu_1 = 0 \\ \vee\ & \nu[\![\mathsf{E}_1]\!] \neq 0 \wedge (\epsilon[\![\mathsf{E}_1]\!] \,;\, \nu[\![\mathsf{E}_2]\!]) = 0 \wedge \nu_1 = 0 \\ \vee\ & \nu[\![\mathsf{E}_1]\!] \neq 0 \wedge (\epsilon[\![\mathsf{E}_1]\!] \,;\, \nu[\![\mathsf{E}_2]\!]) \neq 0 \wedge \nu_1 = 1) \qquad \text{(rule 65)}\end{aligned}$$

and that in terms of its final state,

$$\nu'[\![\mathsf{E}]\!] = (\mu\,\nu_1 : \tau[\![\mathtt{int}]\!] \mid \nu'[\![\mathsf{E}_1]\!] = 0 \wedge \nu_1 = 0$$

$$\vee\ (\nu'[\![\mathsf{E}_1]\!]\ ;\ \epsilon[\![\mathsf{E}_2]\!]) \neq 0 \wedge \nu'[\![\mathsf{E}_2]\!] = 0 \wedge \nu_1 = 0$$

$$\vee\ (\nu'[\![\mathsf{E}_1]\!]\ ;\ \epsilon[\![\mathsf{E}_2]\!]) \neq 0 \wedge \nu'[\![\mathsf{E}_2]\!] \neq 0 \wedge \nu_1 = 1) \qquad \text{(rule 66)}$$

Similar to the case for conditional expressions, $\nu'[\![\mathsf{E}]\!]$ is undefined if the following conditions are simultaneously true:

$$\nu'[\![\mathsf{E}_1]\!] = 0 \tag{153}$$

$$\nu'[\![\mathsf{E}_1]\!]\ ;\ \epsilon[\![\mathsf{E}_2]\!] \neq 0 \tag{154}$$

5.11.2 Disjunction

In our C subset, an expression using the disjunction operator will be of the form

$$\mathsf{E} = \mathsf{E}_1 \,||\, \mathsf{E}_2 \tag{155}$$

where E_1 and E_2 are expressions of type `int`.

abstract state The abstract state of a disjunction expression is the conjunction of the abstract states of its operands:

$$\sigma[\![\mathsf{E}]\!] \mathrel{\hat{=}} \sigma[\![\mathsf{E}_1]\!] \wedge \sigma[\![\mathsf{E}_2]\!] \qquad \text{(rule 67)}$$

abstract effect While evaluating E, E_1 is evaluated first. E_2 would be evaluated only if the result value is zero (*false*) This is because a disjunction is true if either of its operands evaluates to true. Therefore the abstract effect of E is

$$\epsilon[\![\mathsf{E}]\!] \mathrel{\hat{=}} [\Delta\sigma[\![\mathsf{E}]\!] \mid (\epsilon[\![\mathsf{E}_1]\!]\ ;\ \epsilon[\![\mathsf{E}_2]\!]) \wedge \nu[\![\mathsf{E}_1]\!] = 0]$$

$$\vee\ [\Delta\sigma[\![\mathsf{E}]\!] \mid \epsilon[\![\mathsf{E}_1]\!] \wedge \nu[\![\mathsf{E}_1]\!] \neq 0 \wedge \overline{x}' = \overline{x}] \qquad \text{(rule 68)}$$

where $\overline{x}$ represents those variables of E that appear only in E_2. The first disjunct in rule (rule 68) asserts that both E_1 and E_2 are executed whenever E_1 evaluates to zero (*false*). The second disjunct asserts that if E_1 evaluates to non-zero (*true*), only E_1 would be evaluated. Therefore, the values of those variables that appear only in E_2 must be conserved.

abstract value The only case in which a disjunction expression evaluates to zero is when both operands evaluates to zero. In addition, if the first operand evaluates to non-zero, the whole expression evaluates to one without evaluating the second operand at all. The the result of a disjunction expression is always of type `int`. Therefore, the abstract value of E, expressed in terms of its initial state is given by

$$\begin{aligned}\nu[\![\mathsf{E}]\!] = (\mu\,\nu_1 : \tau[\![\texttt{int}]\!] \mid\ & \nu[\![\mathsf{E}_1]\!] \neq 0 \wedge \nu_1 = 1 \\ \vee\ & \nu[\![\mathsf{E}_1]\!] = 0 \wedge (\epsilon[\![\mathsf{E}_1]\!] \,;\, \nu[\![\mathsf{E}_2]\!]) \neq 0 \wedge \nu_1 = 1 \\ \vee\ & \nu[\![\mathsf{E}_1]\!] = 0 \wedge (\epsilon[\![\mathsf{E}_1]\!] \,;\, \nu[\![\mathsf{E}_2]\!]) = 0 \wedge \nu_1 = 0) \qquad \text{(rule 69)}\end{aligned}$$

The same quantity expressed in terms of the final state of E is

$$\begin{aligned}\nu'[\![\mathsf{E}]\!] = (\mu\,\nu_1 : \tau[\![\texttt{int}]\!] \mid\ & \nu'[\![\mathsf{E}_1]\!] \neq 0 \wedge \nu_1 = 1 \\ \vee\ & (\nu'[\![\mathsf{E}_1]\!] \,;\, \epsilon[\![\mathsf{E}_2]\!]) = 0 \wedge \nu'[\![\mathsf{E}_2]\!] \neq 0 \wedge \nu_1 = 1 \\ \vee\ & (\nu'[\![\mathsf{E}_1]\!] \,;\, \epsilon[\![\mathsf{E}_2]\!]) = 0 \wedge \nu'[\![\mathsf{E}_2]\!] = 0 \wedge \nu_1 = 0) \qquad \text{(rule 70)}\end{aligned}$$

The conditions under which $\nu'[\![\mathsf{E}]\!]$ is undefined are very similar to those for conjunction expressions:

$$\nu'[\![\mathsf{E}_1]\!] \neq 0 \qquad (156)$$

$$\nu'[\![\mathsf{E}_1]\!] \; ; \; \epsilon[\![\mathsf{E}_2]\!] = 0 \tag{157}$$

5.11.3 Logical Negation

The third logical operator is logical negation, denoted by `!`. The result value of logical negation is either zero (*false*) or one(*true*). In our C subset, the form of a logical negation expression E is

$$!\,\mathsf{E}_1 \tag{158}$$

where E_1 is an expression of type `int`.

abstract state The abstract state of E is the same as the expression being negated, that is E_1. Therefore,

$$\sigma[\![\mathsf{E}]\!] \triangleq \sigma[\![\mathsf{E}_1]\!] \qquad \text{(rule 71)}$$

abstract effect The operand of logical negation (*i.e.*, E_1) is always evaluated. There is no other effect. Therefore

$$\epsilon[\![\mathsf{E}]\!] \triangleq [\Delta\sigma[\![\mathsf{E}]\!] \mid \epsilon[\![\mathsf{E}_1]\!]] \qquad \text{(rule 72)}$$

abstract value The result value of a logical negation is always of type `int`. The result value of a logical negation is zero if the operand evaluates to non-zero (represented by the first disjunct in (rule 73), and is one if the operand evaluates

to zero (represented by the second disjunct in (rule 73). Therefore, the abstract value of E expressed in terms of its initial state,

$$\nu[\![\mathsf{E}]\!] = (\mu\nu_1 : \tau[\![\mathbf{int}]\!] \mid \nu[\![\mathsf{E}_1]\!] = 0 \wedge \nu_1 = 1 \\ \vee\ \nu[\![\mathsf{E}_1]\!] \neq 0 \wedge \nu_1 = 0) \qquad \text{(rule 73)}$$

Similarly, the abstract value of E expressed in terms of its final state is

$$\nu'[\![\mathsf{E}]\!] = (\mu\nu_1 : \tau[\![\mathbf{int}]\!] \mid \nu'[\![\mathsf{E}_1]\!] = 0 \wedge \nu_1 = 1 \\ \vee\ \nu'[\![\mathsf{E}_1]\!] \neq 0 \wedge \nu_1 = 0) \qquad \text{(rule 74)}$$

5.12 Assignment Expressions

5.12.1 Simple Assignment Expressions

We consider simple assignment expressions first. A simple assignment expression has the following form:

$$\mathsf{E}_1 = \mathsf{E}_2 \qquad (159)$$

where E_1 and E_2 are expressions. E_1 must designate an object of a scalar type T. which is assumed to be the same as the result type of E_2.

abstract state The abstract state of an assignment expression is the conjunction of the abstract states of its operands. Therefore,

$$\sigma[\![\mathsf{E}_1 = \mathsf{E}_2]\!] \mathrel{\hat{=}} \sigma[\![\mathsf{E}_1]\!] \wedge \sigma[\![\mathsf{E}_2]\!] \qquad \text{(rule 75)}$$

abstract effect The effect of the assignment expression stems from evaluating E, which may be thought of being performed in two steps:

1. First, evaluate separately the left operand to obtain the object that it designates, and the right operand to obtain its result value.

2. Assign the result value obtained in step one to the object obtained in step one.

Assuming a left to right order of evaluation, the abstract effect of the first step is

$$[\Delta\sigma[\![\mathsf{E}_1{=}\mathsf{E}_2]\!] \mid \epsilon[\![\mathsf{E}_1]\!] \fatsemi \epsilon[\![\mathsf{E}_2]\!]] \tag{160}$$

Let $\mathcal{A}'[\![\mathsf{E}_1{=}\mathsf{E}_2]\!]$ be the operation schema that represents the abstract effect of the second step. First, we need an abstraction for the result of the right operand expressed in terms of the final state of step one, that is $\nu'[\![\mathsf{E}_2]\!]$. We also need an abstraction for the object that the left operand designates, expressed also in terms of the final state of step one, that is $\omega'[\![\mathsf{E}_1]\!] \fatsemi \epsilon[\![\mathsf{E}_2]\!]$. The composition $\omega'[\![\mathsf{E}_1]\!] \fatsemi \epsilon[\![\mathsf{E}_2]\!]$ is necessary because evaluation of E_1 is only an intermediate step within step one so that the variables of E_1 may be further changed by subsequent evaluation of E_2 (Example 3 contains an instance where such composition is needed). Finally we may define $\mathcal{A}'[\![\mathsf{E}_1{=}\mathsf{E}_2]\!]$ to be

$$\begin{aligned}\mathcal{A}'[\![\mathsf{E}_1{=}\mathsf{E}_2]\!] \mathrel{\widehat=} [\Delta\sigma[\![\mathsf{E}_1{=}\mathsf{E}_2]\!]; \sigma[\![\mathsf{E}_1{=}\mathsf{E}_2]\!]_o \mid \\ ((\omega'[\![\mathsf{E}_1]\!] \fatsemi \epsilon[\![\mathsf{E}_2]\!]) :'= \nu'[\![\mathsf{E}_2]\!])_{[\_/\_'][\_'/\_o]} \wedge \overline{x}' = \overline{x}] \setminus (\__o) \text{ (rule 76)}\end{aligned}$$

The purpose of the additional declaration $\sigma[\![\mathsf{E}_1{=}\mathsf{E}_2]\!]_o$ in the signature of $\mathcal{A}'[\![\mathsf{E}_1{=}\mathsf{E}_2]\!]$ is to assign types to those variables in the predicate part that are subscripted with 'o'. They are hidden so that the effect of the assignment is specified only by the relationship between initial and final states. The partial systematic renaming $[\_/\_']$

abstracts the equivalence between the final state of step one and the initial state of step two. The partial systematic renaming $[\_'/\_o]$ abstracts the retrieval of the final value of the parent of the object that is assigned in step two of the assignment expression; In this abstraction rule, $\overline{x}$ and $\overline{x}'$ denotes the initial and final values of the variables of the assignment expression other than the parent of the object that is assigned in step two of the assignment. The syntactic transformation (':'=') will be described shortly.

Using (rule 76), the abstract effect of E is the sequential composition between the abstract effects of the two steps, that is,

$$\epsilon[\![\mathsf{E}_1 = \mathsf{E}_2]\!] \mathrel{\widehat=} [\Delta\sigma[\![\mathsf{E}_1 = \mathsf{E}_2]\!] \mid (\epsilon[\![\mathsf{E}_1]\!] \fatsemi \epsilon[\![\mathsf{E}_2]\!]) \fatsemi \mathcal{A}'[\![\mathsf{E}_1 = \mathsf{E}_2]\!]] \qquad \text{(rule 77)}$$

defining the syntactic transformation ':'= The operator ':'=' takes two expressions (in Z) of particular forms as operands, and evaluates to a predicate. In general, the left operand will be an abstract object sequentially composed to an operation schema, and the right operand will be a definite description. Applying $:'=$, followed with the partial systematic renamings $[\_/\_'][\_'/\_o]$, must result in a predicate that specifies the final value of the parent of the object being assigned in step two in the assignment expression. Formal description of this syntactic transformation requires an extension to Z notation.

We shall define $:'=$ base on the form of the abstract object within its left operand, as follows.

variable In the simplest case, the abstract object in the left operand of $:'=$ is that of a variable, say v. In this case, we may define

$$\omega'[\![\mathsf{v}]\!] \fatsemi S :'= D \stackrel{\text{def}}{=} v_o = D \qquad (161)$$

for any sequence of operation schemas S, composed sequentially, and definite description D. Definition (161) is justified for the fact that $\omega'[\![\mathbf{v}]\!]$ is v_o, which, due to the absent of no primed variables, must remain constant regardless of the operation schema appended to it.

array subscripting expression When the the abstract object in the left operand of $:'=$ is that of an array subscripting expression, we may define

$$\omega'[\![\mathsf{E}_1\,[\mathsf{E}_2]]\!]\ ;\ S :'= D \stackrel{\text{def}}{=} ((\omega'[\![\mathsf{E}_1]\!]\ ;\ \epsilon[\![\mathsf{E}_2]\!])\ ;\ S) :'= (\omega'[\![\mathsf{E}_1]\!]\ ;\ \epsilon[\![\mathsf{E}_2]\!]\ ;\ S)_{[\_'/\_o]} \boxplus \{((\nu'[\![\mathsf{E}_2]\!]\ ;\ S), D)\} \tag{162}$$

By associativity $(\omega'[\![\mathsf{E}_1]\!]\ ;\ \epsilon[\![\mathsf{E}_2]\!])\ ;\ S$ is the same as $\omega'[\![\mathsf{E}_1]\!]\ ;\ (\epsilon[\![\mathsf{E}_2]\!]\ ;\ S)$, which is composition between an abstract object and an operation schema. In addition, the function application[42, p.83] in (162) due to the $\boxplus$ operator implies a definite description as its result. Therefore, we may apply our definition for $:'=$ recursively to the right side of definition (162). The partial systematic renaming $[\_'/\_o]$ corresponds to retrieval of the value of the parent of the object being assigned at the end of the first step of the assignment.

structure component selection expression When the the abstract object in the left operand of $:'=$ is that of a structure component selection expression, we may define

$$\begin{aligned}\omega'[\![\mathsf{E}_1\,.\,\mathsf{f}_i]\!]\ ;\ S :'= D \stackrel{\text{def}}{=} (\omega'[\![\mathsf{E}_1]\!]\ ;\ S) :'= (\mu\,\nu_1 : \tau[\![\mathsf{E}_1]\!] \mid \nu_1.f_1 = ((\omega'[\![\mathsf{E}_1]\!]\ ;\ S).f_1)_{[\_'/\_o]} \\ \wedge\ \nu_1.f_2 = ((\omega'[\![\mathsf{E}_1]\!]\ ;\ S).f_2)_{[\_'/\_o]} \\ \wedge \cdots \wedge\ \nu_1.f_i = D \\ \wedge \cdots \wedge\ \nu_1.f_m = ((\omega'[\![\mathsf{E}_1]\!]\ ;\ S).f_m)_{[\_'/\_o]})\end{aligned} \tag{163}$$

We may apply our definition for $:'=$ recursively to the right hand side of definition (163).

Let us recall from Example 4 that an abstract value expressed in terms of the final state of a construct may be undefined despite normal execution of the construct. In addition, an abstract object expressed in terms the final state of a construct may contain terms which are abstract values expressed in terms of the final state of some constructs. Hence, a disadvantage of (rule 77) is that whenever $\omega'[\![\mathsf{E}_1]\!]$ or $\nu'[\![\mathsf{E}_2]\!]$ is undefined, $\mathcal{A}'[\![\mathsf{E}_1{=}\mathsf{E}_2]\!]$ would also be undefined. The following example illustrates the situation.

Example 5 Consider the following assignment expression:

```
i=(i?i:i++)
```
(164)

where `i` is of type `int`. Although the abstract value of `i?i:i++` expressed in terms of its final state is undefined, the abstract effect of the assignment expression is still defined. In particular, assuming a left to right evaluation,

$$\epsilon[\![\texttt{i=(i?i:i++)}]\!] \mathrel{\hat{=}} [i, i' : \mathit{Min}..\mathit{Max} \mid i' = i \wedge i \neq 0 \vee i' = 1 \wedge i = 0] \quad (165)$$

To eliminate such undefinedness, we may define the abstract effect of step two of a simple assignment, not only in terms of the initial and final state of step two, but also partially in terms of the initial state of the assignment expression itself. In particular, we shall express the following quantities in step two of the assignment in terms of the initial state of step one: (i) the value being assigned, and (ii) the object being assigned. All other quantities must still be expressed in terms of the final state of the first step.

Let us assume a left to right order of evaluation. Let $\mathcal{A}[\![\mathsf{E}_1{=}\mathsf{E}_2]\!]$ be the abstract effect of step two derived using the alternative approach. First, we need an abstraction of the abstract object of E_1 expressed in terms of the initial state of the assignment expression, that is $\omega[\![\mathsf{E}_1]\!]$. We also need an abstraction of the abstract value of E expressed also in terms of the initial state of the assignment expression, that is $\epsilon[\![\mathsf{E}_1]\!] \,;\, \nu[\![\mathsf{E}_2]\!]$.

We may then define $\mathcal{A}[\![\mathsf{E}_1{=}\mathsf{E}_2]\!]$ to be

$$\begin{aligned}\mathcal{A}[\![\mathsf{E}_1{=}\mathsf{E}_2]\!] \mathrel{\widehat{=}} [\Delta\sigma[\![\mathsf{E}_1{=}\mathsf{E}_2]\!];\ \sigma[\![\mathsf{E}_1{=}\mathsf{E}_2]\!]_o;\ \sigma[\![\mathsf{E}_1{=}\mathsf{E}_2]\!]'' \mid \\ (\omega[\![\mathsf{E}_1]\!]_{[\_''/\_]} := (\epsilon[\![\mathsf{E}_1]\!] \,;\, \nu[\![\mathsf{E}_2]\!])_{[\_''/\_]})_{[\_'/\_o]} \\ \wedge\ \overline{x}' = \overline{x}] \setminus (\__o) \qquad \text{(rule 78)}\end{aligned}$$

The purpose of the additional declarations $\sigma[\![\mathsf{E}_1{=}\mathsf{E}_2]\!]_o$ and $\sigma[\![\mathsf{E}_1{=}\mathsf{E}_2]\!]''$ in the signature of $\mathcal{A}[\![\mathsf{E}_1{=}\mathsf{E}_2]\!]$ is to assign types to the variables in the predicate part that are subscripted with 'o' or double-primed. Specifically, $\sigma[\![\mathsf{E}_1{=}\mathsf{E}_2]\!]''$ denotes the initial state of the assignment expression, which we shall use in specifying step two of the assignment in the alternative approach. The declaration $\sigma[\![\mathsf{E}_1{=}\mathsf{E}_2]\!]_o$ is hidden so that the effect of the assignment is specified only by the relationship between initial and final states of step two, as well as the initial state of the assignment expression. The partial systematic renaming $[\_'/\_o]$ abstracts the retrieval of the final value of the parent of the object that is assigned in step two of the assignment expression. $\overline{x}$ and $\overline{x}'$ denote the initial and final values of the variables of the assignment expression other than the parent of the object that is assigned in step two of the assignment. The partial systematic renaming $[\_''/\_]$ in $(\omega[\![\mathsf{E}_1]\!]_{[\_''/\_]}$ and $(\epsilon[\![\mathsf{E}_2]\!] \,;\, \nu[\![\mathsf{E}_2]\!])_{[\_''/\_]}$ correspond to our technique of expressing both the object and the value being assigned in terms of the initial state of the first step. The syntactic transformation $:=$ will be described shortly.

Using (rule 78), we define the abstract effect of E to be the sequential composition between the abstract effects of the two steps:

$$\epsilon[\![\mathsf{E}_1{=}\mathsf{E}_2]\!] \mathrel{\widehat{=}} [\Delta\sigma[\![\mathsf{E}_1{=}\mathsf{E}_2]\!]; \sigma[\![\mathsf{E}_1{=}\mathsf{E}_2]\!]'' \mid (\epsilon[\![\mathsf{E}_1]\!] \fatsemi \epsilon[\![\mathsf{E}_2]\!] \fatsemi \mathcal{A}[\![\mathsf{E}_1{=}\mathsf{E}_2]\!]) \\ \wedge\ \theta\sigma[\![\mathsf{E}_1{=}\mathsf{E}_2]\!] = \theta\sigma[\![\mathsf{E}_1{=}\mathsf{E}_2]\!]''] \setminus (\_'') \qquad \text{(rule 79)}$$

The predicate $\theta\sigma[\![\mathsf{E}_1{=}\mathsf{E}_2]\!]_o = \theta\sigma[\![\mathsf{E}_1{=}\mathsf{E}_2]\!]''$ in (rule 79) is due to our notation of using double-primed variables to denote the initial state of the assignment expression.

defining the syntactic transformation ':='

The operator ':=' takes two expressions (in Z) of particular forms as operands, and evaluates to a predicate. In general, the left operand will be an abstract object renamed by $[\_''/\_]$, and the right operand will be a definite description. Applying ':=', followed with the partial systematic renaming $[\_'/\_o]$, must result in a predicate that specifies the final value of the parent of the object being assigned in step two of an assignment expression, expressed in terms of both the initial and final state of the first step of the assignment. This syntactic transformation may not be defined within the Z notation.

We may define ':=' based on the form of its left operand, as follows.

variable The simplest case is when the abstract object in the left hand side of the ':=' operator is that of a variable, say v. Since $\omega[\![\mathsf{v}]\!]$ contains no unprimed variables, we have

$$\omega[\![\mathsf{v}]\!]_{[\_''/\_]} := D \stackrel{\text{def}}{=} v_o = D \qquad (166)$$

array subscripting expression Another case we shall consider is when the left operand of ':=' is the abstract object of an array subscripting expression. In this case, we may define

$$\omega[\![\mathsf{E}_1\,[\mathsf{E}_2]]\!]_{[\_''/\_]} := D \stackrel{\text{def}}{=} \omega[\![\mathsf{E}_1]\!]_{[\_''/\_]} := \omega[\![\mathsf{E}_1]\!]_{[\_''/\_][\_/\_o]} \boxplus \{((\epsilon[\![\mathsf{E}_1]\!] \,;\, \nu[\![\mathsf{E}_2]\!])_{[\_''/\_]}, D)\} \qquad (167)$$

The function application in (167) due to the $\boxplus$ operator implies a definite description as its result. Therefore we may apply our definition for ':=' recursively to the right side of definition (167). The partial systematic renaming $[\_/\_o]$ corresponds to retrieval of the value of the parent of the object being assigned at the end of the first step of the assignment. It is here because, all objects of the parent of the object being assigned, except the object being assigned, must still be expressed in terms of the *final* state of the first step of the assignment.

structure component selection expression Another case we shall consider is when the the abstract object on the left hand side of the := operator is that of a structure component selection expression. In this case, we may define

$$\begin{aligned}\omega[\![\mathsf{E}_1\,.\,\mathsf{f}_i]\!] := D \stackrel{\text{def}}{=} \omega[\![\mathsf{E}_1]\!]_{[\_''/\_]} := (\mu\,\nu_1 : \tau[\![\mathsf{E}_1]\!] \mid \\ \nu_1.f_1 = (\omega[\![\mathsf{E}_1]\!].f_1)_{[\_''/\_][\_/\_o]} \\ \wedge\, \nu_1.f_2 = (\omega[\![\mathsf{E}_1]\!].f_2)_{[\_''/\_][\_/\_o]} \\ \wedge \cdots \wedge \nu_1.f_i = D \\ \wedge \cdots \wedge \nu_1.f_m = (\omega[\![\mathsf{E}_1]\!].f_m)_{[\_''/\_][\_/\_o]}) \qquad (168)\end{aligned}$$

We may then apply our definition for := recursively to the right hand side of definition (168).

abstract value The result of evaluating an assignment expression is the value being assigned, that is, the result value of the right operand (E_2). Therefore, the abstract value of E_1=E_2,

$$\nu[\![\mathsf{E}_1\texttt{=}\mathsf{E}_2]\!] = \epsilon[\![\mathsf{E}_1]\!] \fatsemi \nu[\![\mathsf{E}_2]\!] \qquad \text{(rule 80)}$$

$$\nu'[\![\mathsf{E}_1\texttt{=}\mathsf{E}_2]\!] = \nu'[\![\mathsf{E}_2]\!] \fatsemi \mathcal{A}'[\![\mathsf{E}_1\texttt{=}\mathsf{E}_2]\!] \qquad \text{(rule 81)}$$

It is impossible to derive $\nu'[\![\mathsf{E}_1\texttt{=}\mathsf{E}_2]\!]$ using $\mathcal{A}[\![\mathsf{E}_1\texttt{=}\mathsf{E}_2]\!]$ because, by definition, $\nu'[\![\mathsf{E}_1\texttt{=}\mathsf{E}_2]\!]$ may contain no knowledge of the *initial* state of the assignment expression. For example, though the abstract effect of

```
i=(i?i:i++)
```
(169)

is well defined, its abstract value, coincidentally being equal to that of `(i?i:i++)` is not always defined despite normal execution.

Example 6 This example illustrates the application of (rule 79) and the syntactic transformation ':=' on the following assignment expression:

```
a[a[i]++] = a[i++]
```
(170)

For brevity we only show the predicate parts of the schemas involved. Assuming left to right order of evaluation, the effect of the first step of the assignment is

$$i' = i + 1 \wedge a' = a \boxplus \{(i, a[i] + 1)\} \qquad (171)$$

Consider step two of the assignment. The term $\omega[\![\mathsf{E}_1]\!]_{[\_''/\_]} := (\epsilon[\![\mathsf{E}_2]\!] \fatsemi \nu[\![\mathsf{E}_2]\!])_{[\_''/\_]}$ in (rule 78) would be

$$\omega[\![\texttt{a[a[i]++]}]\!]_{[\_''/\_]} := (\epsilon[\![\texttt{a[a[i]++]}]\!] \fatsemi \nu[\![\texttt{a[i++]}]\!])_{[\_''/\_]} \qquad (172)$$

which is

$$\omega[\![\texttt{a[a[i]++]}]\!]_{[\_''/\_]} := (a'' \boxplus \{(i'', a''[i''] +1)\})[i''] \tag{173}$$

Let D be $(a'' \boxplus \{(i'', a''[i''] +1)\})[i'']$, which is equal to $a''[i''] +1$, Using (167) (173) becomes

$$\omega[\![\texttt{a}]\!]_{[\_''/\_]} := \omega[\![\texttt{a}]\!]_{[\_''/\_][\_/\_o]} \boxplus \{((\epsilon[\![\texttt{a}]\!] \; ; \nu[\![\texttt{a[i++]}]\!])_{[\_''/\_]}, a''[i''] +1)\} \tag{174}$$

Since $\nu[\![\texttt{a[i++]}]\!] = a[i]$, (173) would be equivalent to

$$\omega[\![\texttt{a}]\!]_{[\_''/\_]} := a \boxplus \{(a''[i''] \, , a''[i''] +1)\} \tag{175}$$

which, from (166), is equivalent to

$$a_o = a \boxplus \{(a''[i''] \, , a''[i''] +1)\} \tag{176}$$

Substitute (176) into (rule 78). The predicate part would be

$$a' = a \boxplus \{(a''[i''] \, , a''[i''] +1)\} \wedge i' = i \tag{177}$$

According to (rule 79), we perform sequential composition between the two steps (that is, (171) and (177)). This yields

$$a' = a \boxplus \{(i, a[i] +1)\} \boxplus \{(a''[i''] \, , a''[i''] +1)\} \wedge i' = i + 1 \tag{178}$$

Finally, substituting double-primed variables with unprimed counterparts produces the following:

$$a' = a \boxplus \{(i, a[i] +1)\} \boxplus \{(a[i] \, , a[i] +1)\} \wedge i' = i + 1 \tag{179}$$

which would be the abstract effect of the assignment expression (170).

5.12.2 Compound Assignment Expressions

The general form of a compound assignment expression E is

$$E_1 \ \mathbf{aop}= \ E_2 \tag{180}$$

where **aop** is an arithmetic operator (+, −, *, /, or %), E_1 and E_2 are expressions, which, in our C subset, are assumed to be of type `int`.

abstract state The abstract state of a compound assignment expression is the conjunction of the abstract states of its operands:

$$\sigma[\![E]\!] \cong \sigma[\![E_1]\!] \wedge \sigma[\![E_2]\!] \tag{rule 82}$$

abstract effect The meaning of E is exactly the same as

$$E_1 \ = \ E_1 \ \mathbf{aop} \ E_2 \tag{181}$$

with the constraint that E_1 is evaluated only once [14, p.222]. Once again, this may be thought of being performed in two steps:

1. Obtain both the result value of, and the object designated by, the left operand, and obtain separately the result value of the right operand.

2. Apply **aop** to the result values obtained in step one, and assign the resulting value to the object that we have obtained in step one.

The effect of the first step is exactly the same as that of simple assignment expression, with the additional constraint that the arithmetic operation must not result

in overflow. Therefore, assuming a left to right order of evaluation, the abstract effect of step one is

$$[\Delta\sigma[\![\mathsf{E}]\!] \mid \epsilon[\![\mathsf{E}_1]\!] \fatsemi \epsilon[\![\mathsf{E}_2]\!] \land \nu[\![\mathsf{E}_1]\!] \; aop \; (\epsilon[\![\mathsf{E}_1]\!] \fatsemi \nu[\![\mathsf{E}_2]\!]) \in \tau[\![\mathtt{int}]\!]] \qquad (182)$$

where *aop* is the operator in Z that corresponds the the operator **aop** in C, as stated in Table 4.

Let $\mathcal{A}'[\![\mathsf{E}]\!]$ be the operation schema that represents the abstract effect of step two. As for the case of simple assignment expressions, we need an abstraction for the object designated by E_1 and an abstraction for the result value of E_2, both expressed in terms of the final state of step one. In addition, we need an abstraction for the result value of E_1, expressed also in terms of the final state of step one, that is $\nu'[\![\mathsf{E}_1]\!] \fatsemi \epsilon[\![\mathsf{E}_2]\!]$. Then, assuming a left to right order of evaluation, $\mathcal{A}'[\![\mathsf{E}]\!]$ may be defined to be

$$\mathcal{A}'[\![\mathsf{E}]\!] \mathrel{\widehat=} [\Delta\sigma[\![\mathsf{E}]\!]; \sigma[\![\mathsf{E}]\!]_o \mid ((\omega'[\![\mathsf{E}_1]\!] \fatsemi \epsilon[\![\mathsf{E}_2]\!]) :'= (\nu'[\![\mathsf{E}_1]\!] \fatsemi \epsilon[\![\mathsf{E}_2]\!]) \; aop \; \nu'[\![\mathsf{E}_2]\!])_{[\_/\_'][\_'/\_o]}$$
$$\land \overline{x}' = \overline{x}] \setminus (\__o) \qquad \text{(rule 83)}$$

where $\overline{x}$ and $\overline{x}'$ respectively represent the initial and final values of the variables of E other than the parent of the object being assigned in step two. Finally, the abstract effect of E is sequential composition of the abstract effect of the two steps:

$$\epsilon[\![\mathsf{E}]\!] \mathrel{\widehat=} [\Delta\sigma[\![\mathsf{E}]\!] \mid \epsilon[\![\mathsf{E}_1]\!] \fatsemi \epsilon[\![\mathsf{E}_2]\!] \fatsemi \mathcal{A}'[\![\mathsf{E}]\!]$$
$$\land \nu[\![\mathsf{E}_1]\!] \; aop \; (\epsilon[\![\mathsf{E}_1]\!] \fatsemi \nu[\![\mathsf{E}_2]\!]) \in \tau[\![\mathtt{int}]\!]] \qquad \text{(rule 84)}$$

Using the alternative way to specify the abstract effect of step two (that is, specify it partially in terms of the initial state of E), the abstract effect of E is

$$\epsilon[\![\mathsf{E}]\!] \mathrel{\widehat=} [\Delta\sigma[\![\mathsf{E}]\!]; \sigma[\![\mathsf{E}]\!]'' \mid (\epsilon[\![\mathsf{E}_1]\!] \fatsemi \epsilon[\![\mathsf{E}_2]\!] \fatsemi \mathcal{A}[\![\mathsf{E}]\!]) \land \theta\sigma[\![\mathsf{E}]\!] = \theta\sigma[\![\mathsf{E}]\!]''$$

$$\wedge\ \nu[\![\mathsf{E}_1]\!]\ aop\ (\epsilon[\![\mathsf{E}_1]\!]\ ;\ \nu[\![\mathsf{E}_2]\!]) \in \tau[\![\mathtt{int}]\!] \setminus (\_'') \qquad \text{(rule 85)}$$

with $\mathcal{A}[\![\mathsf{E}]\!]$ defined as follows:

$$\mathcal{A}[\![\mathsf{E}]\!] \mathrel{\hat{=}} [\Delta\sigma[\![\mathsf{E}]\!];\ \sigma[\![\mathsf{E}]\!]_o;\ \sigma[\![\mathsf{E}]\!]'' \mid$$
$$(\omega[\![\mathsf{E}_1]\!]_{[\_''/\_]} := (\nu[\![\mathsf{E}_1]\!]\ aop\ (\epsilon[\![\mathsf{E}_1]\!]\ ;\ \nu[\![\mathsf{E}_2]\!]))_{[\_''/\_]})_{[\_'/\_o]}$$
$$\wedge\ \overline{x}' = \overline{x}] \setminus (\__o) \qquad \text{(rule 86)}$$

where $\overline{x}$ and $\overline{x}'$ respectively represents the initial and final values of all variables of E other than the parent of the object being assigned in step two.

abstract value The abstract value of E, expressed in terms of its initial state. is,

$$\nu[\![\mathsf{E}]\!] = (\mu\,\nu_1 : \tau[\![\mathtt{int}]\!] \mid$$
$$\nu_1 = \nu[\![\mathsf{E}_1]\!]\ aop\ (\epsilon[\![\mathsf{E}_1]\!]\ ;\ \nu[\![\mathsf{E}_2]\!])) \qquad \text{(rule 87)}$$

with the constraint that the arithmetic operation does not produce overflow, that is, $\nu[\![\mathsf{E}_1]\!]\ aop\ (\epsilon[\![\mathsf{E}_1]\!]\ ;\ \nu[\![\mathsf{E}_2]\!]) \in \tau[\![\mathtt{int}]\!]$, being implicit. Similarly, the abstract value of E, expressed in terms of its final state, is,

$$\nu'[\![\mathsf{E}]\!] = (\mu\,\nu_1 : \tau[\![\mathtt{int}]\!] \mid$$
$$\nu_1 = (\nu'[\![\mathsf{E}_1]\!]\ ;\ \epsilon[\![\mathsf{E}_2]\!]\ ;\ \mathcal{A}'[\![\mathsf{E}]\!])\ aop\ (\nu'[\![\mathsf{E}_2]\!]\ ;\ \mathcal{A}'[\![\mathsf{E}]\!])) \text{ (rule 88)}$$

The composition $\nu'[\![\mathsf{E}_1]\!]\ ;\ \epsilon[\![\mathsf{E}_2]\!]\ ;\ \mathcal{A}'[\![\mathsf{E}]\!]$ is needed because evaluation of E_1 precedes both the evaluation of E_2 and step two (whose abstract effect is $\mathcal{A}'[\![\mathsf{E}]\!]$). The constraint $\nu[\![\mathsf{E}_1]\!]\ ;\ \epsilon[\![\mathsf{E}_2]\!]\ ;\ \mathcal{A}'[\![\mathsf{E}]\!])\ aop\ (\nu'[\![\mathsf{E}_2]\!]\ ;\ \mathcal{A}'[\![\mathsf{E}]\!]) \in \tau[\![\mathtt{int}]\!]$ is implicit in (rule 88). Once again, it is impossible to derive $\nu'[\![\mathsf{E}]\!]$ using $\mathcal{A}[\![\mathsf{E}]\!]$.

5.13 Preincrement and Predecrement Expressions

The general form of a preincrement or predecrement expression E is

$$\mathsf{idop}\ \mathsf{E}_1 \tag{183}$$

where the expression E_1 is of type `int`. The increment/decrement operator idop is either `++` or `--`. E_1 must designate an object.

abstract state The abstract state of E is that of E_1. Therefore,

$$\sigma[\![\mathsf{E}]\!] \mathrel{\widehat{=}} \sigma[\![\mathsf{E}_1]\!] \tag{rule 89}$$

abstract effect The expression E is equivalent to E_1`+=1`. Let $\overline{x}$ and $\overline{x}'$ represents the initial and final values of all variables of E other than the parent of the object designated by E respectively. Then the abstract effect of E, applying (rule 84), is

$$\begin{aligned}\epsilon[\![\mathsf{E}]\!] &\mathrel{\widehat{=}} [\Delta\sigma[\![\mathsf{E}]\!] \mid (\epsilon[\![\mathsf{E}_1]\!] \fatsemi \epsilon[\![1]\!] \fatsemi \mathcal{A}'[\![\mathsf{E}]\!]) \wedge \nu[\![\mathsf{E}_1]\!]\ aop\ (\epsilon[\![\mathsf{E}_1]\!] \fatsemi \nu[\![1]\!]) \in \tau[\![\mathtt{int}]\!]] \\ &\mathrel{\widehat{=}} [\Delta\sigma[\![\mathsf{E}]\!] \mid (\epsilon[\![\mathsf{E}_1]\!] \fatsemi \mathcal{A}'[\![\mathsf{E}]\!]) \wedge \nu[\![\mathsf{E}_1]\!]\ aop\ 1 \in \tau[\![\mathtt{int}]\!]] \qquad \text{(rule 90)}\end{aligned}$$

with $\mathcal{A}'[\![\mathsf{E}]\!]$ defined accordingly as

$$\begin{aligned}\mathcal{A}'[\![\mathsf{E}]\!] &\mathrel{\widehat{=}} [\Delta\sigma[\![\mathsf{E}]\!]; \sigma[\![\mathsf{E}]\!]_o \mid ((\omega'[\![\mathsf{E}_1]\!] \fatsemi \epsilon[\![1]\!]) :'= (\nu'[\![\mathsf{E}_1]\!] \fatsemi \epsilon[\![1]\!])\ aop\ \nu'[\![1]\!])_{[\_/\_'][\_'/\_o]} \\ &\qquad \wedge \overline{x}' = \overline{x}] \\ &\mathrel{\widehat{=}} [\Delta\sigma[\![\mathsf{E}]\!]; \sigma[\![\mathsf{E}]\!]_o \mid (\omega'[\![\mathsf{E}_1]\!] :'= \nu'[\![\mathsf{E}_1]\!]\ aop\ \nu'[\![1]\!])_{[\_/\_'][\_'/\_o]} \\ &\qquad \wedge \overline{x}' = \overline{x}] \qquad \text{(rule 91)}\end{aligned}$$

Alternatively, the abstract effect of E, using (rule 85), is

$$\epsilon[\![\mathsf{E}]\!] \mathrel{\widehat{=}} [\Delta\sigma[\![\mathsf{E}]\!]; \sigma[\![\mathsf{E}]\!]'' \mid$$

$$(\epsilon[\![\mathsf{E}_1]\!] \mathbin{;} \epsilon[\![1]\!] \mathbin{;} \mathcal{A}[\![\mathsf{E}]\!]) \wedge \nu[\![\mathsf{E}_1]\!]\ aop\ (\epsilon[\![\mathsf{E}_1]\!] \mathbin{;} \nu[\![1]\!]) \in \tau[\![\mathtt{int}]\!] \setminus (\_'')$$
$$\widehat{=}\ [\Delta\sigma[\![\mathsf{E}]\!]; \sigma[\![\mathsf{E}]\!]'' \mid$$
$$(\epsilon[\![\mathsf{E}_1]\!] \mathbin{;} \mathcal{A}[\![\mathsf{E}]\!]) \wedge \nu[\![\mathsf{E}_1]\!]\ aop\ 1 \in \tau[\![\mathtt{int}]\!] \setminus (\_'') \qquad \text{(rule 92)}$$

where $\mathcal{A}[\![\mathsf{E}]\!]$ is defined accordingly to be

$$\mathcal{A}[\![\mathsf{E}]\!] \widehat{=} [\Delta\sigma[\![\mathsf{E}]\!]; \sigma[\![\mathsf{E}]\!]_o; \sigma[\![\mathsf{E}]\!]'' \mid (\omega[\![\mathsf{E}_1]\!]_{[\_''/\_]} := \nu[\![\mathsf{E}_1]\!]_{[\_''/\_]}\ aop\ 1)_{[\_'/\_o]}$$
$$\wedge\ \overline{x}' = \overline{x}] \qquad \text{(rule 93)}$$

abstract value Applying (rule 87) we obtain

$$\nu[\![\mathsf{E}]\!] = (\mu\nu_1 : \tau[\![\mathtt{int}]\!] \mid \nu_1 = \nu[\![\mathsf{E}_1]\!]\ aop\ (\epsilon[\![\mathsf{E}_1]\!] \mathbin{;} \nu[\![1]\!]))$$
$$= (\mu\nu_1 : \tau[\![\mathtt{int}]\!] \mid \nu_1 = \nu[\![\mathsf{E}_1]\!]\ aop\ 1) \qquad \text{(rule 94)}$$

Applying (rule 88) we obtain

$$\nu'[\![\mathsf{E}]\!] = (\mu\nu_1 : \tau[\![\mathtt{int}]\!] \mid \nu_1 = ((\nu'[\![\mathsf{E}_1]\!] \mathbin{;} \epsilon[\![1]\!]) \mathbin{;} \mathcal{A}'[\![\mathsf{E}]\!])\ aop\ (\nu'[\![1]\!] \mathbin{;} \mathcal{A}'[\![\mathsf{E}]\!]))$$
$$= (\mu\nu_1 : \tau[\![\mathtt{int}]\!] \mid \nu_1 = (\nu'[\![\mathsf{E}_1]\!] \mathbin{;} \mathcal{A}'[\![\mathsf{E}]\!])\ aop\ 1) \qquad \text{(rule 95)}$$

5.14 Postincrement and Postdecrement Expressions

The general form of a preincrement or predecrement expression E is

$$\mathsf{E}_1\ \mathsf{idop} \qquad (184)$$

where expression E_1 has type `int`. The increment/decrement operator idop is either ++ or --.

abstract state and effect The abstract states and abstract effects of E_1 **idop** and **idop** E_1 are the same. Therefore, we have the following abstraction rules:

$$\sigma[\![\mathsf{E}]\!] \mathrel{\widehat{=}} \sigma[\![\mathsf{E}_1]\!] \tag{rule 96}$$

$$\epsilon[\![\mathsf{E}]\!] \mathrel{\widehat{=}} [\Delta\sigma[\![\mathsf{E}]\!] \mid (\epsilon[\![\mathsf{E}_1]\!] \mathbin{;} \mathcal{A}'[\![\mathsf{E}]\!]) \wedge \nu[\![\mathsf{E}_1]\!] \mathit{aop}\ 1 \in \tau[\![\mathbf{int}]\!]] \tag{rule 97}$$

$$\begin{aligned}\epsilon[\![\mathsf{E}]\!] \mathrel{\widehat{=}} [\Delta\sigma[\![\mathsf{E}]\!]; \sigma[\![\mathsf{E}]\!]'' \mid \\ (\epsilon[\![\mathsf{E}_1]\!] \mathbin{;} \mathcal{A}[\![\mathsf{E}]\!]) \wedge \nu[\![\mathsf{E}_1]\!] \mathit{aop}\ 1 \in \tau[\![\mathbf{int}]\!] \setminus (\_'')\end{aligned} \tag{rule 98}$$

with $\mathcal{A}'[\![\mathsf{E}]\!]$ and $\mathcal{A}[\![\mathsf{E}]\!]$ defined respectively as

$$\begin{aligned}\mathcal{A}'[\![\mathsf{E}]\!] \mathrel{\widehat{=}} [\Delta\sigma[\![\mathsf{E}]\!]; \sigma[\![\mathsf{E}]\!]_o \mid (\omega'[\![\mathsf{E}_1]\!] :'= \nu'[\![\mathsf{E}_1]\!] \mathit{aop}\ 1)_{[\_/\_'][\_'/\_o]} \\ \wedge\ \overline{x}' = \overline{x}]\end{aligned} \tag{rule 99}$$

$$\begin{aligned}\mathcal{A}[\![\mathsf{E}]\!] \mathrel{\widehat{=}} [\Delta\sigma[\![\mathsf{E}]\!]; \sigma[\![\mathsf{E}]\!]_o; \sigma[\![\mathsf{E}]\!]'' \mid (\omega[\![\mathsf{E}_1]\!]_{[\_''/\_]} := \nu[\![\mathsf{E}_1]\!]_{[\_''/\_]} \mathit{aop}\ 1)_{[\_'/\_o]} \\ \wedge\ \overline{x}' = \overline{x}]\end{aligned} \tag{rule 100}$$

abstract value The result value of E is the result value of E_1, with the constraint that the arithmetic operation must not produce an overflow. Therefore,

$$\nu[\![\mathsf{E}]\!] = (\mu\nu_1 : \tau[\![\mathbf{int}]\!] \mid \nu_1 = \nu[\![\mathsf{E}_1]\!] \wedge \nu[\![\mathsf{E}_1]\!] \mathit{aop}\ 1 \in \tau[\![\mathbf{int}]\!]) \tag{rule 101}$$

Unlike (rule 94), the constraint for the arithmetic operation to be valid, that is $\nu[\![\mathsf{E}_1]\!]\ \mathit{aop}\ 1 \in \tau[\![\mathbf{int}]\!]$, must be explicitly stated in (rule 101), because this is not implied by $\nu_1 = \nu[\![\mathsf{E}_1]\!]$. Similarly, the abstract value of E, expressed in terms its final state, is,

$$\begin{aligned}\nu'[\![\mathsf{E}]\!] = (\mu\nu_1 : \tau[\![\mathbf{int}]\!] \mid \nu_1 = (\nu'[\![\mathsf{E}_1]\!] \mathbin{;} \mathcal{A}'[\![\mathsf{E}]\!]) \\ \wedge\ (\nu'[\![\mathsf{E}_1]\!] \mathbin{;} \mathcal{A}'[\![\mathsf{E}]\!])\ \mathit{aop}\ 1 \in \tau[\![\mathbf{int}]\!])\end{aligned} \tag{rule 102}$$

5.15 Sequential Control Structures

We have dealt with techniques for finding abstractions of definitions and expressions. We now proceed to find abstractions of control structures. For the purpose of this thesis, control structures are classified into

statements statements include expression statements, if-statements, if-then-else statements, and blocks.

sequences of statements sequential execution of statements is expressed in C by a sequence of statements.

function definitions A function definition has the form `T f() B`, where `f` is the *name* of the function, `T` is the *return type* of the function, and `B` is a block which is the *body* of the function. For our C subset, `T` is always `void`, that is, functions do not return any value.

program A program consists of a sequence of variable and function definitions. One of the definitions must be a function named `main`. A C program is executed by invoking the `main` function.

The only abstractions defined for sequential control structures are abstract effect and abstract state. Other abstractions are left undefined.

5.15.1 Expression Statement

An expression may be made into a statement by appending to it the statement terminator, semicolon (`;`). For example, `i++` is an expression, while `i++;` is an *expression statement.* The expression is evaluated but the value is discarded; only its effect remains. Therefore, for an expression statement `E;`, its abstract effect is

the same as that of the expression E, that is,

$$\epsilon[\![\mathsf{E};]\!] \triangleq [\Delta\sigma[\![\mathsf{E};]\!] \mid \epsilon[\![\mathsf{E}]\!]] \qquad \text{(rule 103)}$$

Since E; operates on exactly the set of variables as E, its abstract state is

$$\sigma[\![\mathsf{E};]\!] \triangleq \sigma[\![\mathsf{E}]\!] \qquad \text{(rule 104)}$$

5.15.2 Function Invocation

Our C subset assumes that a function invocation must occur on its own as an expression statement, that is:

```
f();                                                    (185)
```

where f is the name of the function invoked. If f is *non-recursive*, that is, its body may only call non-recursive functions, we may substitute the body of f in place of f();. Therefore, the abstract state and abstract effect of f(); are respectively

$$\sigma[\![\mathsf{f}();]\!] \triangleq \sigma[\![\text{body of f}]\!] \qquad \text{(rule 105)}$$

and

$$\epsilon[\![\mathsf{f}();]\!] \triangleq [\Delta\sigma[\![\mathsf{f}();]\!] \mid \epsilon[\![\text{body of f}]\!]] \qquad \text{(rule 106)}$$

Since the body of a function is a block, its abstractions may be calculated using (rule 108) and (rule 107).

5.15.3 Block

Any statement in C is written within a block, delimited by braces ({}). A *block* consists of a sequence of variable definitions followed by a sequence of statements:

Consider a block that has form

$$\{\mathsf{DS}\} \tag{186}$$

where D, a sequence of variable definitions, consists of local definitions $\mathsf{D}_{\mathsf{local}}$ and static definitions $\mathsf{D}_{\mathsf{static}}$. S is a sequence of statements. We may define the effect of the block as the effect of executing S appended to the effect of initializations performed by D, with static variables definitions skipped and local variables hidden. Therefore the abstract effect of the block is

$$\epsilon[\![\mathsf{DS}]\!] \mathrel{\widehat=} [\Delta\sigma[\![\mathsf{DS}]\!] \mid (\epsilon[\![\mathsf{D}_{\mathsf{local}}]\!] \fatsemi \epsilon[\![\mathsf{S}]\!]) \setminus (v_{local}, v'_{local})] \qquad \text{(rule 107)}$$

where v_{local} and v'_{local} represents the initial and final states values of the local variables of the block in its execution. The operator $(\setminus)$ is the hiding operator in Z.

The abstract state of the block is

$$\sigma[\![\mathsf{DS}]\!] \mathrel{\widehat=} \sigma[\![\mathsf{S}]\!] \setminus (v_{local}, v'_{local}) \qquad \text{(rule 108)}$$

Hiding of local variables from the abstractions of a block is necessary because, Since the values of the local variables defined in a block are not retained when control exits from the block, these variables should not be considered as part of the states of enclosing constructs.

Static variable definitions are skipped (note the absence of $\mathsf{D}_{\mathsf{static}}$ in (rule 107)) because we want to separate the effect of executing the block that applies every time the block is entered, from the initialization of static variables, which only occurs once before the very first entry into the block.

The effect of initializing static variables is included as the effect of function definitions, discussed below.

5.15.4 Function Definition

In general, the abstract effect of a function definition (as opposed to the function body, which is a block) may be defined as the sequential composition of the abstract effect of all the static variable definitions throughout the body of the function. Consider a function f defined to be

$$\texttt{void f()}\{\mathsf{D}_1 \cdots \{\mathsf{D}_2 \cdots \{\cdots \{\mathsf{D}_n \cdots\} \cdots\} \cdots\} \cdots\} \tag{187}$$

where $\mathsf{D}_1, \mathsf{D}_2, \ldots, \mathsf{D}_n$ are sequences of declarations. Let D_1 consists of local variable definitions $\mathsf{D}_{1_{\text{local}}}$ and static variables definitions $\mathsf{D}_{1_{\text{static}}}$, D_2 consists of $\mathsf{D}_{2_{\text{local}}}$ and $\mathsf{D}_{2_{\text{static}}}$, and so on. The abstract effect of the function definition is then

$$\epsilon[\![\mathsf{f}]\!] \mathrel{\widehat{=}} [\Delta\sigma[\![\mathsf{f}]\!] \mid \epsilon[\![\mathsf{D}_{1_{\text{static}}}]\!] \mathbin{;} \epsilon[\![\mathsf{D}_{2_{\text{static}}}]\!] \mathbin{;} \cdots \mathbin{;} \epsilon[\![\mathsf{D}_{n_{\text{static}}}]\!]] \qquad \text{(rule 109)}$$

The variables of the function definition are the static variables defined therein. Therefore its abstract state is

$$\sigma[\![\mathsf{f}]\!] \mathrel{\widehat{=}} \sigma[\![\mathsf{D}_{1_{\text{static}}}]\!] \wedge \sigma[\![\mathsf{D}_{2_{\text{static}}}]\!] \wedge \cdots \wedge \sigma[\![\mathsf{D}_{n_{\text{static}}}]\!] \qquad \text{(rule 110)}$$

Comparing these abstractions against those of a variable definition, we may view a function definition as a global variable of a structure type, with components being the static variables defined anywhere within the function body. Then the (default or explicit) initializers for these static variables may be viewed as the initializers for the components of the global variable.

5.15.5 Sequence of Statements

Sequential execution of statements is represented in C simply by writing these statements in sequence. Let S be the following sequence of n statements

$$\mathsf{S}_1\,;\mathsf{S}_2;\mathsf{S}_3;\cdots;\mathsf{S}_n\,; \tag{188}$$

Its abstract effect is therefore the sequential composition of the abstract effect of the statements, *i.e.*,

$$\epsilon[\![\mathsf{S}]\!] \mathrel{\widehat{=}} [\Delta\sigma[\![\mathsf{S}]\!] \mid \epsilon[\![\mathsf{S}_1]\!] \mathbin{;} \epsilon[\![\mathsf{S}_2]\!] \mathbin{;} \cdots \mathbin{;} \epsilon[\![\mathsf{S}_n]\!]] \qquad \text{(rule 111)}$$

Similarly, the abstract state of S would be the conjunction of the abstract states of all the statements in the sequence:

$$\sigma[\![\mathsf{S}]\!] \mathrel{\widehat{=}} \sigma[\![\mathsf{S}_1]\!] \wedge \sigma[\![\mathsf{S}_2]\!] \wedge \cdots \wedge \sigma[\![\mathsf{S}_n]\!]] \qquad \text{(rule 112)}$$

5.15.6 Program

A program consists of a sequence of variable definitions and function definitions. The abstract effect of a program P is an abstraction of initializations followed by invocation of the `main` function. Therefore the abstract effect of P is sequential composition of all the global declarations and the effect of invoking the `main` function:

$$\epsilon[\![\mathsf{P}]\!] \mathrel{\widehat{=}} [\Delta\sigma[\![\mathsf{P}]\!] \mid \epsilon[\![\mathsf{D}_{\mathsf{global}}]\!] \mathbin{;} \epsilon[\![\texttt{main();}]\!]] \qquad \text{(rule 113)}$$

If `main` is non-recursive (which is the case for our C subset) we may substitute the body of `main` into `main();`. In other words,

$$\epsilon[\![\mathsf{P}]\!] \mathrel{\widehat{=}} [\Delta\sigma[\![\mathsf{P}]\!] \mid \epsilon[\![\mathsf{D}_{\mathsf{global}}]\!] \mathbin{;} \epsilon[\![\text{body of } \texttt{main}]\!] \qquad \text{(rule 114)}$$

Since the body of `main` is a block, we may use (rule 108) and (rule 107) to calculate $\epsilon[\![\text{body of } \texttt{main}]\!]$. Similarly, the abstract state of the program, when the main function is non-recursive, is

$$\sigma[\![\mathsf{P}]\!] \mathrel{\widehat{=}} \sigma[\![\mathsf{D}_{\mathsf{global}}]\!] \wedge \sigma[\![\text{body of } \texttt{main}]\!] \qquad \text{(rule 115)}$$

that is, the conjunction of the abstract states for all the variables that the program may operate on.

5.15.7 If-Statements

Conditional statements in C may be represented either by an if-statement or an if-then-else statement. For an if-statement S of form

```
if (E) S1                                                    (189)
```

where E is any expression of integer type, S_1 is any statement, the abstract state of S is the aggregation of the abstract state of the *condition expression* (E) and the *body* (S_1):

$$\sigma[\![S]\!] \mathrel{\widehat{=}} \sigma[\![E]\!] \wedge \sigma[\![S_1]\!] \qquad \text{(rule 116)}$$

In executing S, E is first evaluated. S_1 is then executed only if the result value is non-zero (true). Therefore the abstract effect of S,

$$\begin{aligned} \epsilon[\![S]\!] \mathrel{\widehat{=}} & [\Delta\sigma[\![S]\!] \mid \epsilon[\![E]\!] \,;\, \epsilon[\![S_1]\!] \wedge \nu[\![E]\!] \neq 0] \\ & \vee [\Delta\sigma[\![S]\!] \mid \epsilon[\![E]\!] \wedge \nu[\![E]\!] = 0 \wedge \overline{x}' = \overline{x}] \end{aligned} \qquad \text{(rule 117)}$$

where $\overline{x}$ represent those variables of the if-statement that occur only in S_1. The first disjunction in (117) corresponds to the case when the condition expression is evaluated to *true* (non-zero) and therefore the body also gets executed. In this case the effect is the composition of the effect of the condition expression and the body. The second disjunct corresponds to the case in which the condition expression evaluates to *false*. In this case the body is not executed and therefore the abstract effect is just the abstract effect of the condition expression and the assertion that those variables that appear only in the body may not be changed.

5.15.8 If-then-else Statements

For an if-then-else statement S of form

```
if (E) S1 else S2                                            (190)
```

where the condition expression (E) is any expression of integer type, both the *then-part* (S_1) and the *else-part* (S_2) may be any statements. The abstract state of S in this case is the conjunction of the abstract states of the condition expression, and that of then-part and that of else-part:

$$\sigma[\![\mathsf{S}]\!] \mathrel{\hat{=}} \sigma[\![\mathsf{E}]\!] \wedge \sigma[\![\mathsf{S}_1]\!] \wedge \sigma[\![\mathsf{S}_2]\!] \qquad \text{(rule 118)}$$

In executing S, E is first executed. If the result is non-zero (*true*) then the then-part (S_1) would be executed. Otherwise the else-part (S_2) would be executed. Therefore the abstract effect of S,

$$\begin{aligned} \epsilon[\![\mathsf{S}]\!] \mathrel{\hat{=}} & [\Delta\sigma[\![\mathsf{S}]\!] \mid (\epsilon[\![\mathsf{E}]\!] \,;\, \epsilon[\![\mathsf{S}_1]\!]) \wedge \nu[\![\mathsf{E}_1]\!] \neq 0 \wedge \overline{z}' = \overline{z}] \\ & \vee [\Delta\sigma[\![\mathsf{S}]\!] \mid (\epsilon[\![\mathsf{E}]\!] \,;\, \epsilon[\![\mathsf{S}_2]\!]) \wedge \nu[\![\mathsf{E}_1]\!] = 0 \wedge \overline{x}' = \overline{x}] \qquad \text{(rule 119)} \end{aligned}$$

where $\overline{x}$ represent those variables of the if-then-else statement that occur only in S_1, $\overline{z}$ represent those variables that appear only in S_2. The first disjunct corresponds to the case when the condition is satisfied and then-part of the if-then-else statement is executed. The abstract effect in this case is the sequential composition of the abstract effect of the condition expression and the then-part, and the constraint that the values of the those variables that appear only in S_2 remains constant. The second disjunct corresponds to the case when the condition is not satisfied and the else-part of the if-then-else statement is executed. The abstract effect in this case is the sequential composition of the abstract effect of the condition expression and the else-part, in conjunction the constraint that those variables that occur only in S_1 may not be altered. Note that the (rule 119) is exactly the same as that for conditional expressions (see (rule 60)).

5.16 Iterative Statements

There are three kinds of iterative statements in C[14, p.241]: while-statement, do-while statement, and for-statement. We shall first derive abstractions for a while-statement, then use the results to derive the abstractions for the other two.

5.16.1 While-Statement

Let us consider a *simplified* while-statement W of form

```
while (E) B                                                    (191)
```

where E, an expression of type `int`, is the *condition expression* of the while-statement; B, a statement, is the *loop body* of the while-statement. For the simplified while-statement the condition expression has null effect.

abstract state The abstract state of W is the conjunction of those of B and E. Therefore,

$$\sigma[\![\mathsf{W}]\!] \mathrel{\widehat{=}} \sigma[\![\mathsf{E}]\!] \wedge \sigma[\![\mathsf{B}]\!] \qquad \text{(rule 120)}$$

abstract effect Let $\overline{v}$ represents the variables of W, and $\overline{x}$ represents those variables that occur in E but not in B. We may write the abstract effect of W as

$$\begin{aligned}\epsilon[\![\mathsf{W}]\!] \mathrel{\widehat{=}} {} & [\Xi\sigma[\![\mathsf{W}]\!] \mid \nu[\![\mathsf{E}]\!] = 0] \\ & \vee [\Delta\sigma[\![\mathsf{W}]\!] \mid \nu[\![\mathsf{E}]\!] \neq 0 \wedge (\exists \overline{v}'' \bullet \overline{x}'' = \overline{x} \wedge \epsilon[\![\mathsf{B}]\!][\_''/\_'] \\ & \qquad \wedge \epsilon[\![\mathsf{W}]\!][\_''/\_])] \qquad \text{(rule 121)}\end{aligned}$$

where $\overline{v}$ and $\overline{v}'$ respectively represents the initial and final state (if the loop terminates normally) of the while-statement. The first disjunct represents the case

in which the condition expression is evaluated to false and therefore the loop is not entered and no change is allowed on the state. The second disjunct represents the complementary case, that is, the condition expression is satisfied. In this case, the effect must be the same as executing the loop body once, then uses the intermediate state, represented by $\overline{v}''$, as the initial state to execute W.

The difference between (rule 121) and *all* previous rules is the recurrence of the abstraction being defined (that is, $\epsilon[\![\mathsf{W}]\!]$) on the right hand side of its definition.

(rule 121) may be written equivalently as

$$\begin{aligned}\epsilon[\![\mathsf{W}]\!] \mathrel{\widehat{=}} & [\Xi\sigma[\![\mathsf{W}]\!] \mid \nu[\![\mathsf{E}]\!] = 0] \\ & \vee [\Delta\sigma[\![\mathsf{W}]\!] \mid \nu[\![\mathsf{E}]\!] \neq 0 \wedge \epsilon[\![\mathsf{B}]\!] \fatsemi \epsilon[\![\mathsf{W}]\!]] \qquad \text{(rule 122)}\end{aligned}$$

proving a proposed predicate part for $\epsilon[\![\mathsf{W}]\!]$ Since Z does not allow recursive schema declarations, (rule 122), as it is, may not be used to specify the effect of the while-statement. However, for a sufficiently short or clearly written while-statement, a domain expert may be able to propose the predicate part of $\epsilon[\![\mathsf{W}]\!]$ and verify it against (rule 122) by checking whether the following predicate is a tautology:

$$\begin{aligned}\epsilon[\![\mathsf{W}]\!] \Leftrightarrow & \nu[\![\mathsf{E}]\!] = 0 \wedge \theta\sigma[\![\mathsf{W}]\!] = \theta\sigma[\![\mathsf{W}]\!]' \\ & \vee \nu[\![\mathsf{E}]\!] \neq 0 \wedge \epsilon[\![\mathsf{B}]\!] \fatsemi \epsilon[\![\mathsf{W}]\!] \qquad (192)\end{aligned}$$

when the proposed predicate part of $\epsilon[\![\mathsf{W}]\!]$ is substituted.

Mili[22] discussed several approaches to verifying an iterative statement by induction. The approach closest to (rule 122) is *subgoal induction theorem*[22, p.165], which may be paraphrased in our notations as

$$\epsilon[\![\mathsf{W}]\!] \mathrel{\widehat{=}} [\Delta\sigma[\![\mathsf{W}]\!] \mid \nu[\![\mathsf{E}]\!] = 0 \Rightarrow \epsilon[\![\mathsf{W}]\!]_{[\_/\_']}$$

$$\wedge\, \nu[\mathsf{E}] \neq 0 \wedge \epsilon[\mathsf{B}] \fatsemi \epsilon[\mathsf{W}] \Rightarrow \epsilon[\mathsf{W}]] \qquad \text{(rule 123)}$$

(rule 122) and (rule 123) are logically different because (rule 123) is satisfied whenever E does *not* executes normally, due to the implications in the rule, which is not the case for (rule 122).

recursive characterization of $\sigma[\mathsf{W}]$ Recall that (rule 122) is not a valid declaration in Z. To work around the problem one may rewrite (rule 122) in form of an axiomatic definition that define the relation between initial and final states under normal execution of the while-statement.

$$\Gamma : \sigma[\mathsf{W}] \leftrightarrow \sigma[\mathsf{W}]$$

$$\begin{aligned}\forall \overline{v}; \overline{v}' \bullet \Gamma(\theta\sigma[\mathsf{W}], \theta\sigma[\mathsf{W}]') \Leftrightarrow\ & (\nu[\mathsf{E}] = 0 \wedge \theta\sigma[\mathsf{W}] = \theta\sigma[\mathsf{W}]) \\ & \vee (\nu[\mathsf{E}] \neq 0 \wedge (\exists \overline{v}'' \bullet \overline{x}'' = \overline{x} \wedge \epsilon[\mathsf{B}][\_''/\_'] \wedge \Gamma(\theta\sigma[\mathsf{W}]'', \theta\sigma[\mathsf{W}]')\end{aligned}$$

Note that in the above axiomatic description, Γ is defined recursively.

We may then use Γ to specify the abstract effect of W to be

$$\epsilon[\mathsf{W}] \mathrel{\widehat{=}} [\Delta\sigma[\mathsf{W}] \mid \Gamma(\theta\sigma[\mathsf{W}], \theta\sigma[\mathsf{W}]')] \qquad \text{(rule 124)}$$

This approach is direct but it neither asserts whether Γ exists, nor any possibility of obtaining a non-recursive definition for Γ.

fixed point characterization of $\epsilon[\mathsf{W}]$ Another approach to characterize the effect of a (simplified) while loop is as a fixed point of a function [13, p.120][41, p.61]. Comparing to the use of an recursive axiomatic definition in (rule 124), this approach trades a recursive definition for a union of infinite number of sets. We shall examine a construction of $\sigma[\mathsf{W}]$ using the techniques given in ([41]).

fixed point theorem [41, p.53] We say that $x \in \mathrm{dom}\, f$ is a *fixed point* of a function f if $f(x) = x$. Let R be a function between sets that takes the following

form in Z:

$$
\begin{array}{|l}
[Y] \\
\hline
R : \mathbb{P}\, Y \pfun \mathbb{P}\, Y \\
\hline
\forall B : \mathbb{P}\, Y \bullet R(B) = \{y : Y \mid (\exists X : \mathbb{P}\, Y \mid X \subseteq B \bullet P(X, y))\}
\end{array}
$$

where $P(X, y)$ is a predicate. The essence of the result is that, if for each of every elements of $R(B)$, there exists a finite set X such that $P(X, y)$ is true, that is, the presence of a corresponding finite subset in B is sufficient to establish the presence of any element in $R(B)$, then R has *least fixed point*, defined to be

$$\mathrm{fix}\, R == \bigcup \{n : \mathbb{N} \mid R^n(\varnothing)\} \tag{193}$$

Let specification variables $\overline{v}$ represent the program variables of the simplified while-statement, and specification variables $\overline{x}$ represent those program variables that occur in the condition expression but not in the loop body. Let us define the function W as follow:

$$
\begin{array}{|l}
W : (\sigma[\![\mathsf{W}]\!] \leftrightarrow \sigma[\![\mathsf{W}]\!]) \pfun (\sigma[\![\mathsf{W}]\!] \leftrightarrow \sigma[\![\mathsf{W}]\!]) \\
\hline
\forall B : \sigma[\![\mathsf{W}]\!] \leftrightarrow \sigma[\![\mathsf{W}]\!] \bullet \\
\quad W(B) = \{\overline{v}\ \overline{v}' \mid \nu[\![\mathsf{E}]\!] = 0 \land \theta\sigma[\![\mathsf{W}]\!] = \theta\sigma[\![\mathsf{W}]\!]' \\
\qquad \lor (\exists \overline{v}'' \mid (\theta W'', \theta W') \in B \land\land \epsilon[\![\mathsf{B}]\!][\_''/\_'] \land \overline{x}'' = \overline{x}) \\
\qquad\quad \land \nu[\![\mathsf{E}]\!] \neq 0 \bullet (\theta W, \theta W')\}
\end{array}
$$

Then, according to [41, p.59], the least fixed point of W, that is, fix W, represents the relationship between initial and final state of the simplified while-statement under normal execution. Therefore, we may represent the abstract state of W as an operation schema using W:

$$\epsilon[\![\mathsf{W}]\!] \mathrel{\widehat=} [\Delta\sigma[\![\mathsf{W}]\!] \mid (\mathrm{fix}\, W)(\theta\mathsf{W}, \theta\mathsf{W}')] \qquad \text{(rule 125)}$$

Consider a general while statement with condition expression E and loop body B:

```
while (E) B
```
(194)

For our C subset the condition expression E is assumed to be of type `int`, but may have arbitrary effect. This may be rewritten *conceptually* as

```
while (E_ν) { E_ϵ B } E_ϵ
```
(195)

where E_ν is an expression that has no effect but has the same result value as that of E. E_ϵ is another expression that has the same effect as E. E_ν, E_ϵ and E have the following relationships:

$$\epsilon[\![\mathsf{E}_\epsilon]\!] \triangleq \epsilon[\![\mathsf{E}]\!] \tag{196}$$

$$\nu[\![\mathsf{E}_\nu]\!] = \nu'[\![\mathsf{E}_\nu]\!] = \nu[\![\mathsf{E}]\!] \tag{197}$$

Asserting $\nu[\![\mathsf{E}_\nu]\!]$ and $\nu'[\![\mathsf{E}_\nu]\!]$ to be equal to $\nu[\![\mathsf{E}]\!]$ rather than $\nu'[\![\mathsf{E}]\!]$ is justified as follows: We have conceptually divided the expression E into two expressions, E_ν and E_ϵ that must execute sequentially, with E_ν always precedes E_ϵ. This means that the initial values of the variables as seen by E_ν must always be the same as those seen by E. Therefore $\nu[\![\mathsf{E}_\nu]\!] = \nu[\![\mathsf{E}]\!]$. In addition, since E_ν has no effect, $\nu[\![\mathsf{E}_\nu]\!] = \nu'[\![\mathsf{E}_\nu]\!]$. Therefore $\nu'[\![\mathsf{E}_\nu]\!] = \nu[\![\mathsf{E}]\!]$ as well. Since the condition expression of the rewritten while loop has null effect, we may use the results from simplified while-statement to perform the abstractions. We need not have explicit forms for E_ϵ or E_ν in order to derive abstractions. Specifically, we may use (rule 125) to obtain the following with $\epsilon[\![\mathsf{W}]\!]$ replaced by

$$\epsilon[\![\mathsf{W}]\!] \triangleq [\Delta\sigma[\![\mathsf{W}]\!] \mid (\text{fix } W)(\theta\sigma[\![\mathsf{W}]\!], \theta\sigma[\![\mathsf{W}]\!]')] \; ; \epsilon[\![\mathsf{E}]\!] \qquad \text{(rule 126)}$$

and W correspondingly declared as the following:

$$
\begin{array}{|l}
W : (\sigma[\mathsf{W}] \leftrightarrow \sigma[\mathsf{W}]) \nrightarrow (\sigma[\mathsf{W}] \leftrightarrow \sigma[\mathsf{W}]) \\
\hline
\forall B : \sigma[\mathsf{W}] \leftrightarrow \sigma[\mathsf{W}] \bullet \\
\quad W(B) = \{\overline{v}\ \overline{v}' \mid \nu[\mathsf{E}] = 0 \wedge \theta\sigma[\mathsf{W}] = \theta\sigma[\mathsf{W}]' \\
\qquad \vee (\exists \overline{v}'' \mid (\theta W'', \theta W') \in B \wedge\wedge (\epsilon[\mathsf{B}] \fatsemi \epsilon[\mathsf{E}])[\_''/\_'']) \\
\qquad\quad \wedge \nu[\mathsf{E}] \neq 0 \bullet (\theta W, \theta W')\}
\end{array}
$$

Unlike the previous definition for W for a simplified while-statement, $\overline{x}'' = \overline{x}$ no longer occur in the above axiomatic definition. This is because, in the above axiomatic definition, $\overline{x}'' = \overline{x}$ has been implied by the composition $(\epsilon[\mathsf{E}] \fatsemi \epsilon[\mathsf{B}])[\_''/\_']$.

5.16.2 Do-While Statement

The general form for a *do-while statement* W is

```
do B while ( E );
```
(198)

where B is a statement that is the body of the loop. E is the condition expression of type `int`.

abstract state The abstract state of a do-while statement is the aggregation of the abstract states of the condition expression and the loop body. Therefore,

$$\sigma[\mathsf{W}] \mathrel{\widehat{=}} \sigma[\mathsf{B}] \fatsemi \sigma[\mathsf{E}] \qquad \text{(rule 127)}$$

abstract effect W may be transformed conceptually into the following sequence of statements that contains a while-statement:

$$\mathsf{B}\ \texttt{while}\ (\mathsf{E}_\nu)\ \{\ \mathsf{E}_\epsilon\ \mathsf{B}\ \}\ \mathsf{E}_\epsilon \qquad (199)$$

This looks almost exactly the same as a while-statement except for additional instance of B at the very beginning. Therefore, if we choose to use a fixed point characterization of ϵ[W], we may use the same declaration for W as for a general while-statement, with ϵ[W] defined to be

$$\epsilon[\mathsf{W}] \mathrel{\hat{=}} \epsilon[\mathsf{B}] \mathbin{;} [\Delta\sigma[\mathsf{W}] \mid (\text{fix } W)(\theta\sigma[\mathsf{W}], \theta\sigma[\mathsf{W}]')] \mathbin{;} \epsilon[\mathsf{E}] \qquad \text{(rule 128)}$$

5.16.3 For-Statement

The general form for a *for-statement* W is

```
for ( E1 ; E2 ; E3 ) B
```
(200)

where B is a statement that is the body of the loop. E_1 is executed first. Then the condition expression E_2, that is of type `int`, is evaluated. If it is non-zero, B is executed, followed by the execution of E_3. The process is repeated until E_2 evaluates to *false*.

abstract state The abstract state of a for-statement is the aggregation of the abstract states of E_1, E_2, E_3, and the loop body. Therefore,

$$\sigma[\mathsf{W}] \mathrel{\hat{=}} \sigma[\mathsf{E}_1] \wedge \sigma[\mathsf{E}_2] \wedge \sigma[\mathsf{E}_3] \wedge \sigma[\mathsf{B}] \qquad \text{(rule 129)}$$

abstract effect W may be transformed conceptually into the following sequence of statements containing a *while-statement*:

```
E1; while ( E2v ) { E2e; B E3; } E2e;
```
(201)

where

$$\epsilon[\![\mathsf{E}_{2_\epsilon}]\!] \mathrel{\widehat{=}} \epsilon[\![\mathsf{E}_2]\!] \tag{202}$$

$$\nu[\![\mathsf{E}_{2_\nu}]\!] \mathrel{\widehat{=}} \nu[\![\mathsf{E}_2]\!] \tag{203}$$

and E_{2_ν} has null effect. We may use (rule 126) to characterize $\epsilon[\![\mathsf{W}]\!]$ by a fixed point of a function. Then the abstract effect would be

$$\epsilon[\![\mathsf{W}]\!] \mathrel{\widehat{=}} \epsilon[\![\mathsf{E}_1]\!] \mathbin{;} [\Delta\sigma[\![\mathsf{W}]\!] \mid (\mathrm{fix}\, R)(\theta\sigma[\![\mathsf{W}]\!], \theta\sigma[\![\mathsf{W}]\!]')] \mathbin{;} \epsilon[\![\mathsf{E}_2]\!] \quad \text{(rule 130)}$$

with W declared to be

$$\begin{array}{|l} W : (\sigma[\![\mathsf{W}]\!] \leftrightarrow \sigma[\![\mathsf{W}]\!]) \nrightarrow (\sigma[\![\mathsf{W}]\!] \leftrightarrow \sigma[\![\mathsf{W}]\!]) \\ \hline \forall B : \sigma[\![\mathsf{W}]\!] \leftrightarrow \sigma[\![\mathsf{W}]\!] \bullet \\ \quad W(B) = \{\overline{v}\ \overline{v}' \mid \overline{z}' = \overline{z} \wedge (\nu[\![\mathsf{E}_2]\!] = 0 \wedge \theta\sigma[\![\mathsf{W}]\!] = \theta\sigma[\![\mathsf{W}]\!]' \\ \qquad \vee (\exists \overline{v}'' \mid (\theta W'', \theta W') \in B \wedge\wedge (\epsilon[\![\mathsf{E}_2]\!] \mathbin{;} \epsilon[\![\mathsf{B}]\!] \mathbin{;} \epsilon[\![\mathsf{E}_3]\!])[\_''/\_']) \\ \qquad\quad \wedge \nu[\![\mathsf{E}_2]\!] \neq 0) \bullet (\theta W, \theta W')\} \end{array}$$

where $\overline{z}$ represents those variables of W that occur only in E_1.

5.17 Break and Continue Statements

Ward[39] used a variable to hold the depth of iteration at all times in order to analyze programs that may break from iterations of arbitrary depth.

Since both break and continue statements concern only with the innermost enclosing iterative statement, we only need to define two local variables for every iterative statement. Here is how the three kinds of iterative statements may be rewritten to include these variables:

while-statement A while-statement of form

```
while (E) B
```
(204)

may be transformed into the following block:

```
{    int β=0;
     while ((β==0) && E) {
         int κ=0;
         guard(B)
     }
}                                                    (205)
```

The flag β tells the while-statement whether a **break** statement has been executed. The flag κ tells the while-statement whether a **continue** statement has been executed. At most one of them may be set at any time. *guard*(B) is a syntactic transformation of B such that the flags are tested and set properly.

do-while statement Similarly, a *do-while statement* of form

```
do B while (E)                                       (206)
```

may be transformed into the following block:

```
{    int β=0;
     do {
         int κ=0;
         guard(B)
     } while ((β==0) && E)
}                                                    (207)
```

for-statement A *for-statement* of form

```
for (E1 ; E2 ; E3) B                                 (208)
```

may be transformed into the following block:

```
{    int β=0;
       for (E1 ; (β==0) && E2 ; (β==0) && E3) {
           int κ=0;
           guard(B)
       }
}                                                          (209)
```

characterizing *guard* We need not attempt to specify *guard* completely. Instead, depending on the kind of construct B, we shall derive the abstract state and effect for *guard*(B).

break statement Unless the continue flag has been set, the break flag should be set at this point. This would prevent further effect until the end of the loop body. Therefore,

$$\sigma[\![guard(\texttt{break;})]\!] \mathrel{\widehat{=}} [\beta, \kappa : \tau[\![\texttt{int}]\!]] \qquad \text{(rule 131)}$$

$$\epsilon[\![guard(\texttt{break;})]\!] \mathrel{\widehat{=}} [\Delta\sigma[\![guard(\texttt{break;})]\!] \mid (\kappa = 0 \wedge \beta' = 1) \vee (\kappa = 1 \wedge \beta' = \beta)] \qquad \text{(rule 132)}$$

continue statement Unless the break flag has been set, the continue flag should be set at this point. This would prevent further effect until the end of the loop

body.

$$\sigma[\![guard(\texttt{continue};)]\!] \mathrel{\widehat{=}} [\beta, \kappa : \tau[\![\texttt{int}]\!]] \qquad \text{(rule 133)}$$

$$\epsilon[\![guard(\texttt{continue};)]\!] \mathrel{\widehat{=}} [\Delta\sigma[\![guard(\texttt{continue};)]\!] \mid (\beta = 0 \wedge \kappa' = 1) \vee (\beta = 1 \wedge \kappa' = \kappa)] \qquad \text{(rule 134)}$$

sequence of statements For every statement in a sequence of statements, we need to check whether a **break** or **continue** has been executed. Therefore,

$$\sigma[\![guard(\mathsf{S}_1\mathsf{S}_2 \cdots \mathsf{S}_n)]\!] \mathrel{\widehat{=}} \sigma[\![guard(\mathsf{S}_1)]\!] \wedge \sigma[\![guard(\mathsf{S})]\!] \wedge \cdots \wedge \sigma[\![guard(\mathsf{S}_n)]\!] \qquad \text{(rule 135)}$$

$$\epsilon[\![guard(\mathsf{S}_1\mathsf{S}_2 \cdots \mathsf{S}_n)]\!] \mathrel{\widehat{=}} \epsilon[\![guard(\mathsf{S}_1)]\!] \,;\, \epsilon[\![guard(\mathsf{S})]\!] \,;\, \cdots \,;\, \epsilon[\![guard(\mathsf{S}_n)]\!] \qquad \text{(rule 136)}$$

where S_1, S_2, ..., S_n are statements in the sequence of statements.

block Let B be a block of form

$$\{\ \overline{D}_{local} \cup \overline{D}_{static}\ \mathsf{S}\ \} \qquad (210)$$

If neither flags have been set, all local variables of the block must be initialized, and control passed to the first statement in the block. In this case, for every statement in the block we need to check whether a **break** or **continue** has been executed. Therefore, in the first disjunct in (rule 5.17) we apply *guard* recursively to these

statements. The second disjunct in (rule 5.17) specifies that if either flags have been set before the block is entered, the effect must be null. Therefore,

$$\sigma[\![guard(\mathsf{B})]\!] \mathrel{\widehat{=}} \sigma[\![\mathsf{B}]\!] \wedge [\beta, \kappa : \tau[\![\mathbf{int}]\!]] \qquad \text{(rule 137)}$$

$$\begin{aligned}\epsilon[\![guard(\mathsf{B})]\!] \mathrel{\widehat{=}} [\Delta\sigma[\![\mathsf{B}]\!] \mid \kappa = 0 \wedge \beta = 0 \wedge \kappa' = 0 \wedge \beta' = 0 \wedge \theta\sigma[\![\mathsf{B}]\!]' = \theta\sigma[\![\mathsf{B}]\!]\ ;\\ (([\epsilon[\![\overline{D}_{local}]\!]\ ;\ \epsilon[\![guard(\mathsf{S})]\!]) \setminus (\overline{D}_{local}, \overline{D}'_{local}))\\ \vee\ [\Xi\sigma[\![guard(\mathsf{B}]\!] \mid \kappa = 1 \vee \beta = 1])\end{aligned} \qquad \text{(rule 138)}$$

expression statement The effect of an expression statement is asserted only when none of the flags have been set. Therefore,

$$\sigma[\![guard(\mathsf{E};)]\!] \mathrel{\widehat{=}} [\sigma[\![\mathsf{E}]\!]; \beta, \beta', \kappa, \kappa' : \tau[\![\mathbf{int}]\!]] \qquad \text{(rule 139)}$$

$$\begin{aligned}\epsilon[\![guard(\mathsf{E};)]\!] \mathrel{\widehat{=}} [\Delta\sigma[\![guard(\mathsf{E};)]\!] \mid \beta = \beta' \wedge \kappa = \kappa'\\ \wedge\ \beta = 0 \wedge \kappa = 0 \wedge \theta\sigma[\![\mathsf{E}]\!]' = \theta\sigma[\![\mathsf{E}]\!]\ ;\ \epsilon[\![\mathsf{E}]\!]\\ \vee\ [\Xi\sigma[\![guard(\mathsf{E})]\!] \mid \beta = 1 \vee \kappa = 1]\end{aligned} \qquad \text{(rule 140)}$$

if-statement Let B be an if-statement of the following form:

```
if (E) S                                                    (211)
```

The effect of an if-statement is asserted only if no flags have been set. In addition, *guard* must be applied to the body of the if-statement. Therefore,

$$\sigma[\![guard(\mathsf{B})]\!] \mathrel{\widehat{=}} [\sigma[\![\mathsf{B}]\!]; \beta, \beta', \kappa, \kappa' : \tau[\![\mathbf{int}]\!]] \qquad \text{(rule 141)}$$

$$\epsilon[\![guard(\mathsf{B})]\!] \mathrel{\hat{=}} [\Delta\sigma[\![guard(\mathsf{B})]\!] \mid \beta = \beta' \wedge \kappa = \kappa' \wedge \beta = 0 \wedge \kappa = 0$$
$$\wedge\ \theta\sigma[\![\mathsf{B}]\!]' = \theta\sigma[\![\mathsf{B}]\!]] \mathbin{;} [\epsilon[\![guard(\mathsf{S})]\!] \mid \nu[\![\mathsf{E}]\!] \neq 0]$$
$$\vee\ [\Xi\sigma[\![guard(\mathsf{B})]\!] \mid \beta = 1 \vee \kappa = 1] \qquad \text{(rule 142)}$$

if-then-else statement Let B be an if-then-else statement of the following form:

```
if (E) S1 else S2
```
(212)

The effect of an if-then-else statement is asserted only if no flags have been set. In addition, *guard* must be applied to both the then-part and the else-part of the if-then-else statement. Therefore,

$$\sigma[\![guard(\mathsf{B})]\!] \mathrel{\hat{=}} [\sigma[\![\mathsf{B}]\!]; \beta, \beta', \kappa, \kappa' : \tau[\![\mathtt{int}]\!]] \qquad \text{(rule 143)}$$

$$\epsilon[\![guard(\mathsf{B})]\!] \mathrel{\hat{=}} [\Delta\sigma[\![guard(\mathsf{B})]\!] \mid \beta = \beta' \wedge \kappa = \kappa' \wedge \beta = 0 \wedge \kappa = 0$$
$$\wedge\ \theta\sigma[\![\mathsf{B}]\!]' = \theta\sigma[\![\mathsf{B}]\!]] \mathbin{;} ([\epsilon[\![guard(\mathsf{S}_1)]\!] \mid \nu[\![\mathsf{E}]\!] \neq 0]$$
$$\vee\ [\epsilon[\![guard(\mathsf{S}_2)]\!] \mid \nu[\![\mathsf{E}]\!] = 0])$$
$$\vee\ [\Xi\sigma[\![guard(\mathsf{B})]\!] \mid \beta = 1 \vee \kappa = 1] \qquad \text{(rule 144)}$$

iterative statement For an iterative statement B, we only need to make sure that it does not execute when one or both flags have been set. We do not need to apply *guard* to the loop body because the `break` and `continue` statements within the loop body do not apply to the enclosing iterative statement.

$$\sigma[\![guard(\mathsf{B})]\!] \mathrel{\hat{=}} [\sigma[\![\mathsf{B}]\!]; \beta, \beta', \kappa, \kappa' : \tau[\![\mathtt{int}]\!]] \qquad \text{(rule 145)}$$

$$\epsilon[\![guard(\mathsf{B})]\!] \mathrel{\widehat{=}} [\Delta\sigma[\![guard(\mathsf{B})]\!] \mid \beta = 0 \wedge \kappa = 0 \wedge \epsilon[\![\mathsf{S}]\!] \wedge \theta\sigma[\![\mathsf{B}]\!]' = \theta\sigma[\![\mathsf{B}]\!]] \; ; \; \epsilon[\![\mathsf{S}]\!]$$
$$\vee \; [\Xi\sigma[\![guard(\mathsf{B})]\!] \mid \beta = 1 \vee \kappa = 1] \qquad \text{(rule 146)}$$

6 Deriving a Specification from Code

This section describes how the five abstractions described in Section 5 may be used for to derive a specification from code.

6.1 Significant Points of Execution

Every abstract operation defined on an abstract model in a model-based specification must respect the abstract state invariant. This means that the abstract state invariant asserted on the abstract model must be true before and after each operation. However, as these operations are broken down into programming language constructs during implementation, it is unnecessary to assert that each *atomic operation* (a non-divisible operation as defined by a particular programming language) in the program respects the abstract state invariant. This is because an abstract operation in the specification might have been implemented by several atomic operations in the program, none of them required to satisfy the abstract state invariant individually. In fact, the only times when a particular variable is required to satisfy the abstract state invariant are at *points* immediately preceding an atomic operation or a sequence of atomic operations that exhibits an externally observable behavior that depend on the value of that variable being 'consistent' (according to the application domain). For our purposes, we define a *significant point of execution* to be a point in the program at which some set of objects satisfy

the abstract state invariant. This means that the code that implements an abstract operation in the specification must reside *between* significant points of execution.

Generally speaking, a significant point of execution may be characterized by two attributes: a point in the program, and an associated set of tuples of, each has the form (*predicate*, *object*). For each tuple, *predicate*, a predicate in terms of the values of some set of the objects whose parents are *active* (according to scope rules) at that point, is satisfied when control reaches that point during execution, the value of *object*, which designate an object whose parent is also active at that point, must satisfy the abstract state invariant. We shall make the following assumptions for simplification:

- A significant point of execution may only occur either between two statements, right before the first statement in a block, or right after the last statement in a block.
- For any tuple (*predicate*, *object*) associated with any significant point of execution, *object* must be a variable, and *predicate* must be *true*.

The first assumption implies that any abstract operation must have been implemented either as a statement or as a sequence of statements. The second assumption implies that, at any significant point of execution, a fixed set of *variables* satisfy the abstract state invariant.

6.2 Representing Abstract Models and Operations

representing abstract model An abstract model of a program may be represented by a single state schema, or organized as a set of state schemas (each

represents a *sub-model*) non-recursively related by schema inclusion. If the program implements multiple independent sets of abstract operations, there may be separate schemas or sets of schemas for the models corresponding to each sets of operations. Here is an example of an abstract model Σ organized hierarchically, consisting of two sub-models

$$\Sigma_1 \mathrel{\widehat=} [a, b : \tau[\mathbf{int}] \mid inv_1(a, b)] \tag{213}$$

$$\Sigma_2 \mathrel{\widehat=} [c : \tau[\mathbf{int}] \mid inv_2(c)] \tag{214}$$

$$\Sigma \mathrel{\widehat=} [\Sigma_1; \Sigma_2 \mid inv_3(a, b, c)] \tag{215}$$

The predicate $inv_1(a, b)$, $inv_2(c)$ and $inv(a, b, c)$ each represents a *part* of the abstract state invariant (denoted by *Inv*). The abstract state invariant of an abstract model is the conjunction of them, that is,

$$Inv = inv_1(a, b) \wedge inv_2(c) \wedge inv_3(a, b, c) \tag{216}$$

Since each schema is interpreted as a part or the whole of the same model, the abstract state invariant is applicable to all the sub-models as well as the abstract model. Therefore, any operation defined on *any* of the sub-model would be subject to the same state invariant *Inv*. In general, the 'effective' state invariant for a sub-model Σ_i is given by

$$inv_i = (\exists \overline{y}_i \bullet Inv) \tag{217}$$

where $\overline{y}_i$ are the components of the abstract model that are not declared in sub-model Σ_i. This state invariant must be used later when we attempt to prove that a set of abstract operations, defined on the abstract model, are indeed implemented by the program. On the other hand, the state invariants in the sub-models may be useful for the purpose of reuse, in which those abstract operations that are

candidates for reuse may be considered quite separately from other operations implemented by the program.

representing abstract operations Consider an abstract model of a program that consists of n sub-models Σ_1, Σ_2, ..., and Σ_n. Let Ω_i be an abstract operation, which operates on some sub-models of the abstract model, say Σ_{i_1}, Σ_{i_2}, ..., Σ_{i_m}, where $1 \leq i_j \leq n$ for each $1 \leq j \leq m$. Let the construct $\mathsf{S_i}$ (which must either be a statement or a sequence of statements) in the program be the implementation of Ω_i. The abstract operation may be specified with the following operation schema:

$$\Omega_i \mathrel{\widehat=} [\Delta\Sigma_{i_1}; \Delta\Sigma_{i_2}; \cdots; \Delta\Sigma_{i_n} \mid \epsilon[\![\mathsf{S_i}]\!] \setminus (\overline{a}_i, \overline{a}'_i)] \tag{218}$$

where $\overline{a}_i$ and $\overline{a}_i$ represents the initial and final values of the *auxiliary variables* that occur in $\mathsf{S_i}$. We shall define *auxiliary variables* of a construct C to be the subset of the variables of C that occur only for programming convenience, and do not correspond to any variables in the abstract model. Auxiliary variables are not included in the abstract model. We shall define the opposite of auxiliary variables to be *essential variables*[11], which correspond to some variables in the abstract model. The relationship between significant points of execution and essential variables is that, at both significant points of execution that encloses S, the essential variables of S must satisfy the state invariant. We emphasize at this point that the classification of program variables as either essential or auxiliary is arbitrary in the sense that the classification may not be inferred from the program alone. The hiding of auxiliary variables in (218) implies the following assumption on the relationship between S and the rest of the program:

- Auxiliary variables, which are useful only for intermediate computation in an implementation, have not been included in the abstract model ($\overline{a}$ and $\overline{a}'$

[11] or *main variables* plus associated flags and counters, *etc.*[15]

are supposed not to be components of Ω_i).

- Initial values of auxiliary variables in executing an implementation of an abstract operation should be immaterial to the result of the operation.
- Final values of auxiliary variables in performing an abstract operation must have no consequence on the behavior of the program for the rest of its execution.

These assumptions imply that auxiliary variable may not carry any persistent data (with respect to the abstract operations), that may be shared among abstract operations.

These assumptions carry with them certain limitations on the kinds of abstract operations that may be formalized. For example, it is impossible to use (218) to formalize a set of fine-grained (in the sense that they perform subtasks of other abstract operations) abstract operations that need to communicate through auxiliary variables. In effect, we choose to omit those operations that perform part of an abstract operations because they concern with how the abstract operations are implemented. Such omission is essential in extracting the essence of a program from its implementation details.

6.3 A Procedure for Deriving a Specification from Code

We have examined the formalism that we may use to represent a specification derived from a program. Now, we need a procedure that enables us to use that formalism to derive specifications from particular programs. Given a program, the procedure calls for the following tasks which may be performed in the following order:

- classify the variables defined throughout the program as either essential or auxiliary.
- form the preliminary abstract model, consisting of one sub-model for each essential variable, all included by a single sub-model.
- identify significant points of execution, where essential variables are deemed to satisfy the abstract state invariant.
- identify abstract operations and specify them on the preliminary abstract model.
- organize the abstract model, and edit the signatures of the abstract operations so that they operate on parts of the organized abstract model.
- assert the abstract state invariant.
- write the specification, include documentation of the procedure performed.

6.4 Proof Obligations for Derivation of Specification from a Program

In deriving from a program a specification in the form described above, there are a number of pieces of information in the specification about the program that is absent in the program itself:

- The distinction between essential and auxiliary variables. They differ in two important ways:
  - Essential variables must be properly initialized (refer to 'Abstract Initial State' to see how this may be captured).

  - Each essential variable must satisfy the abstract state invariant at respective significant points of execution.

- By asserting certain constructs in the program as implementing abstract operations, process boundaries[20] are artificially imposed on to the program, due to the assumptions made about the relationship between the construct that implement the abstract operation and the rest of the program.

- When a non-trivial abstract state invariant (that is, a predicate other than *true*) is asserted on the abstract model, further restriction is imposed on the validity of the program relative to the specification. Specifically, termination of the program may no longer guarantee its logical validity (with respect to its specification) because the program may fail to establish the required invariants at significant points of execution.

On the other hand, the following information, that is available from the program, are absent in the specification:

- the context in which the an abstract operation is actually invoked (only the precondition remains).

- the order of invocation of the abstract operations.

Proper comparison between a program and a specification derived from it requires us to establish a refinement relation between them. The general criteria for establishing refinement relation is that the program may not exhibit any behavior contradictory to its specification. For our purposes, we may use a similar conditions as stated by Wordsworth in [42, p.169]:

safety condition:

$$\mathbf{pre}\,\epsilon[\![\mathsf{P}[\Omega_1/\epsilon[\![\mathsf{S}_1]\!]][\Omega_2/\epsilon[\![\mathsf{S}_2]\!]]\cdots[\Omega_n/\epsilon[\![\mathsf{S}_n]\!]]]\!] \Rightarrow \mathbf{pre}\,\epsilon[\![\mathsf{P}]\!] \tag{219}$$

liveness condition:

$$\begin{aligned}&\mathbf{pre}\,\epsilon[\![\mathsf{P}[\Omega_1/\epsilon[\![\mathsf{S}_1]\!]][\Omega_2/\epsilon[\![\mathsf{S}_2]\!]]\cdots[\Omega_n/\epsilon[\![\mathsf{S}_n]\!]]]\!] \wedge \epsilon[\![\mathsf{P}]\!]\\ &\Rightarrow \epsilon[\![\mathsf{P}[\Omega_1/\epsilon[\![\mathsf{S}_1]\!]][\Omega_2/\epsilon[\![\mathsf{S}_2]\!]]\cdots[\Omega_n/\epsilon[\![\mathsf{S}_n]\!]]]\!]\end{aligned} \tag{220}$$

The notation $[\Omega_i/\epsilon[\![\mathsf{S}_i]\!]]$ means syntactic substitution of Ω_i for $\epsilon[\![\mathsf{S}_i]\!]$. Ω_1, Ω_2, Ω_n are abstract operations identified in program P that are not enclosed by any other abstract operations. Abstract operations that are enclosed by other abstract operations would appear in the declaration of their enclosing abstract operations. It is imperative that (219) and (220) may be used only when, for any pairs of abstract operations, the constructs that implement them are not interleaving.

6.4.1 Sufficient Proof Obligations

The proof obligations presented in (219) and (220) involves evaluating the abstract effect of the entire program. In certain cases we may use a stronger proof obligation so that we may restrict our effort to smaller constructs.

rendering an implicit state invariant explicit When we classify all variables of a program to be essential, and only assert the abstract state invariant, we may omit application of conditions (219) and (220) if the following is true for *every* abstract operation Ω_i:

$$\epsilon[\![\mathsf{S}_i]\!] \wedge inv_i \Rightarrow inv_i' \tag{221}$$

where S_i is the code that implements Ω_i, inv_i is state invariant applicable to Ω_i according to equation (217). (221) asserts that whenever precondition of operation is satisfied, the asserted state invariant is automatically satisfied. In effect, an implicit state invariant is rendered explicit.

reinitialization of auxiliary variables If the code for every abstract operation initializes all the auxiliary variables that it uses, the second and third assumptions made about the relationship between the code that implements an abstract operation and the program (p.135) would always be true. Therefore, in such cases, (221) is applicable even if not all variables are classified as essential.

6.5 Abstract Initial State

We may identify the first significant point of execution, with respect to control flow, that *all* [12] essential variables simultaneously satisfy the abstract state invariant as the *point of initialization*. For a program that has the point of initialization, we may develop an abstract initial state based on the effect of execution between the beginning (or *entry point*) of the program (including initializations of static variables) and the point of initialization. Unfortunately, the execution path between the entry point and the point of initialization, in general, is not a construct according to our classification. For example, the point of initialization may reside within an if-then-else statement, as shown in the following program: (This example also illustrates a case in which the 'point' of initialization is distributed between two points):

```
void main () {
    Dlocal    /* local declarations */
    S1    /* some initializations */
    if (E) {
        S2    /* additional initializations depending
                    on the input */
            /* ← initialization done at this point */
```

[12] a general program does not need to have any significant points of execution where all essential variables satisfy the abstract state invariant

```
        S3    /* rest of the processing */
    } else {
        S4    /* additional initializations depending
                    on the input */
            /* ← initialization done at this point */
        S5    /* rest of the processing */
    }
    S6    /* rest of the processing */
}
```

(222)

To handle this, we may derive an abstraction rule particularly for the problem at hand:

$$\Sigma \mathrel{\widehat{=}} [\sigma[\![\mathsf{E}]\!] \wedge \sigma[\![\mathsf{S}_2]\!] \wedge \sigma[\![\mathsf{S}_4]\!])' \mid inv(\overline{v})]$$

$$\Omega_{init} \mathrel{\widehat{=}} [\Sigma' \mid \epsilon[\![\mathsf{S}_1]\!] \fatsemi [(\sigma[\![\mathsf{E}]\!] \wedge \sigma[\![\mathsf{S}_2]\!] \wedge \sigma[\![\mathsf{S}_4]\!])' \mid \nu[\![\mathsf{E}]\!] \neq 0 \wedge \epsilon[\![\mathsf{E}]\!] \fatsemi \epsilon[\![\mathsf{S}_2]\!]$$
$$\vee\ \nu[\![\mathsf{E}]\!] = 0 \wedge \epsilon[\![\mathsf{E}]\!] \fatsemi \epsilon[\![\mathsf{S}_4]\!]] \setminus (\overline{a}')] \qquad \text{(rule 147)}$$

In (rule 6.5) $\overline{v}$ represents the essential variables that are of E, S_2, or S_4, and $\overline{a}$ represents the auxiliary variables. Note that we do not need to hide $\overline{a}$, initial values of the auxiliary variables, because they are not declared in the above schema at all. Similarly, in the first disjunct in the predicate part, we do not need to specify that those variables of S_4 that are not of E or S_2 must remain constants, nor do we need to specify in the second disjunct that those variables of S_2 that are not of E or S_4 must remain constants.

7 Obtaining Higher Level Abstractions

The specification obtained from using the steps outlined in Section 6.3 is restricted in the sense that the representation of the model is dictated by data structures employed by the program. To make the specification more understandable, sometimes a different representation is desirable. An approach to obtain a different representation for the model is *data refinement*[33, 8, 42]. *Data refinement* specifies the relationship between two specifications which may have different representations for their models. In particular, a set of conditions may be used to prove whether one specification is a *correct* implementation (or *refinement*) of another[33, p.136]. In this section, we shall compare three such different sets of conditions to identify their significance in deriving higher level abstractions in reverse engineering.

7.1 Operation Refinement

Operation refinement[33, p.135-6] is a limited form of data refinement in which the models of the specification and its refinement must be the same.

7.1.1 Per-Operation Criteria

For a *concrete operation Cop* to be a refinement of an *abstract operation Aop*, two conditions must be satisfied:

safety condition:

$$\textbf{pre}\ Aop \Rightarrow \textbf{pre}\ Cop \tag{223}$$

liveness condition:

$$\textbf{pre}\ Aop \wedge Cop \Rightarrow Aop \tag{224}$$

One way to prove that one specification is a refinement of another is to prove that every abstract operation in the abstract specification and its corresponding con-

crete operation in the concrete specification satisfy (223) and (224). This criteria is useful in development, in which a specification is divided into components, implemented by several teams. Every team must ensure that it produces a correct implementation of its portion of the specification.

7.1.2 Complete Programs Criteria

A weaker refinement criteria is to establish the refinement conditions between every pairs of *programs*[8, p.241], rather than pairs of operations, derived from the two specifications. For our purposes, a *complete program* consists of sequential compositions of any sequence of operations, appended to the abstract initial state (by sequential composition). Let *CInit* and *AInit* be the abstract initial states of a concrete and an abstract specifications respectively, For *every* complete program $\mathcal{P}_c$, derived from a concrete specification that includes n operations:

$$\mathcal{P}_c \cong CInit \fatsemi Cop_{i_1} \fatsemi Cop_{i_2} \fatsemi \cdots \fatsemi Cop_{i_n} \tag{225}$$

where i_1, i_2, ..., i_n are indices between 1 and n, to be a refinement of the corresponding complete program $\mathcal{P}_a$, derived from an abstract specification:

$$\mathcal{P}_a \cong AInit \fatsemi Aop_{i_1} \fatsemi Aop_{i_2} \fatsemi \cdots \fatsemi Aop_{i_n} \tag{226}$$

the following conditions must be satisfied:

safety condition:

$$\mathbf{pre}\,\mathcal{P}_a \Rightarrow \mathbf{pre}\,\mathcal{P}_c \tag{227}$$

liveness condition:

$$\mathbf{pre}\,\mathcal{P}_a \wedge \mathcal{P}_c \Rightarrow \mathcal{P}_a \tag{228}$$

7.1.3 Specification Substitution Criteria

Yet another criteria for operation refinement is to prove that the operations in two specifications are extracted from the same program. For any program P, in order for statements S_1, S_2, ..., S_n to be implementations of both abstract operations Aop_1, Aop_2, ..., Aop_n, and more concrete operations Cop_1, Cop_2, ..., Cop_n, the refinement conditions would be

safety condition:

$$\begin{aligned} &\mathbf{pre}\,\epsilon[\![\mathsf{P}[Aop_1/\epsilon[\![\mathsf{S}_1]\!]][Aop_2/\epsilon[\![\mathsf{S}_2]\!]]\cdots[Aop_n/\epsilon[\![\mathsf{S}_n]\!]]]\!] \\ &\Rightarrow \mathbf{pre}\,\epsilon[\![\mathsf{P}[Cop_1/\epsilon[\![\mathsf{S}_1]\!]][Cop_2/\epsilon[\![\mathsf{S}_2]\!]]\cdots[Cop_n/\epsilon[\![\mathsf{S}_n]\!]]]\!] \end{aligned} \tag{229}$$

liveness condition:

$$\begin{aligned} &\mathbf{pre}\,\epsilon[\![\mathsf{P}[Aop_1/\epsilon[\![\mathsf{S}_1]\!]][Aop_2/\epsilon[\![\mathsf{S}_2]\!]]\cdots[Aop_n/\epsilon[\![\mathsf{S}_n]\!]]]\!] \\ &\wedge\ \epsilon[\![\mathsf{P}[Cop_1/\epsilon[\![\mathsf{S}_1]\!]][Cop_2/\epsilon[\![\mathsf{S}_2]\!]]\cdots[Cop_n/\epsilon[\![\mathsf{S}_n]\!]]]\!] \\ &\Rightarrow \epsilon[\![\mathsf{P}[Aop_1/\epsilon[\![\mathsf{S}_1]\!]][Aop_2/\epsilon[\![\mathsf{S}_2]\!]]\cdots[Aop_n/\epsilon[\![\mathsf{S}_n]\!]]]\!] \end{aligned} \tag{230}$$

Here is an example to illustrates that complete programs criteria is weaker than per-operation criteria: Let specification $\mathcal{C}$ includes two operations. Operation *CAdd* specifies addition of an item to a list if the item has not already exist in the list. Operation *CDel* specifies deletion of the first occurrence of an item from the list. No state invariant is asserted on the list. At the abstract initial state the list is empty. Let specification $\mathcal{A}$ includes two operations. Operation *AAdd* is exactly the same as *CAdd*. Operation *ADel* specifies deletion of all occurrences of an item from the list. Using the per-operation criteria, since *CDel* is not an acceptable implementation of *ADel*, specification $\mathcal{C}$ would not be a refinement of $\mathcal{A}$. However, according to complete programs criteria, $\mathcal{C}$ is a refinement of $\mathcal{A}$.

The specification substitution criteria is a yet weaker criteria than the others. This is because this criteria may be seen as restricting the conditions required for complete programs criteria only to certain sequences of operations (rather than to *any* sequences of operations).

All the three criteria described above may be used in reverse to obtain higher level abstractions in reverse engineering. Per-operation criteria provides a easy to prove but strong proof obligation. Complete programs criteria is useful for verifying the use of an abstract data type in a program. Specification substitution criteria provides a difficult to prove but weak proof obligation in justifying higher level understanding of a program.

7.2 Data Refinement

Unlike operation refinement, data refinement may occur between specifications that have different models [33, p.137]. Hence, in data refinement, we need to establish a relationship between the models of the two specifications. Such relationship may be represented by a *forward simulation*[42, p.163] (or *abstraction schema*[33, p.137]) Let *FSim* be the forward simulation between AS, the model of the abstract specification, and CS, the model of the concrete specification. Then *FSim* is a schema of the following form:

$$FSim \mathrel{\widehat=} [AS; CS \mid \mathbf{P}] \tag{231}$$

The signature includes both the concrete and abstract models. The predicate part of the schema, $\mathbf{P}$, specifies the relationship between the concrete and abstract models.

If both specifications are organized hierarchically in the form described in Section 6.2, and every submodel in one specification has a corresponding submodel

in the other, the forward simulation may be defined on each pair of corresponding submodels.

7.2.1 Per-Operation Criteria

For concrete operation Cop_i, defined on submodel k in the concrete specification, to be a refinement of abstract operation Aop_i, defined on corresponding submodel k in the abstract specification, the refinement conditions would be[33, p.138]

safety condition:

$$\mathbf{pre}\ Aop_i \wedge FSim_k \Rightarrow \mathbf{pre}\ Cop_i \tag{232}$$

liveness condition:

$$\mathbf{pre}\ Aop_i \wedge FSim_k \wedge Cop_i \Rightarrow Aop_i \tag{233}$$

where $FSim_k$ is the forward simulation between submodel k in the abstract and the concrete specification. Let *CInit* and *AInit* be the initial states of concrete and abstract specifications. An additional refinement condition is that the abstract initial states between the specifications must also be consistent[33, p.138], that is,

$$CInit[\_/\_'] \Rightarrow (\exists\, AS \bullet AInit[\_/\_'] \wedge FSim) \tag{234}$$

7.2.2 Complete Programs Criteria

Let $\mathcal{P}_a$ and $\mathcal{P}_c$ be corresponding programs from the abstract and the concrete specifications respectively. The conditions for refinement would be

safety condition:

$$\mathbf{pre}\,\mathcal{P}_a \wedge FSim \Rightarrow \mathbf{pre}\,\mathcal{P}_c \tag{235}$$

liveness condition:

$$\mathbf{pre}\,\mathcal{P}_a \wedge FSim \wedge \mathcal{P}_c \Rightarrow \mathcal{P}_a \tag{236}$$

where $FSim$ is the forward simulation between the abstract and concrete models. We do not need to prove the condition on consistency between initial states because this is implied by the safety condition.

7.2.3 Specification Substitution Criteria

To derive the specification substitution criteria for data refinement, we may define the forward simulation between the two specification on the program, instead of between the specifications. Let the notation $FSim_{a_i}$ denotes a schema of the following form:

$$FSim_{a_i} \mathrel{\widehat=} [\sigma[\![\mathsf{S_i}]\!]; AS_i \mid \cdots] \tag{237}$$

which is the forward simulation between the abstract state of $\mathsf{S_i}$, and the submodel AS_i on which the abstract operation Aop_i is defined. Similarly, let the notation $FSim_{c_i}$ denotes a schema of the following form:

$$FSim_{c_i} \mathrel{\widehat=} [\sigma[\![\mathsf{S_i}]\!]; CS_i \mid \cdots] \tag{238}$$

which is the forward simulation between the abstract state of $\mathsf{S_i}$, and the submodel CS_i on which the more concrete operation Cop_i is defined.

For any program P, in order for statements $\mathsf{S}_1, \mathsf{S}_2, \ldots, \mathsf{S}_n$ to be implementations of both abstract operations $Aop_1, Aop_2, \ldots, Aop_n$, and more concrete operations $Cop_1, Cop_2, \ldots, Cop_n$, the refinement conditions would be

safety condition:

$$\mathbf{pre}\,\epsilon[\![\mathsf{P}[FSim_{a_1} \fatsemi Aop_1 \fatsemi FSim_{a_1}[\_'/\_]/\epsilon[\![\mathsf{S}_1]\!]][FSim_{a_2} \fatsemi Aop_2 \fatsemi FSim_{a_2}[\_'/\_]/\epsilon[\![\mathsf{S}_2]\!]]$$

$$\cdots [FSim_{a_n} \,;\, Aop_n \,;\, FSim_{a_n}[\_'/\_]/\epsilon[\![\mathsf{S_n}]\!]]]$$
$$\Rightarrow \mathbf{pre}\,\epsilon[\![\mathsf{P}[FSim_{c_1} \,;\, Cop_1 \,;\, FSim_{c_1}[\_'/\_]/\epsilon[\![\mathsf{S_1}]\!][FSim_{c_2} \,;\, Cop_2 \,;\, FSim_{c_2}[\_'/\_]/\epsilon[\![\mathsf{S_2}]\!]]$$
$$\cdots [FSim_{c_n} \,;\, Cop_n \,;\, FSim_{c_n}[\_'/\_]/\epsilon[\![\mathsf{S_n}]\!]]] \quad (239)$$

liveness condition:

$$\mathbf{pre}\,\epsilon[\![\mathsf{P}[FSim_{a_1} \,;\, Aop_1 \,;\, FSim_{a_1}[\_'/\_]/\epsilon[\![\mathsf{S_1}]\!][FSim_{a_2} \,;\, Aop_2 \,;\, FSim_{a_2}[\_'/\_]/\epsilon[\![\mathsf{S_2}]\!]]$$
$$\cdots [FSim_{a_n} \,;\, Aop_n \,;\, FSim_{a_n}[\_'/\_]/\epsilon[\![\mathsf{S_n}]\!]]]$$
$$\wedge\, \epsilon[\![\mathsf{P}[FSim_{c_1} \,;\, Cop_1 \,;\, FSim_{c_1}[\_'/\_]/\epsilon[\![\mathsf{S_1}]\!][FSim_{c_2} \,;\, Cop_2 \,;\, FSim_{c_2}[\_'/\_]/\epsilon[\![\mathsf{S_2}]\!]]$$
$$\cdots [FSim_{c_n} \,;\, Cop_n \,;\, FSim_{c_n}[\_'/\_]/\epsilon[\![\mathsf{S_n}]\!]]]$$
$$\Rightarrow \epsilon[\![\mathsf{P}[FSim_{a_1} \,;\, Aop_1 \,;\, FSim_{a_1}[\_'/\_]/\epsilon[\![\mathsf{S_1}]\!][FSim_{a_2} \,;\, Aop_2 \,;\, FSim_{a_2}[\_'/\_]/\epsilon[\![\mathsf{S_2}]\!]]$$
$$\cdots [FSim_{a_n} \,;\, Aop_n \,;\, FSim_{a_n}[\_'/\_]/\epsilon[\![\mathsf{S_n}]\!]]] \quad (240)$$

8 Conclusions and Future Work

The objective of this thesis is to present a formal approach for reverse engineering. The thesis (i) provides a detailed description of the abstractions used in the reverse engineering process, (ii) describes in detail the method to obtain these abstractions from a program written in a C subset, (iii) enumerates the method to aggregate the abstractions into a formal functional specification of the software system being implemented by the C subset, and finally (iv) provides adequate justifications for the correctness of the extracted specification (In particular, we have compared the (logical) strength of three sets of such conditions and their application in reverse engineering). Subsequent paragraphs in this section discuss our experience and findings during the reverse engineering process.

However, this thesis represents only a preliminary work in this area. Although this thesis provides a method to derive a specification from code, a lot more are needed to be done on how one develops a specification that is useful for understanding or reimplementation. Nevertheless, the thesis advocates the use of formal methods in reverse engineering and may serve as a motivation or even basis for research in the this direction.

The abstraction rules for assignment expressions appear to be much more complicated than those found in the literature. For example, in [31, p.119], the precondition for an assignment statement $x := E$ is $Q[E/x]$, where Q is the postcondition. The precondition refers to the syntactic substitution of E into all free occurrences of x in Q. This technique is not applicable to our approach because the *object* being assigned to in an expression is assumed to be implicit in the sense that it may not occur as an identifier in the assignment expression. In fact, this is always the case when anything other than a variable name occurs on the left side of the assignment. This problem is a natural consequence of attempting to deal with composite variables without reducing them into simple variables. For example, REDO's approach to reverse engineering using formal method[3], due to one of the normalization rules, requires components of a composite variable to be separated into simple variables[3, p.204].

Recall (rule 76) for step two of a simple assignment expression:

$$\mathcal{A}'[\![\mathsf{E}_1{=}\mathsf{E}_2]\!] \mathrel{\widehat{=}} [\Delta\sigma[\![\mathsf{E}_1{=}\mathsf{E}_2]\!]; \sigma[\![\mathsf{E}_1{=}\mathsf{E}_2]\!]_o \mid ((\omega'[\![\mathsf{E}_1]\!] \fatsemi \epsilon[\![\mathsf{E}_2]\!]) :'= \nu'[\![\mathsf{E}_2]\!])_{[\_/\_'][\_'/\_o]} \wedge \overline{x}' = \overline{x}] \setminus (\__o) \qquad (241)$$

Although the object being assigned is made implicit in this rule, its parent is not, due to the predicate $\overline{x}' = \overline{x}$, where $\overline{x}$ represents those variables of the assignment expression other than the parent of the object being assigned. Therefore, this rule may still represent difficulties if the assumption (page 58) that the parent of the

object designated by an object designation expression is not fixed is lifted (the lift is required for arbitrary pointer arithmetics to be covered).

A major extension required for this thesis is to handle pointer variables, and arbitrary pointer arithmetics. Instead of using a huge array to represent the memory, as proposed in [23, p.313], we may use a definite description. The form of the abstract object of an expression E would then be the following:

$$(\mu\omega_1 : \tau[\![\mathsf{E}]\!] \mid P(\omega_1, \overline{v}, \overline{w}_o)) \tag{242}$$

in which more than one variable would have subscript 'o' (denoted by $\overline{w}_o$). It is not obvious at this stage how the syntactic transformation ':'=' (page 98) and ':=' (page 102) may be modified to handle the situation.

Among the three sets of refinement conditions discussed in Section 7, non-trivial examples of usage only exist for per-operation criteria (there is an example for a form of complete program criteria in [8, p.242-4] where the two specifications differ only in one operation)[42, Ch.6]. It seems that verifying complete program criteria and specification substitution criteria are both very difficult.

Case studies are needed, both for validating the methodology, assessing difficulty in proving refinement conditions, and accumulating heuristics in classifying variables between being essential and auxiliary.

The abstractions rules may be extended to handle integer types other than `int` quite easily. A new kind of abstraction, say $\rho[\![\mathsf{T}_1, \mathsf{T}_2]\!]$, may define a relation between $\tau[\![\mathsf{T}_1]\!]$ and $\tau[\![\mathsf{T}_2]\!]$, that specifies the type conversion between types T_1 and T_2. However, pointer arithmetics will likely to complicate the rules for such simple constructs as addition and subtraction.

Finally, tool support is essential, as with any formal approach. Possible directions on tool development includes theorem proving (for example, [29]), and

browsing of specification and program text, preferably in an integrated environment. A tool has been developed for representing a vast subset of ANSI C as SGML[38, p.186]. As pointed out in [38, p.189], Z may also be represented in a similar way.

References

[1] R.S. Arnold. *Software Reengineering.* IEEE Computer Society Press, 1993.

[2] K. Bennett, M.M. Cornelius, and D. Robson. Software Maintenance. In J.A. McDermid, editor, *Software Engineer's Reference Book*, chapter 20. Butterworth-Heinemann Ltd., 1991.

[3] P.T. Breuer; K. Lano; J. Bowen. Understanding Programs through Formal Methods. In *REDO Compendium*, pages 195–223. John Wiley & Sons Ltd., U.K., 1993.

[4] M.L. Brodie. The Promise of Distributed Computing and the Challenges of Legacy Information Systems. In *Advances in Object-Oriented Database Systems.* Springer-Verlag, 1994.

[5] T. Cahill. UNIFORM. In *REDO Compendium*, pages 123–9. John Wiley & Sons Ltd., U.K., 1993.

[6] Gerald C. Gannod; Betty H.C. Cheung. A Two-Phase Approach to Reverse Engineering Using Formal Methods. In *International Conference on Formal Methods in Programming and Their Applications, (Novosibivsk, Russia; Jun 28 - Jul 3, 1993)*, pages 336–48, 1993.

[7] Guide Int'l Corp. Application Reengineering. Guide Pub GPP-208, Guide Int'l Corp., Chicago, 1989.

[8] J. Woodcock; J. Davis. *Using Z.* Prentice Hall Europe, 1996.

[9] J. Beck; D. Eichmann. Program and Interface Slicing for Reverse Engineering. In *Proceedings of Working Conference on Reverse Engineering, (Baltimore, MD, U.S.A.; May 21 – 23, 1993)*, pages 54–63, 1993.

[10] J. Mylopoulos et al. Towards an Integrated Toolset for Program Understanding. In *Proceedings of the 1994 IBM CAS Conference (CASCON '94), (Toronto, Ontario; October 31 - November 3, 1994)*, pages 19–31. International Business Machines, 1994.

[11] J. MacRae G. Whittington, C.T. Spracklen. Applications of Artificial Neural Networks to Reverse Software Engineering. In L.I. Burke C.H. Dagli and Y. C. Shin, editors, *Proceedings of Intelligent Engineering Systems through Artificial Neural Networks, (St. Louis, Missouri, USA; Nov 10 - 12, 1991).* ASME Press, 1991.

[12] Gerald Catolico Gannod. The Application of Formal Methods to the Reverse Engineering of Imperative Program Code. Master's thesis, Computer Science Department, Michigan State University, 1994.

[13] Carl A. Gunter. *Semantics of Programming Languages: Structures and Techniques.* Foundations of Computing Series. MIT Press, Cambridge, Massachusetts London, England, 1992.

[14] Samuel P. Harbison. *C: A Reference Manual.* Prentice-Hall, Inc., 4th edition, 1995.

[15] K. Lano; H. Haughton. *Reverse Engineering and Software Maintenance: a Practical Approach.* McGraw-Hill International (UK) Ltd., 1994.

[16] K.C. Lano; P.T. Breuer; H. Haughton. Reverse Engineering COBOL via Formal Methods. In *REDO Compendium*, pages 225–47. John Wiley & Sons Ltd., U.K., 1993.

[17] Philip A. Hausler; Mark G. Pleszkoch; Richard C. Linger; Alan R. Hevner. Using Function Abstraction to Understand Program Behavior. *IEEE Software*, pages 55–63, Jan 1990.

[18] IEEE. *IEEE Guide to Software Requirements Specifications*, IEEE Std 830-1984 edition, 1984.

[19] J.Q. Ning; A. Engberts; W. Kozaczynski. Recovering Reusable Components from Legacy Systems by Program Segmentation. In *Proceedings of Working Conference on Reverse Engineering, (Baltimore, MD, U.S.A.; May 21 - 23, 1993)*, pages 64–72, 1993.

[20] P.T. Breuer; K. Lano. Creating Specification from Code: Reverse-Engineering Techniques. *Journal of Software Maintenance: Research and Practice*, 3(3):145–62, Sep 1991.

[21] Bertrand Meyer. *Introduction to the Theory of Programming Languages.* Prentice Hall International (U.K.), 1990.

[22] Mili. *Introduction to Program Verification.* Van Nostrand Reinhold Company Inc., 1985.

[23] M. Ward; F.W. Calllss; M. Munro. The Maintainer's Assistant. In *Proceedings of Conference on 1989 Software Maintenance, (Miami, FL, U.S.A.; Oct 16 - 19, 1989)*, pages 307–15, New York, 1989. IEEE Computer Society Press.

[24] Stephen Brien; John Nicholls. *Z Base Standard.* Oxford University Computing Laboratory, Programming Research Group, version 1.0 edition, Nov 1992.

[25] D. Craigen; S. L. Gerhart; T. J. Ralston. An International Survey of Industrial Applications of Formal Methods. Technical Report NIST GCR 93/626-V1 & 2,

Atomic Energy Control Board of Canada, US National Institute of Standards and Technology, and US Naval Research Laboratories, 1993.

[26] Susan Horwitz; Thomas Reps. The Use of Program Dependence Graphs in Software Engineering. In *Proceedings of 1992 International Conference of Software Engineering, (Melbourne, Vic., Australia; May 11 - 15, 1992)*, pages 392–411, 1992.

[27] Peter Baumann; Jürg Fässler; Markus Kiser; Zafer Öztürk; Lutz Richter. Semantics-Based Reverse Engineering. Technical Report ifi-94.08, Institut für Informatik der Universität Zürich, Winterthurerstr, Zürich, Switz, 1994.

[28] Kit Kamper; Spencer Rugaber. A Reverse Engineering Methodology for Data Processing Applications. Technical Report GIT-SERC-90/02, College of Computing, Georgia Institute of Technology, Mar 1990.

[29] Mark Saaltink. *The Z/EVES System.* ORA Canada, Sep 1995.

[30] Steve Westlund; Jim Gehringer; Dennis Sandstedt. Software Maintenance. Technical Report Vol. 5, No. 2 (WP 88-37), Washington University, St. Louis, Missouri, 1991.

[31] Robert W. Sebesta. *Concepts of Programming Languages.* The Benjamin/Cummings Publishing Company, Inc., 2nd edition, 1993.

[32] C.M. Overstreet; R. Cherinka; R. Sparks. Using bidirection data flow analysis to support software reuse. Technical Report TR-97-28, Computer Science Department, Old Dominion University, Norfolk, VA, Jan 1994.

[33] J.M. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall, 2nd edition, 1992.

[34] Tim Tiemens. Cognitive models of program comprehension. Dec 1989.

[35] H.A. Müller; M.A. Orgun; S.R. Tilley; J.S. Uhl. A Reverse Engineering Approach to Subsystem Structure Identification. *Software Maintenance: Research and Practice*, 5(4):181–204, Dec 1993.

[36] H.J. van Zuylen. *REDO Compendium.* John Wiley & Sons Ltd., U.K., 1993.

[37] H.J. van Zuylen. Understanding in reverse engineering. In *REDO Compendium*, pages 81–92. John Wiley & Sons Ltd., U.K., 1993.

[38] D.D. Cowan; D.M. Germán; C.J.P. Lucena; A. von Staa. Enhancing code for readability and comprehension using SGML. In *Proceedings of 1994 IEEE International Conference on Software Maintenance, (Vancouver, B.C., Canada; May 11 - 15, 1992)*, pages 181–90, 1992.

[39] Martin Ward. *Proving Program Refinements and Transformations.* PhD thesis, St. Annes College, Oxford, June 1989.

[40] Ted J. Biggerstaff; Bharat G. Mitbander; Dallas E. Webster. Program understanding and the concept assignment problem. *Communications of the ACM*, 37(5):72–83, May 1994.

[41] Glynn Winskel. *The formal semantics of programming languages: an introduction.* Foundations of computing. Massachusetts Institute of Technology Press, 1993.

[42] J.B. Wordsworth. *Software Development with Z: A Practical Approach to Formal Methods in Software Engineering.* Int'l computer science. Addison-Wesley, 1992.