November 16, 2000

SRC Research Report 160

Data abstraction and information hiding

K. Rustan M. Leino
and
Greg Nelson

COMPAQ

Systems Research Center
130 Lytton Avenue
Palo Alto, California 94301

http://research.compaq.com/SRC/

Compaq Systems Research Center

SRC's charter is to advance the state of the art in computer systems by doing basic and applied research in support of our company's business objectives. Our interests and projects span scalable systems (including hardware, networking, distributed systems, and programming-language technology), the Internet (including the Web, e-commerce, and information retrieval), and human/computer interaction (including user-interface technology, computer-based appliances, and mobile computing). SRC was established in 1984 by Digital Equipment Corporation.

We test the value of our ideas by building hardware and software prototypes and assessing their utility in realistic settings. Interesting systems are too complex to be evaluated solely in the abstract; practical use enables us to investigate their properties in depth. This experience is useful in the short term in refining our designs and invaluable in the long term in advancing our knowledge. Most of the major advances in information systems have come through this approach, including personal computing, distributed systems, and the Internet.

We also perform complementary work of a more mathematical character. Some of that lies in established fields of theoretical computer science, such as the analysis of algorithms, computer-aided geometric design, security and cryptography, and formal specification and verification. Other work explores new ground motivated by problems that arise in our systems research.

We are strongly committed to communicating our results; exposing and testing our ideas in the research and development communities leads to improved understanding. Our research report series supplements publication in professional journals and conferences, while our technical note series allows timely dissemination of recent research findings. We seek users for our prototype systems among those with whom we have common interests, and we encourage collaboration with university researchers.

Data abstraction and information hiding

K. Rustan M. Leino and Greg Nelson

November 16, 2000

Publication History

An earlier draft of this note, to which there exist some citations in the literature, had the title "Abstraction and specification revisited" and internal document number KRML 71.

Abstract

This paper describes an approach for verifying programs in the presence of data abstraction and information hiding, which are key features of modern programming languages with objects and modules. The paper focuses on the property of modular soundness, that is, the property that the separate verifications of the individual modules of the program suffice to ensure the correctness of the composite program. The paper introduces a new specification language construct, the *abstraction dependency*, and argues that it is needed to achieve modular soundness in the presence of data abstraction and information hiding. This paper discusses in detail two varieties of abstraction dependencies: static and dynamic. The paper also presents a new technical definition of modular soundness as a monotonicity property of verifiability with respect to scope and uses this technical definition to formally prove the modular soundness of a programming discipline for static dependencies.

Contents

The romance of the precise is not the elision
Of the tired romance of imprecision.
It is the ever-never-changing same,
An appearance of Again, the diva-dame.
— Wallace Stevens

0 Introduction

This paper describes an approach for verifying programs in the presence of data abstraction, object types, and information hiding. The genesis of this work was the Extended Static Checking project (ESC) [8], which applies program verification technology to systems programs written in Modula-3. The aim of ESC is not to prove full functional correctness, but to prove the absence of common errors, such as array index errors, **nil** dereference errors, race conditions, deadlocks, etc.

One of the biggest problems we encountered in the ESC project is that the verification methodology we know from the literature does not seem to apply to the systems programs in the Modula-3 libraries. The problem is not that the programs use low-level tricks or unsafe code; the problem is that the programs use patterns of modularization and data abstraction that are richer than those treated in the verification literature. This is not an artifact of Modula-3, but would apply to any modern object-oriented language.

The data abstraction technology we know from the literature extends and refines the seminal paper on data abstraction by C.A.R. Hoare in 1972 [18]. In particular, Hoare and all subsequent treatments that we know impose the requirement that all of the concrete variables used to represent an abstraction must be declared in the same module. This requirement is too strict: if it were applied to the Modula-3 libraries, many small modules would have to be combined, with a loss of desirable information-hiding. Writing specifications is supposed to improve the structure of a program, so it is ironic that standard treatments of data abstraction are incompatible with good modularization. Therefore, in this paper we weaken Hoare's requirement and allow the concrete variables used to represent an abstraction to be divided among several modules.

A key technical challenge is to check modules where an abstract variable is visible, some of the concrete variables used to represent it are visible, but the abstraction representation function is not visible. To meet this challenge, we introduce a new specification construct called the *abstraction dependency*. This construct specifies that an abstraction connection exists between the variables, but does not specify the actual abstraction representation, which can be confined to a more private scope. There are different types of dependencies, and these types produce a useful taxonomy of the patterns of abstraction in modular software.

Abstraction dependencies give the programmer considerable freedom in arranging the declarations of abstract variables, concrete variables, abstraction representation functions, and dependencies among the modules of a program. Too much freedom: without further restrictions, we would lose the property of modular soundness, that is, the property that the separate verifications of the individual modules of the program suffice to ensure the correctness of the composite program. We therefore impose several requirements, called *modularity requirements*, and argue that modular verification is sound for programs that meet the modularity requirements.

1 On the need for data abstraction

Before we get into the details of our generalization, we set the stage by reviewing the rôle of data abstraction in modular verification.

To check that a large program does what it is supposed to do, we must study it piece by piece. Nobody's short-term memory is big enough to hold all the details of a large program. If the checking effort (formal or informal) is to be manageable, we cannot afford to re-examine the body of a procedure for every one of its calls. This is the reason for writing specifications, formal or informal. Given specifications, we check that each procedure meets its specification, assuming that the procedures it calls meet theirs.

This checking process is called modular verification, and for simple programming languages it has been understood since C.A.R. Hoare's work on axiomatic semantics in the 1960s. (As long as the bulk of the verification is done modularly, we do not exclude simple link-time checks, such as the check that each procedure is implemented somewhere in the program.) The central goal of this paper is to understand modular verification in the presence of two modern programming features: data abstraction and information hiding.

A procedure specification includes a precondition and a postcondition. The precondition is the part of the contract to be fulfilled by the caller of the procedure, and the postcondition is the part of the contract to be fulfilled by the procedure implementation. But precondition and postcondition are not enough: the specification also includes a "modifies list" that limits which variables the procedure is allowed to modify. Without the modifies list, the contract would allow a procedure to have arbitrary side effects on any variable not constrained by the postcondition, which would make the contract useless to the client.

It is possible to view the modifies list as syntactic sugar for extra conjuncts in the postcondition, asserting that every variable not mentioned in the modifies list is unchanged. That is, in a program with three variables x, y, and z, the specification

requires P **modifies** x **ensures** Q

could be "desugared" into

requires P **ensures** $Q \ \wedge \ y = y' \ \wedge \ z = z'$

in which primed variables denote post-values and unprimed variables denote pre-values. We cannot, however, use this desugaring to pretend that each procedure specification consists of a precondition and postcondition only. The reason is that, in modular verification, we never know, when verifying a procedure, what the set of all variables in the final program will be. Perhaps x, y, and z are the only variables visible where the procedure is declared, but more variables may be visible where the procedure is called. Therefore, in this paper we take the view that the modifies list is an integral part of the specification. Although we will rewrite modifies lists, the rewriting is different for different scopes.

Unfortunately, and perhaps surprisingly to those who have used verification more in principle than in practice, the methodology described so far is still inadequate. In many cases, it would be preposterous to try to list every piece of state that might be modified by a call to a procedure. For example, what would be the list for the `putchar` procedure from the C standard I/O library? What `putchar` does is simply write a character to output, but anybody who has implemented an I/O system will be aware that the list of what can be modified during the execution of a call to `putchar` is very long. It includes, for example, the I/O buffers, the internal state of the device drivers for the disk and network, the device registers in these drivers, and the disk and network themselves. The minor problem is that this list is long; the major problem is that the variables in the list are not visible at the point of declaration of `putchar`, and to make them visible would be to give up on information hiding, which would be to resign the game before it starts.

The solution to this difficulty —at least the only solution that we can imagine— is data abstraction. Abstractly, `putchar` modifies a single abstract variable, of a simple type (say, sequence of byte). All the internal state, from buffers to devices, must be treated as concrete state that is part of the representation of the abstract state.

Some people see data abstraction as an algorithm design methodology only, as a methodology for deriving an efficient algorithm from a simple algorithm by changing the representation of the state. We have no quarrel with their use of data abstraction, but our point is that data abstraction is also an essential ingredient in any scheme for modular verification of large systems, since it seems to be the only hope for writing a useful modifies list for a procedure whose implementation changes the system state at many levels of abstraction.

Having identified the general idea of the solution to the `putchar` problem as data abstraction, we would add that the patterns of data abstraction that arise in verifying `putchar` are beyond the current state of the art of specification: we believe that no semantics or methodology presented in the literature is equal to the task. We hope this paper will be a useful step in this neglected area.

2 Validity as an abstract variable

The generalized data abstraction described in this paper is relevant regardless of whether verification is being used for full functional correctness or for more limited aims, such as the ESC aim of verifying the absence of certain classes of errors only. The examples in this paper will be ESC verifications. These verifications tend to have a typical form, which is described in this section.

In a typical ESC verification, we associate two abstract variables with each type, $valid$ and $state$. The first of these records whether objects of the type satisfy the internal representation invariant required by the implementation, and the second represents the abstract value of variables of the type.

If we were verifying full functional correctness, we would have to write many specifications about the $state$ variable. But in doing extended static checking, we rarely say anything about the state. We aren't proving that the program meets its full functional specification, only that it doesn't crash. The main purpose of the $state$ variable is to account for the side effects in the implementations of the methods, which otherwise would lead to spurious errors reported by the verifier. Indeed, in many ESC verifications, we don't even bother to provide the concrete representation for $state$.

In contrast to $state$, the checking performed by ESC depends critically on $valid$. Most operations on an object o will have $valid[o]$ as a precondition. The checker uses the concrete representation for $valid$ to translate $valid[o]$ into a concrete precondition, which it then uses in proving that the implementation of the operation does not cause an error.

In Hoare's original paper on data abstraction, the notion of a validity invariant was built into the methodology. Initialization was required to establish validity and all other operations were required to preserve it. In contrast, we consider *valid* to be an abstract variable like any other; the programmer explicitly provides *valid* as a precondition (and/or postcondition), and the implementation infers the details of validity in terms of the concrete state via the usual process of data abstraction. Our approach has several advantages over Hoare's, of which we mention one: we allow operations like closing a file, which destroy validity. Such operations are frequently essential in order to deallocate resources.

3 Definition of notation

This section introduces the notation and terminology that we use in the rest of the paper.

Modularity. A *program* is a collection of *declarations*. Declarations introduce names for entities (such as types, abstract and concrete variables, and methods) and/or specify properties of named entities (such as subtype relationships, representations of abstract variables, method specifications, and method implementations). The declarations of a program are partitioned into *units* (sometimes called interfaces and modules). The declarations in effect in a unit are its own declarations and the declarations in effect in units that it *imports*. If an entity E is declared in a unit M, it is known as $M.E$ in importers of M and known simply as E within M. For example:

unit M	**unit** N **import** M
type T	... uses of $M.T$ ...
... uses of T ...	

In this paper, we sometimes write E instead of $M.E$ when M is clear from the context.

A set of units D is called a *scope* if it is closed under imports, that is, if whenever a unit M in D imports a unit N, then N is also in D. A declaration is *visible* in a scope if it appears in one of the units in the scope.

We use units and imports in this paper since they are simple and extremely general. Restrictive patterns are common in practice. For example, Modula-3 requires that every unit be an interface unit, which can declare procedures and methods but not declare implementations, or an implementation unit, which can declare implementations but which cannot be imported. As another example, CLU imposes a correspondence between units and type declarations. We have not imposed such restrictions in this paper, because they seem orthogonal to the modularity issues that we are discussing.

Types. In this paper, we will use primitive types like **int** and **bool**, as well as object types and array types. Our objects are like those of Simula and Modula-3: they are implicitly references, and each object type has a uniquely determined direct supertype. More precisely, an *object* is either **nil** or a reference to a set of data fields and methods; a method is a procedure that will accept the object as its first parameter. Equality of objects is reference equality. An *object type* determines the names and types of a prefix of the fields and the names and signatures of a prefix of the methods of its objects.

An object type T is declared

$$\textbf{type}\ T <: S$$

where S is an object type declared elsewhere. This introduces the name T for a new object type whose direct supertype is S, meaning that T contains all the fields and methods of S and possibly includes other fields and methods declared elsewhere. The "$<: S$" is optional; if omitted, S defaults to an anonymous object type serving as the root of the subtype hierarchy.

Every object has a *dynamic type* determined when it is allocated. Every expression has a *static type* determined at compile time. If v is the dynamic value of an expression E, v has dynamic type D, and E has static type S, then conventional static type-checking rules assure that D is a subtype of S.

We consider a data field, abstract or concrete, to be a map from objects to values. Thus, where others write

$$\textbf{class}\ T = \{\ \ldots f\colon \textbf{int}\ \ldots\ \}$$

we write

$$\begin{array}{l}\textbf{type}\ T\\ \textbf{var}\ f\colon T \to \textbf{int}\end{array}$$

Also, we write $f[t]$ where others write $t.f$ to denote the value of the f field of object t. We refer to T and **int** as the *index type* and *range type* of f, respectively.

The **class** notation forces f to be co-declared with T, whereas our notation allows them to be declared independently. This generality is not problematical; in fact, it simplifies the semantics.

If T is a type, we write

$\mathbf{array}[T]$

to denote the type of (references to) arrays with element type T. If a is of type $\mathbf{array}[T]$ and is non-**nil**, then $\mathbf{number}(a)$ denotes the number of elements in a, and $a[i]$ denotes element i of a for $0 \le i < \mathbf{number}(a)$. To properly model the fact that arrays are references, we introduce the predeclared map variable $elems$: the expression $elems[a]$ denotes the sequence of elements referred to by an array a. For example, $a = b$ means that a and b reference the same sequence, while $elems[a] = elems[b]$ means that the sequences referenced have the same elements. In fact, $a[i]$ is shorthand for $elems[a][i]$.

If T is an object type, $\mathbf{new}(T)$ allocates and returns a new object of dynamic type T. For any type T, $\mathbf{new}(T, n)$ allocates and returns a new array of dynamic type $\mathbf{array}[T]$ and of length n.

A method m for type T is declared by

proc $m(t\colon T, \ldots\ args\ \ldots)\colon R$
  requires P
  modifies w
  ensures Q

where in the *signature* "$(t\colon T, \ldots\ args\ \ldots)\colon R$", T is an object type, t is the self parameter, $args$ lists the names and types of any additional parameters, and R is the result type. In this paper, all parameters are in-parameters. In addition to declaring the name and signature of the method, the declaration associates with it as a specification the precondition P, postcondition Q, and modifies list w. A program can contain at most one declaration for a given method for a given type; for example, we don't allow strengthening a method specification in a subtype (this is a simplification that does not actually limit expressiveness, see p. 348 of [31]). In the postcondition, **result** denotes the result value, primed variables denote values in the post-state, and unprimed variables denote values in the pre-state. If the precondition or postcondition is omitted, it defaults to $true$; if the modifies list is omitted, it defaults to the empty list.

A method m for type T can be implemented differently for each subtype of T. A method implementation of m for some subtype U of T is declared by

impl $m(u\colon U, \ldots\ args\ \ldots)\colon R$ **is** S **end**

where S is an executable statement, and the implementation signature

$$(u\colon U, \ldots\ args\ \ldots)\colon R$$

coincides with the declared signature except (possibly) for the type of the first parameter. Statement S must satisfy (that is, the verifier checks that it satisfies) the specification associated with the m method for T. The ideas in this paper don't depend on the particular executable statements allowed. The examples in this paper use Algol-like executable statements, whose meaning we hope will be clear to the reader.

A method is called by

$$t.m(\ldots\ args\ \ldots)$$

where t is an object (the actual self parameter), m is a method name, and $args$ is a list of any additional actual parameters. The static type of t is used in determining the declaration and specification of m. The declaration is used to type-check the actual parameters and determine the static type of the result, the specification is used to reason about the semantics of the call. The dynamic type of t is used at run-time to determine which implementation of m to invoke. Since all method implementations are proved to meet their specifications, and since the dynamic type of t is a subtype of the static type of t, it is sound to reason about the semantics of the dynamic dispatch in this way.

Abstraction. A data field can be declared to be *abstract* by preceding its declaration with **spec**. For example:

$$\textbf{spec var}\ valid\colon T \to \textbf{bool}$$

An abstract field occupies no memory at run-time; it is a fictitious field whose value (or *representation*) is defined as a function of other fields. The representation is declared by a syntax like

$$\textbf{rep}\ valid[t\colon T] \equiv f[t] \neq 0 \tag{0}$$

which means that for any object t of type T, the abstract value of $valid[t]$ is $true$ if and only if $f[t] \neq 0$.

The representation of an abstract variable can be different for different subtypes. As an example, consider the object type Rat representing rational numbers, and two of its subtypes, $Ratio$, which represents each rational as a ratio,

$$
\begin{array}{l}
\textbf{type } \mathit{Rat} \\
\textbf{spec var } \mathit{valid} \colon \mathit{Rat} \to \textbf{bool} \\
\textbf{type } \mathit{Ratio} <: \mathit{Rat} \\
\textbf{var } \mathit{num}, \mathit{den} \colon \mathit{Ratio} \to \textbf{int} \\
\textbf{rep } \mathit{valid}[r \colon \mathit{Ratio}] \equiv \mathit{den}[r] > 0 \\
\textbf{type } \mathit{CFrac} <: \mathit{Rat} \\
\textbf{var } \mathit{parquo} \colon \mathit{CFrac} \to \textbf{array}[\textbf{int}] \\
\textbf{rep } \mathit{valid}[\mathit{cf} \colon \mathit{CFrac}] \equiv \\
\quad \mathit{parquo}[\mathit{cf}] \neq \textbf{nil} \wedge \\
\quad \langle \forall i :: 1 \leq i < \textbf{number}(\mathit{parquo}[\mathit{cf}]) \Rightarrow \mathit{parquo}[\mathit{cf}][i] > 0 \rangle
\end{array}
$$

Figure 0: An example program, illustrating that representation of an abstract variable can be subtype-specific.

and CFrac, which represents each rational as a continued fraction (which is a representation of a rational as a sequence of integers), see Figure 0. These declarations specify that the concrete representation of $\mathit{valid}[q]$ varies depending on the dynamic type of q: for rationals represented as ratios, validity means that the denominator is positive, whereas for continued fractions, validity means that each partial quotient is positive, except possibly the first.

A **rep** declaration given for a type T applies to all non-**nil** objects of type T, including those whose dynamic type is a subtype of T. One might think that it would be possible to override a **rep** declaration for T with another **rep** declaration for some subtype of T, but this is not allowed. This rule is enforced at link-time.

The variables appearing in the right-hand side of the **rep** declaration for an abstract variable are called *dependencies* of the abstract variable. The dependencies can themselves be either concrete or abstract. Our notion of dependencies is not to be confused with use-def dependencies [2].

A major novelty of our approach is to require that dependencies be declared explicitly. For example, the representation (0) would cause an "undeclared dependency" error unless $f[t]$ were declared as a dependency of $\mathit{valid}[t]$, which is done by a declaration of the form

$$\textbf{depends } \mathit{valid}[t \colon T] \textbf{ on } f[t]$$

In this paper, we sometimes omit the " $: T$ " when T is obvious or unimportant. The **depends** declaration can be subtype-specific, just like the **rep** declaration. For example, the representations in Figure 0 might be accompanied by

$$\textbf{depends}\ valid[r\colon Ratio]\ \textbf{on}\ den[r]$$
$$\textbf{depends}\ valid[cf\colon CFrac]\ \textbf{on}\ parquo[cf],\ elems[parquo[cf]]$$

The validity of the continued fraction cf depends both on the array $parquo[cf]$ and on the contents of the array. These are different dependencies and both must be declared, as shown above. The validity of the ratio r depends only on $den[r]$.

This paper is principally concerned with two forms of dependencies, static and dynamic. A *static* dependency has the form

$$\textbf{depends}\ a[t\colon T]\ \textbf{on}\ c[t] \tag{1}$$

A *dynamic* dependency has the form

$$\textbf{depends}\ a[t\colon T]\ \textbf{on}\ c[b[t]] \tag{2}$$

In each case, a is an abstract variable and c is either an abstract or a concrete variable. In the case of the dynamic dependency, b is concrete. A dependency on the contents of an array counts as a dynamic dependency, with *elems* playing the rôle of c. Other forms of dependencies will be discussed in Section 9.1, but static and dynamic dependencies are more common and fundamental.

A major goal of this paper is to design a discipline for the placement of dependency declarations in a multi-module program. The paper is long, but the main conclusion is short: the static dependency (1) must be visible wherever c is, and the dynamic dependency (2) must be visible wherever b is.

Dependencies affect the verification process in several ways. One way is *modifies list desugaring*. For example, in a scope where

$$\textbf{depends}\ a[t]\ \textbf{on}\ c[t]$$

is visible, the modifies list

$$\textbf{modifies}\ a[t]$$

is desugared into something like

$$\textbf{modifies}\ a[t],\ c[t]$$

This reflects the common-sense view that the license to modify an abstract variable implies the license to modify its representation. The precise details of modifies list desugaring will be described later in the paper.

4 Example: Readers

From our experience with ESC, we have found that dependencies are not just a detail but a key ingredient of the specification language that we used constantly. However, since dependencies are a tool for programming in the large, no small example does them full justice. This section presents the smallest example we know that motivates the essential points: a simplified version of *readers*, which are the object-oriented buffered input streams used in the standard I/O library of Modula-3. A key point that the example will illustrate is that modern information hiding together with subtyping creates situations where both an abstract variable and one or more of its dependencies are visible, but the associated representation is not visible. In these situations, sound modular verification would be impossible, but dependencies save the day.

Readers (and their output counterparts, *writers*) were invented by Stoy and Strachey for the OS6 operating system [47]. Although Stoy and Strachey never used the word "object" or "class" in describing them, they are in fact one of the most compelling examples of the engineering utility of object-oriented programming. Each reader is an object with a buffer and a method for refilling the buffer. Different subtypes of readers override the refill method with code appropriate to that type of reader; for example, a disk reader fills the buffer from the disk, a network reader from the network.

As part of the ESC project, we have mechanically verified the absence of errors from most of the Modula-3 standard I/O library, including all the standard reader subtypes. In this paper we want to focus on generalized data abstraction, and many of the complexities of the actual I/O system would distract us from this focus, so we will simplify the reader interface rather drastically. (The actual code and specifications that we have used as input to the Extended Static Checker can be found on the Web [13].)

Our simplified interface *Rd* declares the type *T* representing a reader, and specifies the two methods *getChar* and *close*, see Figure 1. Since our examples show ESC verifications only, we specify the range type of *state* as **any**, and we ignore the effects on *state* in the **ensures** clauses. We use the convention that *rd.getChar*() returns -1 when *rd* is exhausted, and otherwise returns the next byte of input. The specification of *close* reflects the design decision that a reader can be closed only once (a second call to *close* requires validity, which may have been destroyed by the first call).

Next we describe the unit that defines the generic buffer structure (by generic, we mean common to all readers, as opposed to subtype-specific), see Figure 2.

unit *Rd*
  type *T*
  spec var $valid: T \to$ **bool**
  spec var $state: T \to$ **any**
  proc $getChar(rd: T)$: **int**
    requires $valid[rd]$
    modifies $state[rd]$
    ensures $-1 \leq$ **result** < 256
  proc $close(rd: T)$
    requires $valid[rd]$
    modifies $valid[rd], state[rd]$

Figure 1: The interface Rd, which declares type T representing readers.

The integer $cur[rd]$ is the index in the abstract stream rd of the next byte to be returned by $getChar$. The integers $lo[rd]$ and $hi[rd]$ delimit the range of bytes in the abstract stream that are contained in the buffer $buff[rd]$ (see Figure 3).

Interface *RdRep* declares and specifies the *refill* method, but leaves its implementation to various subtypes. The convention used by *refill* is that the call $rd.refill()$ must make at least one new byte available (that is, it must establish $cur[rd] < hi[rd]$), unless rd is exhausted, in which case it must establish the condition $cur[rd] = hi[rd]$.

The postconditions of *getChar* and *refill* don't reflect the conventions for signaling that the reader is exhausted (nor does the variable *state* describe the condition that the reader is exhausted), because our example is an ESC verification, not a verification of full functional correctness.

The **rep** declaration reveals the representation of the abstract variable *valid* in terms of the concrete variables *lo*, *cur*, *hi*, and *buff*. In addition, because subtypes may have their own validity invariants, the interface declares the abstract variable *svalid*, and adds the conjunct $svalid[rd]$ to the representation of $valid[rd]$. The intended meaning of $svalid[rd]$ is that rd satisfies the validity invariant of its dynamic type. Each subtype of $Rd.T$ will include a **rep** declaration specifying the representation of *svalid* for readers of that subtype. For example, a reader for a disk file would include a file handle as one of its fields, and its *svalid* would include the validity of the file handle.

unit *RdRep* **import** *Rd*
  var $lo, cur, hi\colon Rd.T \to$ **int**
  var $buff\colon Rd.T \to$ **array**[**byte**]
  spec var $svalid\colon Rd.T \to$ **bool**
  rep $valid[rd\colon Rd.T] \equiv$
    $rd \neq$ **nil** $\wedge$
    $0 \leq lo[rd] \leq cur[rd] \leq hi[rd] \wedge$
    $buff[rd] \neq$ **nil** $\wedge$ $hi[rd] - lo[rd] \leq$ **number**$(buff[rd]) \wedge$
    $svalid[rd]$
  proc $refill(rd\colon Rd.T)$
    requires $valid[rd]$
    modifies $state[rd]$
    ensures $cur[rd] = cur'[rd]$
  depends $valid[rd\colon Rd.T]$ **on** $lo[rd], cur[rd], hi[rd], buff[rd], svalid[rd]$
  depends $state[rd\colon Rd.T]$ **on** $lo[rd], cur[rd], hi[rd], buff[rd],$
    $elems[buff[rd]]$
  depends $svalid[rd\colon Rd.T]$ **on** $lo[rd], hi[rd], buff[rd]$

Figure 2: The interface *RdRep*, which defines the buffer structure common to all objects of type *Rd.T*.

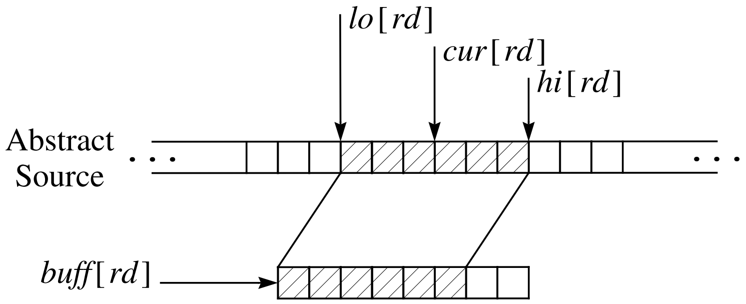


Figure 3: Buffer representation of readers.

unit *RdImpl* **import** *Rd*, *RdRep*
  impl *getChar*(*rd*: *Rd.T*): **int is**
    if $cur[rd] = hi[rd]$ **then** *rd.refill*() **end** ;
    if $cur[rd] = hi[rd]$ **then**
      result $:= -1$
    else
      result $:= buff[rd][cur[rd] - lo[rd]]$;
      $cur[rd] := cur[rd] + 1$
    end
  end

Figure 4: The implementation unit *RdImpl*, which contains the implementation of the method *getChar*.

The **depends** declaration for *valid* is explained by our requirement that dependencies be explicit—without it, the checker would complain that the **rep** declaration for *valid* contains undeclared dependencies. The **depends** declarations for *state* and *svalid* are more subtle and will be explained later.

Next we present the generic implementation in Figure 4.

To give the flavor of an ESC verification, consider checking that $cur[rd] - lo[rd]$ is a valid index into $buff[rd]$ in the implementation of *getChar*. Since $valid[rd]$ is a precondition of *getChar*, and is preserved by *rd.refill*(), we conclude that $valid[rd]$ holds at the first semicolon. Thus, the validity of the index boils down to showing that

$$0 \le cur[rd] - lo[rd] \ \wedge\ cur[rd] - lo[rd] < \mathbf{number}(buff[rd]) \tag{3}$$

follows from

$$valid[rd] \ \wedge\ cur[rd] \neq hi[rd] \tag{4}$$

Since *RdImpl* imports *RdRep*, the representation of $valid[rd]$ is visible. Since this representation contains the conjunct $lo[rd] \le cur[rd]$, the first conjunct of (3) follows immediately. The proof of the second conjunct is:

$$cur[rd] - lo[rd]$$
$$< \quad \{ \ cur[rd] \le hi[rd] \ \wedge\ cur[rd] \neq hi[rd] \ \text{(from (4))} \ \}$$

unit *BlankRd* **import** *Rd*
  type $T <: Rd.T$
  proc $init(brd\colon T,\ n\colon \mathbf{int})\colon T$
    requires $0 \leq n$
    modifies $valid[brd]$, $state[brd]$
    ensures $valid'[brd]\ \wedge\ \mathbf{result} = brd$

Figure 5: Unit *BlankRd* declares a subtype $BlankRd.T$ of $Rd.T$, whose readers deliver streams of blanks.

$$\begin{array}{ll} & hi[rd] - lo[rd] \\ \leq & \qquad \{ \quad valid[rd] \quad \} \\ & \mathbf{number}(buff[rd]) \end{array}$$

Returning to general comments about $rd.getChar()$, notice that the implementation modifies $cur[rd]$, but the **modifies** clause in the specification of the method *getChar* does not mention $cur[rd]$. Why does the checker not complain? Because of modifies list desugaring, as mentioned in the previous section. Modifies list desugaring gives *getChar* the license to modify $cur[rd]$, because *getChar* is specified to modify $state[rd]$, which is declared in *RdRep* to depend on $cur[rd]$. This explains why $cur[rd]$ was declared a dependency of $state[rd]$.

The object-oriented I/O stream design by Stoy and Strachey illustrates the flexibility of subtyping: having carefully designed the central abstraction $Rd.T$, it can serve as the blueprint for dozens of useful subtypes. To illustrate the modularity issues that arise when a subtype is defined, we now give the interface (Figure 5) and implementation (6) of a trivial type of reader, a *blank reader*, which delivers a sequence of blanks whose length is determined at initialization time. More precisely, the expression $\mathbf{new}(BlankRd.T).init(n)$ allocates, initializes, and returns a reader that delivers a stream of exactly n blanks. The conjunction "$\mathbf{result} = brd$" in the postcondition specifies that the *init* method return the object that it initializes, a convention we have found useful. The method *init* stores the argument n in the field $num[brd]$ for later use by the method *refill*. The method also initializes the *lo*, *cur*, and *hi* fields in the obvious way, allocates a buffer of size up to 8192 (that is, up to 8 kilobytes), and fills the buffer with blanks (code 32).

As we shall see later, it is critical to the verification of the module that each blank reader *brd* satisfy the invariant $hi[brd] \leq num[brd]$. Therefore, the unit *BlankRdImpl* provides a subtype-specific representation for *svalid*, effectively

```
unit BlankRdImpl import Rd, RdRep, BlankRd
  var num: BlankRd.T → int
  rep svalid[brd: BlankRd.T] ≡ hi[brd] ≤ num[brd]
  impl init(brd: BlankRd.T, n: int): BlankRd.T is
      num[brd] := n ;
      buff[brd] := new(byte, min(8192, n)) ;
      lo[brd] := 0 ; cur[brd] := 0 ;
      hi[brd] := number(buff[brd]) ;
      for i := 0 to hi[brd] − 1 do
        buff[brd][i] := 32
      end ;
      result := brd
    end
  impl refill(brd: BlankRd.T) is
      lo[brd] := cur[brd] ;
      hi[brd] := min(lo[brd] + number(buff[brd]), num[brd])
    end
  depends state[brd: BlankRd.T] on num[brd]
  depends svalid[brd: BlankRd.T] on num[brd]
```

Figure 6: The blank reader implementation unit *BlankRdImpl*.

strengthening the general reader validity invariant as needed for the particular subtype $BlankRd.T$.

Notice that $brd.refill()$ must be proved to maintain $valid[brd]$, because its modifies list does not allow it to modify $valid[brd]$. Among the proof obligations associated with maintaining $valid[brd]$ is that at exit,

$$cur'[brd] \leq hi'[brd]$$

Since the body of $refill$ does not change $cur[brd]$ and makes $hi'[brd]$ equal to

$$\mathbf{min}(cur[brd] + \mathbf{number}(buff[brd]), num[brd])$$

proving this postcondition boils down to showing that each argument to **min** is at least $cur[brd]$. For the first argument, this follows from the fact that the **number** of any array is non-negative. The proof for the second argument is:

$$\begin{array}{ll}
 & valid[brd] \\
\Rightarrow & \quad \{ \ \mathbf{rep} \text{ for } valid \ \} \\
 & cur[brd] \leq hi[brd] \ \wedge \ svalid[brd] \\
= & \quad \{ \ brd \text{ is of type } BlankRd.T\text{, } \mathbf{rep} \text{ for } svalid \text{ for this type} \ \} \\
 & cur[brd] \leq hi[brd] \ \wedge \ hi[brd] \leq num[brd] \\
\Rightarrow & \quad \{ \ \text{transitivity} \ \} \\
 & cur[brd] \leq num[brd]
\end{array}$$

We present this calculation in detail to illustrate that the verification of the $refill$ method of even the trivial $BlankRd.T$ requires the subtype-specific validity conjunct. (The need for the $svalid$ conjunct is more conspicuous in more interesting reader subtypes.)

Read-only by specification. The calculation that $cur[brd] \leq num[brd]$ follows from $valid[brd]$ would be in vain if the generic code could modify $hi[brd]$. If, for example, the generic implementation of $rd.getChar()$ would sometimes increment $hi[rd]$, then it could destroy $svalid[rd]$, which could cause all kinds of errors. To prevent this, the Modula-3 interface from which we translated $RdRep$ contains the following English comment:

The generic code modifies $cur[rd]$, but not $lo[rd]$, $hi[rd]$, or $buff[rd]$. (5)

This guarantee is essential to subtypes, since between calls to their $refill$ methods, they may need to know that lo, hi, and $buff$ have not been changed by the generic code.

How do we translate the sentence (5) into a formal specification? Modula-3 does not have any kind of **readonly** qualifier for field declarations. Java and C++ have qualifiers like `private` and `protected`, but these declarations don't help with the current problem. Both of them allow the implementation to read and write the fields, while limiting the access from subtypes (and from other clients). What we need here is almost the opposite: a declaration that will forbid the generic implementation from writing the fields, while allowing subtypes to write them. We can hardly expect a programming language to have a declaration qualifier that enforces this highly particular access policy, but modular verification will not be sound unless the access policy is formally stated and enforced.

We wrestled with the problem for some time before realizing happily that **depends** provides a neat solution. In fact, the third dependency declaration in *RdRep*, which states that $svalid[rd]$ depends on $lo[rd]$, $hi[rd]$, and $buff[rd]$, is the desired formalization of (5). For if the generic code were to modify any of these fields, the presence of the dependency would alert the checker to the possibility that *svalid*, and therefore *valid*, might be changed. In other words, in a scope where $lo[rd]$, $hi[rd]$, and $buff[rd]$ are known to be part of the representation of $svalid[rd]$, but the explicit representation is unknown, the only hope for maintaining $svalid[rd]$ invariant is to avoid modifying $lo[rd]$, $hi[rd]$, and $buff[rd]$. We call this technique "read-only by specification".

Summary. To repeat our main conclusion from this example, we see that modular verification with subtyping creates situations where both an abstract variable and one or more of its dependencies are visible, but where the associated representation is not visible. We have seen two instances of this:

- The dependencies of *state* are specified in *RdRep*, but no representation for *state* is specified. The dependencies must be visible so that the implementations of operations that modify the state (for example, *getChar*) will have the license to modify the concrete variables that represent the state. The representation declaration cannot be visible, for two reasons. First, because we are doing extended static checking only, we never give a representation for the state. Second, even if we were doing full-scale verification, the representation would be subtype-specific, but the dependencies must be visible in the generic scope.

- The dependencies of *svalid* are specified in *RdRep*, but no representation for *svalid* is given there. The dependencies are necessary to prevent generic operations from modifying the variables that are reserved for the use of subtypes. But it is clearly impossible to present a representation declaration for *svalid* in the *RdRep* scope, since the whole point of *svalid* is to allow subtypes to include their own invariants as part of validity: these invariants can't be known in the generic scope.

Explicit dependencies may seem verbose, and it would be nice to be able to infer them automatically. But this will not always be possible. For example, of the three **depends** declarations in *RdRep*, we can imagine inferring the first (from the **rep** declaration for *valid*), but the second and third could not be automatically inferred since they are used to specify non-trivial design decisions (namely, which fields can be modified by generic code, and which can be modified by subtypes only).

This concludes our example of the rôle of dependencies in modular verification. In the remainder of the paper, we investigate different kinds of dependencies and the way they affect the verification process.

5 Static dependencies

In this section, we describe more fully how dependencies affect the verification process. Our guiding principles are:

- *Abstract function principle.* An abstract variable is a function of the concrete variables on which it depends.

- *Abstraction modification principle.* The license to modify an abstract variable implies the license to modify its concrete representation, but the license to modify a concrete variable does not imply the license to modify an abstract variable that depends on it.

Our technique is to rewrite preconditions, postconditions, modifies lists, and **rep** declarations into equivalent forms that contain concrete variables only. In this section, we will describe the rewriting steps and explain how they follow from the principles.

We confine ourselves to static dependencies for simplicity; in Section 7, we will extend this material to dynamic dependencies.

5.0 Functionalization

Guided by the abstraction function principle, and following in the footsteps of Hoare, we introduce a new function symbol for each abstract variable. In this paper, we write $\mathcal{F}.a$ to denote the function symbol introduced for the abstract variable a. The idea is that $\mathcal{F}.a$ gives a's value as a function of the concrete state. Occurrences of a in preconditions and postconditions are replaced by function applications of the form $\mathcal{F}.a(\ldots)$. For example, $a[t] > 6$ becomes $\mathcal{F}.a(\ldots)[t] > 6$. The arguments to $\mathcal{F}.a$ are the variables on which a depends (together with other arguments that will be introduced later). The process of substituting $\mathcal{F}.a$ for a is called *functionalization*.

The **rep** declaration for a is rewritten into an appropriate axiom about $\mathcal{F}.a$ (a *rep axiom*). If a is visible but its representation is not, then $\mathcal{F}.a$ occurs in the rewritten program but its rep axiom does not. In this case, the theorem prover treats $\mathcal{F}.a$ as an uninterpreted function.

Functionalization and pointwise axioms. There are more details to be presented about functionalization. We will introduce them with an example. Consider

$$\begin{array}{l}\textbf{spec var } a\colon T \to X\\ \textbf{var } c\colon T \to Y\\ \textbf{var } d\colon T \to Z\\ \textbf{depends } a[t\colon T] \textbf{ on } c[t], d[t]\end{array} \qquad (6)$$

Then occurrences of a are replaced by the expression $\mathcal{F}.a(c, d)$. Had c for example also been abstract, functionalization would continue, producing the expression $\mathcal{F}.a(\mathcal{F}.c(\ldots), d)$.

Notice that c, d, and $\mathcal{F}.a(c, d)$ are all maps. In typical functionalized expressions, we can expect to encounter expressions like $\mathcal{F}.a(c, d)[t]$. Allowing $\mathcal{F}.a$ to take maps as arguments is technically convenient, but without further restrictions, it would allow $\mathcal{F}.a(c, d)[t]$ to depend on the entire maps c and d, which we do not want: our view is that the dependency declaration 6) implies that $a[t]$ is unchanged by a modification to $c[s]$ or $d[s]$ for $s \neq t$. We enforce this point of view by imposing a *pointwise axiom* on each abstraction function. In the case of a, c, and d above, this axiom is:

$$\begin{array}{l}\langle \forall\, t\colon T,\ c0,\ c1,\ d0,\ d1\ ::\\ \qquad c0[t] = c1[t] \ \wedge\ d0[t] = d1[t]\\ \qquad \Rightarrow\ \mathcal{F}.a(c0, d0)[t] = \mathcal{F}.a(c1, d1)[t]\ \rangle\end{array} \qquad (7)$$

We would like to emphasize that in (7), variables $c0$, $c1$, $d0$, and $d1$ are dummies, not program variables. Even if program variables c and d were abstract, there would be no need to functionalize the dummies in (7).

For each abstract variable, there will be a pointwise axiom for each subtype of its index type, since different subtypes may have different dependencies. For example, consider

> **type** T
> **spec var** $a: T \to X$
> **type** $U <: T$
> **var** $c: U \to Y$
> **depends** $a[u: U]$ **on** $c[u]$
> **type** $V <: T$
> **var** $d: V \to Z$
> **depends** $a[v: V]$ **on** $d[v]$

There will be three pointwise axioms, one for each of the types T, U, and V. The axiom for T is (7), the same as the axiom where c and d had index type T. The axiom for U is

$$\langle \forall u: U,\ c0,\ c1,\ d0,\ d1 :: \\ \quad c0[u] = c1[u] \\ \quad \Rightarrow \mathcal{F}.a(c0, d0)[u] = \mathcal{F}.a(c1, d1)[u] \rangle$$

The axiom for V is similar.

When rewriting a postcondition, a post-value a' leads to post-values in the arguments to $\mathcal{F}.a$. For example, the postcondition of *init* for blank readers includes the conjunct

$$valid'[brd]$$

This is rewritten into

$$\mathcal{F}.valid(lo', cur', hi', buff', \mathcal{F}.svalid(rd, lo', hi', buff'))[rd]$$

The number of arguments of $\mathcal{F}.a$ depends on the number of dependencies of a that are visible in the scope where the rewriting takes place. For example, in the unit *RdRep*, $\mathcal{F}.svalid$ has four arguments, whereas in *BlankRdImpl*, $\mathcal{F}.svalid$ has five arguments because of the extra dependency of $svalid[brd]$ on $num[brd]$. Within the verification of any one unit, all occurrences of $\mathcal{F}.a$ have the same number of arguments.

Rep axioms. We now explain how a **rep** declaration is rewritten into a rep axiom. A **rep** declaration has the form

> **rep** $a[t\colon T] \;\equiv\; R$

where the only free variables allowed in R are fields that are dependencies of a, and each occurrence of such a field must be indexed by the dummy t. For definitiveness, suppose that these dependencies are

> **var** $c\colon T \to X$
> **var** $d\colon T \to Y$
> **depends** $a[t\colon T]$ **on** $c[t], d[t]$

This **rep** declaration is rewritten into the rep axiom

$$\langle \forall\, t\colon T,\; cV,\; dV \;::\; \mathcal{F}.a(cV, dV)[t] = R(c, d := cV, dV) \;\rangle$$

in which we use the assignment operator to denote substitution. In this axiom, we have appended V's in the names of the dummies to emphasize that they are universally quantified dummies, not the program variables c and d.

The same treatment works with minor alterations to accommodate subtype-specific **rep** declarations and dependencies. For example, the **rep** declarations in

> **type** T
> **spec var** $a\colon T \to W$
> **var** $c\colon T \to X$
> **depends** $a[t\colon T]$ **on** $c[t]$
>
> **type** $T0 <: T$
> **var** $d\colon T0 \to Y$
> **depends** $a[t\colon T0]$ **on** $d[t]$
> **rep** $a[t\colon T0] \;\equiv\; R0$
>
> **type** $T1 <: T$
> **var** $e\colon T1 \to Z$
> **depends** $a[t\colon T1]$ **on** $e[t]$
> **rep** $a[t\colon T1] \;\equiv\; R1$

produce the rep axioms

$$\langle \forall t \colon T0,\ cV,\ dV,\ eV \ :: \ \mathcal{F}.a(cV, dV, eV)[t] = R0(c, d := cV, dV) \ \rangle$$
$$\langle \forall t \colon T1,\ cV,\ dV,\ eV \ :: \ \mathcal{F}.a(cV, dV, eV)[t] = R1(c, e := cV, eV) \ \rangle$$

Note that different rep axioms are produced for the different **rep** declarations. Note also that all dependencies of a for any subtype become arguments to $\mathcal{F}.a$, and each axiom ignores those arguments that are irrelevant to its subtype.

Examples. In the unit *RdRep* described previously, the precondition of *refill* is written

$$valid[rd]$$

Using the static dependencies of $valid[rd]$, this precondition is rewritten into

$$\mathcal{F}.valid(lo, cur, hi, buff, svalid)[rd]$$

Since *svalid* is itself abstract, the rewriting continues:

$$\mathcal{F}.valid(lo, cur, hi, buff, \mathcal{F}.svalid(lo, hi, buff))[rd] \tag{8}$$

which is the final functionalized form of $valid[rd]$ in the scope *RdRep*.

As an example of a rep axiom, the **rep** for *valid* in *RdRep* is rewritten into

$$\begin{aligned}
&\langle \forall rd \colon Rd.T,\ loV,\ curV,\ hiV,\ buffV,\ svalidV \ :: \\
&\quad \mathcal{F}.valid(loV, curV, hiV, buffV, svalidV)[rd] \ \equiv \\
&\qquad rd \neq \mathbf{nil} \ \wedge \\
&\qquad 0 \leq loV[rd] \leq curV[rd] \leq hiV[rd] \ \wedge \\
&\qquad buffV[rd] \neq \mathbf{nil} \ \wedge \\
&\qquad hiV[rd] - loV[rd] \leq \mathbf{number}(buffV[rd]) \ \wedge \\
&\qquad svalidV[rd] \ \rangle
\end{aligned} \tag{9}$$

To see how these formulas work together, consider the verification of the method *refill*. The rewritten precondition (8) together with the rep axiom (9) allow the verifier to conclude that $buff[rd] \neq \mathbf{nil}$, by instantiating *buffV* to *buff*, *loV* to *lo*, and so on.

Because *RdRep* contains no **rep** declaration for *svalid*, $\mathcal{F}.svalid$ remains an uninterpreted function in this scope. A subtype-specific rep axiom is produced in the scope of an implementation of a reader subtype like *BlankRd*.

(In this description, we have glossed over some detailed rules that prevent representations for different subtypes from producing inconsistent values of an abstract variable at **nil**.)

Our final example illustrates reasoning about the abstraction function as an uninterpreted function symbol. Consider the following generic procedure, which replaces a reader's buffer, copying the contents of the old buffer into the new:

$$
\begin{array}{l}
\textbf{proc}\ \mathit{copyBuffer}(\mathit{rd}\colon \mathit{Rd.T}) \\
\quad \textbf{requires}\ \mathit{valid}[\mathit{rd}] \\
\quad \textbf{modifies}\ \mathit{state}[\mathit{rd}] \\
\textbf{impl}\ \mathit{copyBuffer}(\mathit{rd}\colon \mathit{Rd.T})\ \textbf{is} \\
\qquad \textbf{var}\ \mathit{nb} := \textbf{new}(\textbf{byte}, \textbf{number}(\mathit{buff}[\mathit{rd}]))\ \textbf{in} \\
\qquad\quad \textbf{for}\ i := 0\ \textbf{to}\ \textbf{number}(\mathit{buff}[\mathit{rd}]) - 1\ \textbf{do} \\
\qquad\qquad \mathit{nb}[i] := \mathit{buff}[\mathit{rd}][i] \\
\qquad\quad \textbf{end}\ ; \\
\qquad\quad \mathit{buff}[\mathit{rd}] := \mathit{nb} \\
\qquad \textbf{end} \\
\quad \textbf{end}
\end{array}
$$

The proof that *copyBuffer* maintains $\mathit{valid}[\mathit{rd}]$ boils down to proving

$$
\begin{array}{l}
\mathit{lo}[\mathit{rd}] = \mathit{lo}'[\mathit{rd}]\ \wedge\ \mathit{hi}[\mathit{rd}] = \mathit{hi}'[\mathit{rd}]\ \wedge\ \mathit{elems}[\mathit{buff}[\mathit{rd}]] = \mathit{elems}'[\mathit{buff}'[\mathit{rd}]] \\
\Rightarrow \\
\mathit{svalid}[\mathit{rd}] = \mathit{svalid}'[\mathit{rd}]
\end{array}
$$

which functionalizes into

$$
\begin{array}{l}
\mathit{lo}[\mathit{rd}] = \mathit{lo}'[\mathit{rd}]\ \wedge\ \mathit{hi}[\mathit{rd}] = \mathit{hi}'[\mathit{rd}]\ \wedge\ \mathit{elems}[\mathit{buff}[\mathit{rd}]] = \mathit{elems}'[\mathit{buff}'[\mathit{rd}]] \\
\Rightarrow \\
\mathcal{F}.\mathit{svalid}(\mathit{lo}, \mathit{hi}, \mathit{buff})[\mathit{rd}] = \mathcal{F}.\mathit{svalid}(\mathit{lo}', \mathit{hi}', \mathit{buff}')[\mathit{rd}]
\end{array}
$$

But this cannot be proved, since distinct arrays may have the same elements. Thus, the checker would reject *copyBuffer*, warning that it possibly destroys the validity of *rd*. This warning is accurate, since generic code is not allowed to modify *buff*. For example, the design of readers allows a subtype to cache the buffer pointer, but such a cache would be invalidated unexpectedly by *copyBuffer*. Thus, reasoning about the abstraction function as an uninterpreted function symbol enforces the read-only by specification idiom.

An alternative design for readers would have replaced the dependency

depends $svalid[rd]$ **on** $buff[rd]$

by

depends $svalid[rd]$ **on** $elems[buff[rd]]$

In this design, *copyBuffer* would be legal, and it would be illegal for subtypes to assume that the buffer pointer remains unchanged by generic code. This happens not to be the approach taken by the Modula-3 I/O library.

So much for rewriting preconditions and postconditions. Now we consider rewriting modifies lists.

5.1 Modifies list desugaring

Guided by the abstraction modification principle (page 19), we introduce a closure operation on modifies lists. The closure operation expands the modifies list as required by the first half of the principle without expanding it so much as to violate the second half of the principle. The rewritten specification allows a method to modify a field $f[s]$ (abstract or concrete) if and only if the closure of the method's modifies list includes $f[s]$. Thus the rewriting is parameterized by the definition of closure. In this section, we first define the rewriting from a closed modifies list, and then define the closure operation appropriate for static dependencies. In Section 7.1, we will define the closure operation for dynamic dependencies.

Modification constraints. Closed modifies lists are rewritten into *modification constraints*. Consider a specification

modifies M **ensures** P (10)

occurring in a scope D. We rewrite this specification into

modifies N **ensures** $P \wedge Q$

where N is the list of all concrete maps f for which a term of the form $f[E]$ occurs in the closure of M, and Q is a conjunction with one conjunct for each map variable visible in the scope. The conjunct for a particular map f asserts that $f[s]$ changes only where it is allowed to change. That is, if $\{f[E_1], \ldots, f[E_n]\}$ is the set of terms in the closure of M of the form $f[\ldots]$ (that is, the set of terms whose outer map variable is f), then the conjunct for f is

$$\langle \forall s :: f[s] = f'[s] \ \vee \ s = E_1 \ \vee \ \ldots \ \vee \ s = E_n \rangle$$

We call this conjunct the *modification constraint* for f, and we call $\{E_1, \ldots, E_n\}$ the set of *modification points* of f. The modification constraint for a map variable limits the points at which the variable may be modified, that is, it protects the variable from change at other points. In particular, if the dependencies of an abstract variable a are changing at points where a itself is not allowed to be changed, a's modification constraint limits the modification of a's representation to preserve the values at points where a is not allowed to change. We say that a is protected from changes to its representation.

(When verifying an implementation, ESC does not bother to produce a modification constraint for a map if a syntactic scan of the implementation determines that the map is never changed by the implementation.)

This strengthening of the postcondition occurs before the postcondition is functionalized.

Closure definition. A set of terms M is *statically closed* in a scope D if

$$a[E] \in M \ \wedge \ \text{“}\textbf{depends } a[t] \textbf{ on } c[t]\text{”} \in D \ \Rightarrow \ c[E] \in M$$

(We have intentionally ignored the type of E and the index types of a and c in this definition. Thus, a closed set of terms may include $f[E]$ even if E is not of the index type of f. Actually, ESC does use type information to produce a smaller closure, but in retrospect, we don't think it makes much difference.)

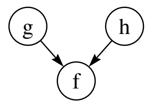
The *static closure* of a modifies list is its smallest statically closed superset.

For example, in the scope of the unit *BlankRdImpl*, the static closure of $valid[brd]$ is

$$lo[brd],\ cur[brd],\ hi[brd],\ buff[brd],\ svalid[brd],\ num[brd]$$

Example. As an artificial example, suppose that f is a concrete field and consider the dependencies

depends $g[t]$ **on** $f[t]$
depends $h[t]$ **on** $f[t]$



and the modifies list

> **modifies** $g[u], f[v]$

The static closure of this modifies list is

> **modifies** $g[u], f[u], f[v]$

This produces the rewritten specification

> **modifies** f
> **ensures** $\langle \forall s :: g[s] = g'[s] \lor s = u \rangle \land$
> $\langle \forall s :: f[s] = f'[s] \lor s = u \lor s = v \rangle \land$
> $\langle \forall s :: h[s] = h'[s] \rangle$

Notice that since there are no modification points for h, the conjunct for h in the rewritten postcondition does not allow it to be changed anywhere. Thus, the second half of the abstraction modification principle is satisfied: the license to modify $g[u]$ does not imply the license to modify $h[u]$, even though $g[u]$ and $h[u]$ have the concrete dependency $f[u]$ in common. Also, the license to modify $f[v]$ does not imply the license to modify $g[v]$ or $h[v]$.

Finally, functionalization produces

> **modifies** f
> **ensures** $\langle \forall s :: \mathcal{F}.g(f)[s] = \mathcal{F}.g(f')[s] \lor s = u \rangle \land$
> $\langle \forall s :: f[s] = f'[s] \lor s = u \lor s = v \rangle \land$
> $\langle \forall s :: \mathcal{F}.h(f)[s] = \mathcal{F}.h(f')[s] \rangle$

That is, the specification allows changes to f at indices u and v, provided the changes preserve the value of $\mathcal{F}.h(f)[s]$ for all s, and $\mathcal{F}.g(f)[s]$ for all s except u.

6 Soundness of modular verification

We remind the reader that we are interested in modular soundness, that is, the property that the separate verifications of the individual modules of the program suffice to ensure the correctness of the composite program.

The standard approach for reasoning about procedure calls breaks down for modular programs. The standard approach reasons about a procedure call by assuming that it meets its specification, and discharges this assumption by verifying the implementation of the procedure. The approach breaks down if the specification is interpreted differently in the two contexts. But as we have seen, the meaning of a modifies list depends on the scope in which it is used. In particular, it may be desugared differently when reasoning about a call to a procedure than when reasoning about the implementation of the procedure.

To be more precise about modular soundness, we will define *scope monotonicity*, which means that anything verifiable in a scope is also verifiable in any larger scope. Then, we will argue that modular soundness is equivalent to scope monotonicity. The notion of scope monotonicity seems to be new.

For a scope D and a procedure implementation P in D, the judgment

$$D \vdash P$$

means that P meets its specification in D. More precisely, let

$$\textbf{requires}\ \mathit{Pre}\ \textbf{modifies}\ M\ \textbf{ensures}\ \mathit{Post} \tag{11}$$

be the result of desugaring the specification of P in scope D, as described in Section 5. Let A be the body of P, and let R be the conjunction of pointwise axioms and rep axioms in D, as described in Section 5.0. The requirement is that R implies that A meets the specification (11). In checking this, the verification condition generator reasons about method calls within A by using their specifications as desugared in D.

We say that $\vdash$ is *monotonic with respect to scope* if, for any procedure implementation P and scopes D and E,

if $D \subseteq E$, then $D \vdash P$ implies $E \vdash P$

If we can prove that $\vdash$ is monotonic with respect to scope, then it is reasonable to say that our modular verification system is sound. For, if P has been verified in a scope D, that is, if we have proved $D \vdash P$, it follows by monotonicity that $E \vdash P$, where E is the entire program. Thus, anything that verifies in a limited scope would also verify had there been no information hiding and all information had been global.

It is too much to hope that $\vdash$ be monotonic in any program whatsoever. We will impose some requirements, called *modularity requirements*, such that $\vdash$ is monotonic in any program that meets the requirements. We will also argue that these requirements are reasonable from a methodological point of view, that is, that they don't rule out useful designs.

Our notion of modular soundness is different from the soundness of an axiomatic semantics with respect to an operational semantics. The consistency of axiomatic and operational semantics is certainly important, but it concerns the conventional control structures of programming in the small, like iteration and conditionals. These are mostly irrelevant to the issues of information hiding in programming in the large, which are the issues of concern in this paper. In this paper, we simply assume that the standard operational semantics is consistent with the axiomatic semantics of a single-module program. Therefore, any discrepancy between the axiomatic and operational semantics is due to unsound modular verification.

6.0 Visibility requirement

Our first modularity requirement is the *visibility requirement*. A program satisfies the visibility requirement if each of its static dependencies

depends $a[t]$ **on** $c[t]$

is visible in every scope in which both a and c are visible.

It is easy to see that this requirement is necessary to have any hope of achieving scope monotonicity. Suppose there were a scope where a and c were visible but the dependency is not. In such a scope, it is provable that a change to c has no effect on a. But this would not be provable in a larger scope where the dependency is visible.

The requirement is necessary for informal as well as formal checking. If a program violated the requirement, it would be impossible to reason about a and c in the scope where they are visible but the dependency is not. In such a scope, an assignment to c could change a unexpectedly, and a call to a procedure that modifies a could change c unexpectedly. Nothing in the program text warns of either side effect. Almost all failures of scope monotonicity can be traced to unexpected side effects of this sort.

For example, consider what would happen if the dependency

depends $svalid[rd]$ **on** $hi[rd]$

were placed not in unit $RdRep$ but in unit $BlankRdImpl$. A modular checker would then allow the generic implementation, where the dependency of $svalid$ on hi is not visible, to increase the value of $hi[rd]$ beyond $num[rd]$, which for blank readers destroys validity.

6.1 Top-down requirement

The second modularity requirement is the *top-down requirement*. A program satisfies the top-down requirement if, for each of its static dependencies

$$\textbf{depends}\ a[t]\ \textbf{on}\ c[t]$$

variable a is visible in every scope in which c is visible.

Here's an example of a rather pathological program unit that violates the top-down requirement.

$$
\begin{array}{l}
\textbf{unit}\ U\ \textbf{import}\ Rd, RdRep \\
\quad \textbf{type}\ T <: Rd.T \\
\quad \textbf{spec var}\ isEven: T \to \textbf{bool} \\
\quad \textbf{depends}\ isEven[t: T]\ \textbf{on}\ cur[t] \\
\quad \textbf{rep}\ isEven[t: T] \ \equiv\ cur[t]\ \textbf{mod}\ 2 = 0 \\
\quad \textbf{proc}\ P(t: T) \\
\qquad \textbf{requires}\ Rd.valid[t], isEven[t] \\
\qquad \textbf{modifies}\ Rd.state[t] \\
\quad \textbf{impl}\ P(t: T)\ \textbf{is} \\
\qquad\quad t.getChar()\ ;\ \textbf{assert}\ cur[t]\ \textbf{mod}\ 2 = 0 \\
\qquad \textbf{end}
\end{array}
$$

This pathological unit would verify, since

- the precondition of P requires $isEven[t]$,
- $isEven[t]$ does not appear in the modifies list of $getChar$, and consequently, $isEven[t]$ is formally provable at the exit of the call to $t.getChar()$, and
- the representation of $isEven[t]$ implies $cur[t]\ \textbf{mod}\ 2 = 0$.

But of course the assert would fail at run-time, since $getChar$ will change the parity of $cur[t]$.

At first, this problem may not seem like a failure of scope monotonicity, but it is. The $getChar$ method verified in the scope $RdImpl$, where it was presented earlier in the paper. But if the scope $RdImpl$ were expanded by importing the unit U, then $getChar$ would no longer verify, because it does not preserve the value of $isEven[t]$.

To put it another way, the problem is that $isEven$ is not visible in the scope where $getChar$ is implemented, and therefore the desugaring of $getChar$'s specification does not strengthen the postcondition to protect $isEven$ from change. The top-down requirement ensures that $isEven$ is visible wherever cur is, and thus any procedure that modifies cur and claims not to modify $isEven$ will be checked appropriately.

Here is an explanation of the name of this requirement. The reader package was designed in a top-down fashion, and cur was introduced as part of the concrete representation of $Rd.state$ and $Rd.valid$. To come along later and define a new unit (U) that attempts to use cur for part of the representation of something else ($isEven$) would be a violation of top-down design. We believe that imposing the top-down requirement for static dependencies does not rule out any useful designs. As we shall see in Section 7, the situation is more interesting for dynamic dependencies.

6.2 Static placement rule

There is a simple discipline that guarantees that both the visibility and top-down requirements are satisfied, called the *static placement rule*: simply place each static dependency

depends $a[t]$ **on** $c[t]$

in the unit that declares c. We leave it to the reader to show that the visibility and top-down requirements follow from this rule. Furthermore, the converse is almost true: for programs without cyclic imports, if both requirements are satisfied, then the static placement rule is satisfied as well. Thus, if we'd like, we can replace both requirements by the rule. We have stated the requirements separately, because they seem to be separable concerns, and are used in different parts of the soundness proof.

6.3 Residues

The visibility and top-down requirements are two giant steps toward modular soundness. But they don't quite reach the goal. If they did, then the following implication would be true, for any procedure implementation P and scopes D

and E containing static dependencies only:

$D \vdash P$ and $D \subseteq E$ and E satisfies the two modularity requirements
$\Rightarrow$
$E \vdash P$

Unfortunately, this is false according to the way of functionalizing abstract variables described in Section 5.0. There is one more technicality that must be introduced to fix the problem, called *residues*. Here is an artificial program that demonstrates the problem:

unit A
  type T
  spec var $a\colon T \to$ **any**
  var $c\colon T \to$ **int**
  depends $a[t\colon T]$ **on** $c[t]$
  proc $outer(t\colon T)$
  proc $inner(t\colon T)$ **modifies** $a[t]$ **ensures** $c[t] = c'[t]$
  impl $outer(t\colon T)$ **is** $t.inner()$ **end**

The absence of a modifies list for $outer$ means that a call to $outer$ has no side effects. We will now argue that without residues, unit A verifies. We then argue that it should not verify. Finally, we will define residues and explain how they fix the problem.

As described in Section 5, the modifies list $a[t]$ of the call $t.inner()$ has the static closure $a[t], c[t]$, so the rewritten specification of $t.inner()$ (before functionalization) is

modifies c
ensures $c[t] = c'[t] \wedge$
$$\langle \forall s \ :: \ c[s] = c'[s] \ \vee \ s = t \rangle \wedge$$
$$\langle \forall s \ :: \ a[s] = a'[s] \ \vee \ s = t \rangle$$

After functionalization, the specification is

modifies c
ensures $c[t] = c'[t] \wedge$
$$\langle \forall s \ :: \ c[s] = c'[s] \ \vee \ s = t \rangle \wedge$$
$$\langle \forall s \ :: \ \mathcal{F}.a(c)[s] = \mathcal{F}.a(c')[s] \ \vee \ s = t \rangle$$

The first two lines imply that $c[s]$ does not change for any s. The third line then implies that $a[s]$, that is, $\mathcal{F}.a(c)[s]$, also does not change for any s. Therefore, the call $t.inner()$ has no side effects at all, and the body of $outer$ will verify.

But we now argue that $outer$'s body should not verify. Consider the following unit B, providing an implementation of $inner$.

> **unit** B **import** A
>   **var** $d: A.T \to$ **int**
>   **depends** $a[t: A.T]$ **on** $d[t]$
>   **impl** $inner(t: A.T)$ **is** $d[t] := 0$ **end**

Unit B reveals another dependency (d) of a, which the implementation of $inner$ in fact modifies. Unit B will verify in isolation, because $inner$ modifies only variables in the static closure of its modifies list $a[t]$.

We are in trouble, because $outer$'s side effect on d will be unexpected in a scope that sees d together with $outer$'s specification:

> **unit** C **import** A, B
>   **proc** $R(t: A.T)$ **modifies** $a[t]$
>   **impl** $R(t: A.T)$ **is**
>   **var** $dd := d[t]$ **in** $t.outer()$; **assert** $dd = d[t]$ **end**
>   **end**

This implementation verifies, because $outer$'s modifies list does not include d, but clearly the assert will fail at run-time.

This is a failure of scope monotonicity, because although $outer$'s body verifies in the unit A, it would not do so in the larger scope of unit B, where d is visible and the call $t.inner()$ will be desugared to have a side effect on $d[t]$.

We blame the failure on the body of $outer$. Here's an informal explanation of why: Procedure $outer$, which is specified to be side-effect free, calls $inner$, which modifies a. Although a depends on c, it should not be inferred that a depends only on c. Therefore, the call to $inner$ should be inconsistent with $outer$'s modifies list.

Individual residues. To change our rewriting so that $outer$'s body will not verify, we introduce residues. The *residue* of an abstract variable a, written $res.a$, can be viewed as a stand-in for those of a's dependencies that are not visible. Residues are introduced automatically by the verifier and cannot be mentioned explicitly in specifications or programs. The verifier treats every abstract variable declaration

$$\textbf{spec var } a\colon T \rightarrow X$$

as a shorthand for the three declarations

$$\begin{array}{l} \textbf{spec var } a\colon T \rightarrow X \\ \textbf{var } res.a\colon T \rightarrow \textbf{any} \\ \textbf{depends } a[t\colon T] \textbf{ on } res.a[t] \end{array}$$

The implicit dependency of a on $res.a$ introduces $res.a$ into the static closure of any modifies list that mentions a, just as for explicit dependencies.

Desugaring of modifies lists as described in Section 5 will now work out soundly for this example. The modifies list $a[t]$ of the call $t.inner()$ in the body of $outer$ has the static closure $a[t], res.a[t], c[t]$, so the rewritten specification of $t.inner()$ (before functionalization) is

$$\begin{array}{l} \textbf{modifies } c, res.a \\ \textbf{ensures } c[t] = c'[t] \wedge \\ \qquad \langle \forall s :: c[s] = c'[s] \vee s = t \rangle \wedge \\ \qquad \langle \forall s :: res.a[s] = res.a'[s] \vee s = t \rangle \wedge \\ \qquad \langle \forall s :: a[s] = a'[s] \vee s = t \rangle \end{array}$$

This allows both $a[t]$ and $res.a[t]$ to change, and therefore the implementation of $outer$ will not verify.

Shared residues. We are now very close to modular soundness, so close that it took our colleague Jim Saxe to find a sufficiently pathological example to demonstrate that we are not yet there. The example is shown in Figure 7. In this scope, the implementation of $outer$ verifies, because $a[t]$ and $res.a[t]$ are allowed to be changed, $c[t]$ is restored to its initial value, $res.b[t]$ is not changed by the body, and the invariance of $b[t]$ (*i.e.*, of $\mathcal{F}.b(res.b, c)[t]$) follows from the invariance of $res.b$ and c. But in a larger scope in which it is revealed that a and b have a common dependency, $outer$ will not verify:

$$\begin{array}{l} \textbf{unit } E \textbf{ import } D \\ \quad \textbf{var } d\colon D.T \rightarrow \textbf{int} \\ \quad \textbf{depends } a[t\colon D.T] \textbf{ on } d[t] \\ \quad \textbf{depends } b[t\colon D.T] \textbf{ on } d[t] \end{array}$$

```
unit D
  type T
  spec var a: T → int
  spec var b: T → int
  var c: T → int
  depends b[t: T] on c[t]
  proc outer(t: T) modifies a[t]
  proc inner(t: T) modifies a[t]
  impl outer(t: T) is
      var cc := c[t] in
        c[t] := 0 ; t.inner() ; c[t] := cc
      end
    end
```

Figure 7: Example program that motivates shared residues.

In scope E, the required proof of invariance of $b[t]$ for $outer$ does not go through. The modification constraint for b that is added to the postcondition of $outer$ is

$$\langle \forall s :: \mathcal{F}.b(res.b, \grave{c}, \grave{d})[s] = \mathcal{F}.b(res.b, \acute{c}, \acute{d})[s] \rangle \tag{12}$$

where the accents on c and d denote their initial and final values. We do not accent $res.b$, because nothing in this example modifies it. The modification constraints for b and d that we get to assume at exit from $inner$ are

$$\begin{aligned} &\langle \forall s :: \grave{d}[s] = \acute{d}[s] \ \vee \ s = t \rangle \wedge \\ &\langle \forall s :: \mathcal{F}.b(res.b, \bar{c}, \grave{d})[s] = \mathcal{F}.b(res.b, \bar{c}, \acute{d})[s] \rangle \end{aligned} \tag{13}$$

where $\bar{c}$ denotes the value of c on entry and exit of $inner$, that is,

$$\bar{c} = store(\grave{c}, t, 0)$$

where the expression $store(\grave{c}, t, 0)$ denotes a map like $\grave{c}$ but mapping t to 0. Because d is unmodified except in the call to $inner$, $\grave{d}$ and $\acute{d}$ serve to denote the values of d around the call to $inner$ as well as around the implementation of $outer$.

But (12) does not follow from (13). Although *inner* is constrained to modify d only in ways that preserve $\mathcal{F}.b(res.b, c, d)$, this constraint is in force only for the value of c at the time of the call to *inner*. Therefore, scope monotonicity and modular soundness do not hold.

To restore modular soundness, we must arrange either that *outer* verifies in unit E or that it does not verify in unit D. We choose the latter, that is, we take the view that *outer* was misprogrammed: modifying part of the representation (c) of an abstraction (b) whose representation is hidden and then calling a method (*inner*) that may manipulate the abstraction is methodologically unjustifiable, even if the modification of c is restored after the call.

Consider that the example might continue as follows:

rep $b[t\colon T] \;\equiv\; c[t] \cdot d[t]$
impl $inner(t\colon T)$ **is**
  if $c[t] = 0$ **then** $d[t] := d[t] + 1$ **end**
 end

This possible continuation shows clearly that the failure of *outer* to verify in unit E is appropriate, and therefore its verification in unit D is inappropriate.

The essential difficulty revealed by Saxe's example is that two abstract variables that have no common dependency in a small scope may turn out to have a common dependency in a larger scope. To fix our proof system to be modularly sound, we will force all small-scope verifications to respect the possibility that larger scopes may reveal common dependencies. To do this, we introduce another residue variable, a *shared residue* $sres$ to augment the individual residues introduced earlier.

In more detail, $sres$ is a predeclared variable visible in all scopes. The verifier treats every abstract variable declaration

spec var $a\colon T \to X$

as shorthand for

spec var $a\colon T \to X$
depends $a[t\colon T]$ **on** $sres[t]$
var $res.a\colon T \to$ **any**
depends $a[t\colon T]$ **on** $res.a[t]$

The combination of individual and shared residues achieves modular soundness. For Saxe's example, the attempted verification of *outer* in unit D will now fail: the details of the failure are exactly the previously described details of the failure of *outer* to verify in unit E (see formulas (12) and (13)) with *sres* playing the rôle of d.

It seems necessary to introduce both the shared residue and the individual residues. Here is an example that shows that the shared residue alone does not suffice for modular soundness. We begin with a small unit G:

unit G
  type T
  spec var $a: T \to X$
  spec var $b: T \to Y$
  proc $outer(t: T)$ **modifies** $a[t]$
  proc $inner(t: T)$ **modifies** $b[t]$ **ensures** $b[t] = b'[t]$
  impl $outer(t: T)$ **is** $t.inner()$ **end**

The following unit H shows that in a larger scope, the call to *inner* may have side effects that are not allowed by *outer*'s specification:

unit H **import** G
  var $c: T \to Z$
  depends $b[t: T]$ **on** $c[t]$

But with the shared residue variable only, *inner*'s modification to the shared residue is consistent, in unit G, with *outer*'s modification constraint. To achieve the verification failure that we need, we must distinguish $res.a$ from $res.b$.

6.4 Modular soundness for static dependencies

The appendix contains a proof of modular soundness for programs whose dependencies are static, and that satisfy the visibility and top-down requirements, given that specifications are desugared as described in Section 5 and residues are used.

This marks the end of our presentation of static dependencies. In the next section, we describe dynamic dependencies.

7 Dynamic dependencies

Most of the dependencies that arise in top-down program design are static. By a top-down design, we mean a design in which each successive layer of implementation provides the representation of the abstraction specified in layers above.

However, not all useful designs are top-down. A bottom-up design is often better, in which an object type is defined and later used to build higher-level objects, which may not even have been envisioned at the time the first type was defined. Most of the dependencies that arise in bottom-up design are dynamic.

Recall that a dynamic dependency has the form

$$\textbf{depends}\ a[t]\ \textbf{on}\ c[b[t]]$$

This means that the abstract state $a[t]$ is represented in terms of the concrete state $c[b[t]]$, which is a field not of the object t but of the separate object $b[t]$. The field b is called a *pivot field*. Pivot fields introduce a level of indirection that makes dynamic dependencies more complicated than static dependencies. Static dependencies allow the representation of an abstraction to be divided among several modules; dynamic dependencies allow it also to be divided among several dynamically allocated objects.

For example, sequences are useful abstractions. To define sequences and then use them in different ways is a bottom-up approach and leads to the use of dynamic dependencies. To see this, consider a set type $Set.T$ implemented in terms of a sequence type $Seq.T$. Somewhere in the set implementation, there will be a field, say q, declared as

$$\textbf{var}\ q\colon Set.T \to Seq.T$$

The representation of the validity and state of a set s will inevitably involve properties of the sequence $q[s]$. Almost always, for example, set validity requires validity of the underlying sequence, in which case we have

$$\textbf{rep}\ Set.valid[s\colon Set.T] \equiv\ \ldots\ \wedge\ Seq.valid[q[s]]$$

This **rep** declaration requires the dependency

$$\textbf{depends}\ Set.valid[s\colon Set.T]\ \textbf{on}\ Seq.valid[q[s]]$$

which is dynamic, with pivot field q.

Although we have built a checker that handles dynamic dependencies, we don't understand them as well as static dependencies. In particular, we haven't proved any soundness theorem about them, and our view of their modularity requirements is still evolving.

In this section, we will explain how dynamic dependencies affect functionalization and modifies list desugaring, and then explain what we believe about their modularity requirements.

7.0 Functionalization

Functionalization in the presence of dynamic dependencies is analogous to functionalization in the presence of static dependencies only. Both of the fields in the right-hand side of the dynamic dependency become arguments to the abstraction function.

For example, in the presence of the dependencies

$$\textbf{depends}\ a[t]\ \textbf{on}\ e[t]$$
$$\textbf{depends}\ a[t]\ \textbf{on}\ c[b[t]]$$

the functionalized form of $a[x]$ is

$$\mathcal{F}.a(e, c, b)[x]$$

or, more precisely, taking residues into account,

$$\mathcal{F}.a(sres, res.a, e, c, b)[x]$$

The pointwise axiom for $\mathcal{F}.a$ is

$$\begin{aligned}
&\langle\, \forall\, t, sres0, sres1, res.a0, res.a1, e0, e1, c0, c1, b0, b1 ::\\
&\qquad sres0[t] = sres1[t] \;\wedge\; res.a0[t] = res.a1[t] \;\wedge\; e0[t] = e1[t] \;\wedge\\
&\qquad c0[b0[t]] = c1[b1[t]]\\
&\quad\Rightarrow\\
&\qquad \mathcal{F}.a(sres0, res.a0, e0, c0, b0)[t] \;=\\
&\qquad \mathcal{F}.a(sres1, res.a1, e1, c1, b1)[t]\, \rangle
\end{aligned}$$

We don't introduce anything like residue variables for dynamic dependencies.

7.1 Modifies list desugaring

Unlike functionalization, which is pretty much the same for static and dynamic dependencies, modifies list desugaring is surprisingly different in the two cases. It has taken us several tries to converge on a desugaring that suits all the examples that we know.

In this subsection, we assume that no abstract variable depends, directly or indirectly, on itself. This restriction will be lifted in Section 9.0.

To explain the issues, we start by exploring the obvious extension of the approach for static dependencies, and show how this goes wrong. Then we give what we think is the right desugaring, followed by two more supporting examples. Finally, we impose a restriction that seems to be necessary to make the desugaring sound.

Recall the main points of Section 5.1:

- the definition of closure,
- the rule that **modifies** M allows the modification of anything in the closure of M, and
- the modification constraints that enforce the rule.

We will reuse the second and third points. That is, to accommodate dynamic dependencies, we redefine closure and leave everything else the same.

The need to close upwards. Recall that a set of terms M is statically closed in a scope D if it satisfies the property

$$a[E] \in M \ \wedge \ \text{“}\textbf{depends}\ a[t]\ \textbf{on}\ c[t]\text{”} \in D \ \Rightarrow \ c[E] \in M \tag{14}$$

The obvious extension to include dynamic dependencies is to require in addition:

$$a[E] \in M \ \wedge \ \text{“}\textbf{depends}\ a[t]\ \textbf{on}\ c[b[t]]\text{”} \in D \ \Rightarrow \ c[b[E]] \in M \tag{15}$$

To give this new closure a name, we define a set of terms M to be *downward closed* in a scope D if it satisfies (14) and (15). Will we get a good desugaring if we replace “static closure” with “downward closure” in Section 5.1? Unfortunately not.

To explain why the replacement doesn't work, we give a straightforward example of integer sets implemented in terms of extensible integer sequences. Figure 8 shows the two interfaces, together with ESC-style specifications. (In these interfaces, we have varied our convention and elected not to return anything from the *init* methods.) A simple implementation of all *Set* objects, in which all elements are kept in a sequence with duplicates allowed, begins as shown in Figure 9.

The whole point of this example is: what will be the effective modifies list used in reasoning about the call to $Seq.init$ in the body of $Set.init$? Since $Seq.init(sq)$ modifies $valid[sq]$, $state[sq]$, and $length[sq]$, the modifies list (before closure) of the call $q[st].init()$ is

$$\textbf{modifies}\ Seq.valid[q[st]], Seq.state[q[st]], Seq.length[q[st]]$$

unit *Set*
  type T
  spec var $valid: T \to$ **bool**
  spec var $state: T \to$ **any**
  proc $init(st: T)$
    modifies $valid[st], state[st]$
    ensures $valid'[st]$
  proc $insert(st: T,\ i\colon$ **int**$)$
    requires $valid[st]$
    modifies $state[st]$
  proc $delete(st: T,\ i\colon$ **int**$)$
    requires $valid[st]$
    modifies $state[st]$
  proc $member(st: T,\ i\colon$ **int**$)\colon$ **bool**
    requires $valid[st]$

unit *Seq*
  type T
  spec var $valid: T \to$ **bool**
  spec var $length: T \to$ **int**
  spec var $state: T \to$ **any**
  proc $init(sq: T)$
    modifies $valid[sq], state[sq], length[sq]$
    ensures $valid'[sq] \ \wedge\ length'[sq] = 0$
  proc $addhi(sq: T,\ i\colon$ **int**$)$
    requires $valid[sq]$
    modifies $state[sq], length[sq]$
    ensures $length'[sq] = length[sq] + 1$
  proc $get(sq: T,\ i\colon$ **int**$)\colon$ **int**
    requires $valid[sq] \ \wedge\ 0 \le i < length[sq]$

Figure 8: The interfaces *Set* for sets and *Seq* for sequences.

unit *SetImpl* **import** *Set*, *Seq*
  var q: $Set.T \to Seq.T$
  rep $valid[st{:}\, Set.T] \;\equiv\; st \neq \textbf{nil} \;\wedge\; q[st] \neq \textbf{nil} \;\wedge\; Seq.valid[q[st]]$
  depends $valid[st{:}\, Set.T]$ **on** $q[st]$, $Seq.valid[q[st]]$
  depends $state[st{:}\, Set.T]$ **on** $Seq.state[q[st]]$, $Seq.length[q[st]]$
  impl $init(st{:}\, Set.T)$ **is**
      $q[st] := \textbf{new}(Seq.T)\ ;\ q[st].init()$
    end
  impl $insert(st{:}\, Set.T,\ i{:}\, \textbf{int})$ **is** $q[st].addhi(i)$ **end**
   ⋮

Figure 9: The implementation unit *SetImpl*.

This is also the modifies list after closure, since it is already downward closed (not counting residues, which we will ignore since they are irrelevant to this example). Transforming the closed modifies list into modification constraints, the postcondition of the rewritten specification is

$$
\begin{aligned}
\textbf{ensures}\ & \langle \forall\, sqv :: \ Seq.valid[sqv] = Seq.valid'[sqv] \ \vee\ sqv = q[st] \ \rangle \wedge \\
& \langle \forall\, sqv :: \ Seq.state[sqv] = Seq.state'[sqv] \ \vee\ sqv = q[st] \ \rangle \wedge \\
& \langle \forall\, sqv :: \ Seq.length[sqv] = Seq.length'[sqv] \ \vee\ sqv = q[st] \ \rangle \wedge \\
& \langle \forall\, stv :: \ Set.valid[stv] = Set.valid'[stv] \ \rangle \wedge \\
& \langle \forall\, stv :: \ Set.state[stv] = Set.state'[stv] \ \rangle \wedge \\
& \langle \forall\, stv :: \ q[stv] = q'[stv] \ \rangle
\end{aligned}
$$

The fourth conjunct “protects” the higher-level abstraction *Set.valid* from a change to its representation. In the old world of static dependencies only, this was necessary, but in the new world with dynamic dependencies, it is preposterous. The whole purpose of the *Seq.init* call is to modify the validity of the enclosing set.

The example shows that using the downward closure produces too strong an **ensures** clause, that is, too small a closure.

Let us summarize what the example has shown about the difference between static dependencies and dynamic dependencies. In the presence of a static dependency of $a[t]$ on $c[t]$, the presence of the term $c[x]$ in the modifies list does not, and should not, imply the presence of $a[x]$ in the closure, since the license to modify a concrete variable does not imply the license to modify an abstract variable that depends on it. However, in the presence of a dynamic dependency of $a[t]$ on $c[b[t]]$, the example shows that the presence of the term $c[x]$ in the modifies list should imply the presence of $a[t]$ in the closure, for any t such that $b[t] = x$. That is, we must close upwards as well as downwards.

Dynamic closure. In the next few paragraphs, we define the closure that we use when desugaring modifies lists in the presence of dynamic dependencies, which we call the *dynamic closure*. We have already indicated that it is larger than the downward closure. In fact, it is the union of the downward closure with a portion of the upward closure (defined soon).

Another change from our previous treatment is that the closure will contain expressions of the form f^{-1}. We call these "map inverses", but they are not to be thought of as ordinary notation, for example, the notation does not imply that f is invertible: they are a syntactic fiction that will be eliminated when the closure is transformed into a modification constraint. The elimination is achieved by rewriting an equality of the form

$$s = f_1^{-1}[f_2^{-1}[\cdots[f_n^{-1}[E]]]] \tag{16}$$

into

$$f_n[\cdots[f_2[f_1[s]]]] = E$$

All of the map inverses will be eliminated by this rewriting, because the terms of the closure of a modifies list affect the rewritten specification only in modification constraints, in which map inverses will occur only in equalities of the form (16).

The *dynamic closure* of a modifies list M in a scope D is the union of the downward closure of M with the upward closure of the flexible subset of M.

The *flexible subset* of a set of terms M in a scope D consists of those terms $f[E]$ where D contains no dependency of the form **depends** $a[t]$ **on** $f[t]$.

A set of terms M is *upward closed* in a scope D if

$$\begin{array}{l} c[E] \in M \ \wedge \ \text{“}\textbf{depends}\ a[t]\ \textbf{on}\ c[t]\text{”} \in D \ \Rightarrow \ a[E] \in M \\ c[E] \in M \ \wedge \ \text{“}\textbf{depends}\ a[t]\ \textbf{on}\ c[b[t]]\text{”} \in D \ \Rightarrow \ a[b^{-1}[E]] \in M \end{array}$$

The *upward closure* of a set of terms is its smallest upward-closed superset.

Examples. Let use redo the $Set.init$ example with the new rule. The desugaring begins, as before, with the modifies list from the specification, namely

$$\textbf{modifies}\ Seq.valid[q[st]],\ Seq.state[q[st]],\ Seq.length[q[st]]$$

The dynamic closure of this list includes the term

$$Set.valid[q^{-1}[q[st]]]$$

since the scope includes the dependency

$$\textbf{depends}\ Set.valid[st]\ \textbf{on}\ Seq.valid[q[st]]$$

This extra term in the closure weakens the modification constraint for $Set.valid$. With the downward closure, the constraint was

$$\langle \forall\, stv \ :: \ Set.valid[stv] = Set.valid'[stv]\ \rangle$$

However, with the dynamic closure, the constraint is

$$\langle \forall\, stv \ :: \ Set.valid[stv] = Set.valid'[stv]\ \lor\ stv = q^{-1}[q[st]]\ \rangle$$

which when map inverses are eliminated becomes

$$\langle \forall\, stv \ :: \ Set.valid[stv] = Set.valid'[stv]\ \lor\ q[stv] = q[st]\ \rangle$$

which in turn is functionalized to

$$\begin{aligned}\langle \forall\, stv \ :: \ & \mathcal{F}.Set.valid(sres, res.Set.valid, q, \\ & \qquad \mathcal{F}.Seq.valid(sres, res.Seq.valid))[stv] \\ & = \\ & \mathcal{F}.Set.valid(sres', res.Set.valid', q', \\ & \qquad \mathcal{F}.Seq.valid(sres', res.Seq.valid'))[stv] \\ \lor\ & q[stv] = q[st]\ \rangle\end{aligned}$$

which eliminates the problem since the disjunct $q[stv] = q[st]$ allows the method to change the validity of st. (The disjunct also allows the method to change the validity of any other set whose q field coincides with the q field of st. This accurately reflects the semantics of the situation, and we take it as evidence that our rewriting is appropriate. It is a different issue whether the designer of $Set.T$ should allow such sharing of the q field—probably not, as explained in Section 9.3.)

Here is an example to show why the dynamic closure is the union of two closures, rather than, for example, the upward closure of the downward closure or some kind of bi-directional closure. Suppose that we were doing full functional verification instead of extended static checking only, and that sets were represented by sequences without duplicates. Then we would have the dependency

depends $Set.valid[st]$ **on** $Seq.state[q[st]]$

since the **rep** declaration for $Set.valid[st]$ would forbid duplicates in $q[st]$, which is an assertion about $Seq.state[q[st]]$. In addition, we still have the dependency

depends $Set.state[st]$ **on** $Seq.state[q[st]]$

If the dynamic closure were the upward closure of the downward closure, then the dynamic closure of the modifies list

modifies $Set.state[st]$

would include $Set.valid[st]$. Thus, in the scope of the implementation, any operation that changes the state of a set would be allowed also to modify its validity, which would be preposterous. (It would also be unsound, since in the scope of a client of the Set interface, such a side effect would be unexpected.)

Finally, here is an example to show why the dynamic closure contains the full upward closure of the flexible terms of a modifies list, rather than a single level. Suppose R is a subtype of $Rd.T$ (see Section 4), that

var $rq: R \to Seq.T$

is a sequence-valued field of readers of type R, and that the subtype-specific validity of R readers depends on the validity of the associated sequence:

depends $svalid[r: R]$ **on** $Seq.valid[rq[r]]$

In this scenario, we would argue that a call that modifies $Seq.valid[rq[r]]$ should be allowed to modify both $svalid[r]$ and $Rd.valid[r]$.

Dependency segregation. Our desugaring of modifies lists requires a restriction, which we call the *dependency segregation restriction*: no field c occurs both in a static dependency of the form $a[t]$ on $c[t]$ and in a dynamic dependency of the form $z[s]$ on $c[b[s]]$. Because of the visibility and top-down requirements, this restriction can easily be enforced modularly.

To see that this restriction is necessary, consider the following example:

unit A
$\vdots$
depends $a[t]$ **on** $c[t]$
proc $P(t)$ **modifies** $c[t]$
unit B **import** A
$\vdots$
depends $z[s]$ **on** $c[b[s]]$
... call $t.P()$ ...

Because $c[t]$ is not in the flexible subset of the modifies list of P, the caller in B expects the value of $z[b^{-1}[t]]$ to be unchanged. However, if the implementation of P is placed in unit A (or in any unit where the dynamic dependency is not visible), then no modification constraint will be added to the implementation to enforce the unchangedness of z.

The dependency segregation restriction does not seem to rule out any useful programs.

7.2 Modularity requirements for dynamic dependencies

The visibility and top-down requirements that we impose for static dependencies both have analogues for dynamic dependencies, but the analogues are significantly different from the originals. One of the differences is that because pivot fields can be updated dynamically, several of the requirements for dynamic dependencies can be checked only by reasoning about specifications, not by a simple check on the placement of declarations. Our checker enforces the requirements of this sort by transforming an annotated input program into another annotated program that will verify exactly when the input program would verify and the input program obeys the requirements.

Before getting into these deep waters, we describe the one modularity requirement for dynamic dependencies that can be checked simply by looking at the placement of declarations.

Pivot visibility requirement. The *pivot visibility requirement* requires that a dynamic dependency

depends $a[t]$ **on** $c[b[t]]$

be visible anywhere b is.

This can be enforced simply by checking that the dependency is placed in the same unit as the declaration of b.

Here is the reason we impose the requirement. If there were a scope where a and b are visible but the dependency is not, then a modification to $b[t]$ could change $a[t]$ unexpectedly. The requirement is not burdensome, since the module that implements the abstraction a usually declares both the pivot field and the dependency.

It would probably be sound to require only that the dependency be visible where both a and b are, but we have not found any examples where the extra flexibility of this weaker requirement would be of any engineering use.

Absence of abstract aliasing. The visibility requirement for static dependencies prevents unexpected side effects between an abstract variable and its representation. For the dynamic dependency of $a[t]$ on $c[b[t]]$, the pivot visibility requirement prevents unexpected side effects between a and b, but we still need to protect against unexpected side effects between a and c. This is the function of *absence of abstract aliasing*.

For static dependencies, the problem is solved by requiring the dependency to be visible anywhere both a and c are, but for dynamic dependencies, this would be undesirably strict. For example, consider the sets and sequences described earlier. This strict version of the requirement would force the dependency (and therefore the pivot field as well) to be declared in the public interface *Set* instead of in the private implementation where they belong. (It is obviously unreasonable to place the pivot and dependency declarations in the public interface *Seq*, since sets may not have been envisioned when sequences were defined.)

To find the right modularity requirement, we focus on the situation that goes wrong, and use our judgment as programmers to assign blame. The situation that goes wrong is an unexpected side effect between $a[t]$ and $c[u]$ for some values t and u. For the side effect to happen, it must be that $b[t] = u$. For the side effect to be unexpected, it must occur in a scope where a and c are visible but the dependency is not. Because of the pivot visibility requirement, it must therefore be that b is not visible either.

More formally, we say that *abstract aliasing* occurs if execution reaches some point in the program text where, for some expressions E and F and pivot field b, all free variables of E and F are visible, b is not visible, and $E = b[F] \;\wedge\; E \neq$ **nil**. Notice that the condition $E = b[F]$ makes sense even outside the scope of b, since b's value exists even at program points where b is not visible. We require that programs be designed so that abstract aliasing does not occur. This requirement, together with the pivot visibility requirement, is the analogue for dynamic dependencies of the visibility requirement for static dependencies.

So much for the definition of abstract aliasing. A further question is to find a static discipline for avoiding the problem.

One simple discipline that prevents abstract aliasing would be to forbid communicating a pivot value $b[t]$ into or out of the scope declaring b. All forms of communication must be forbidden, including communication via procedure parameters, procedure results, and global and heap locations. We say that the value of a pivot field transferred into or out of the scope of the field's declaration is *leaked*.

Unfortunately, the simple discipline of forbidding all leaking is too strict, for several reasons. For example, initialization methods occasionally take parameters that are stored into pivot fields. Also, methods of container classes must return the elements of the container. Most compellingly, to operate on a pivot, an implementation of an abstraction must pass the pivot value to the pivot's own methods.

We have defined a more flexible discipline for avoiding leaking, which solves the three problems mentioned in the previous paragraph. But our solution is not totally satisfactory, and instead of describing it in this paper, we refer the reader to the companion paper *Wrestling with rep exposure* [7].

Disjoint ranges requirement. The *disjoint ranges requirement* states that pivot fields declared in distinct units have disjoint ranges. That is, if b and d are pivot fields whose declarations occur in different units, then, at any procedure boundary,

$$\langle \forall s, t :: b[s] = d[t] \Rightarrow b[s] = \textbf{nil} \rangle$$

where s and t range over non-**nil** objects. The requirement is enforced by rewriting pre- and postconditions.

To motivate the disjoint ranges requirement, we first recall the motivation for the top-down requirement for static dependencies. In the presence of the dependencies

depends $a[t]$ **on** $c[t]$
depends $v[t]$ **on** $c[t]$

the checker protects related abstractions by adding the postcondition

$$v[t] = v'[t]$$

to any procedure specified with **modifies** $a[t]$. If the dependency of $v[t]$ on $c[t]$ were not visible, the checker would be unable to add this postcondition, making modular verification unsound. Soundness is achieved for static dependencies by imposing the top-down requirement.

The top-down requirement works for static dependencies, but it would be ridiculously strict to generalize it in the obvious way for dynamic dependencies. For example, it would be too strict to require that *Set.valid* be visible wherever *Seq.valid* is, since *Set* is a higher-level abstraction, which quite possibly was not envisioned when *Seq* was designed.

The disjoint ranges requirement is the analogue of the top-down requirement, but for dynamic instead of static dependencies. Consider the following variables and dependencies:

> **depends** $a[t]$ **on** $c[b[t]]$
> **depends** $v[t]$ **on** $c[d[t]]$

and a procedure P specified with **modifies** $a[t]$. Then P is allowed to modify $a[t]$ and $c[b[t]]$, but not $v[s]$ for any s, not even when $d[s] = b[t]$. If d is visible in the scope containing P's body, then modifies list desugaring adds an appropriate conjunct to the postcondition. But if d is not visible in that scope, the only way to guarantee that $v[s]$ is unchanged is to guarantee $d[s] \neq b[t]$, which is ensured by the disjoint ranges requirement.

Swinging pivots restriction. Consider the modifies list

> **modifies** $b[t]$

It gives a procedure the license to modify b, but only at t. More precisely, the procedure is required to establish the postcondition

$$\langle \forall s :: b_0[s] = b'[s] \ \vee \ s = t \rangle$$

where, in this discussion, we write b' for the final value of b and b_0 for the initial value of b.

Now consider the modifies list

> **modifies** $b[t], c[b[t]]$

It, too, gives a procedure the license to modify b, only at t. In addition, it allows the modification of c at one point only. But is this point $b_0[t]$ or $b'[t]$? It is traditional to choose the first alternative, allowing the modification of c only at $b_0[t]$, and we follow this tradition. However, the possibility that $b_0[t]$ may be different from $b'[t]$ causes a difficulty, which we will now describe.

Consider the following artificial procedure that returns from a reader not the current character but the second character:

> **proc** *secondChar*(*rd*: *Rd*.*T*): **int**
>   **requires** *valid*[*rd*]
>   **modifies** *state*[*rd*]
> **impl** *secondChar*(*rd*: *Rd*.*T*) **is**
>   **result** := *rd*.*getChar*() ;
>   **result** := *rd*.*getChar*()
>   **end**

We certainly hope that this implementation will verify: since *rd*.*getChar*()'s specification requires *valid*[*rd*] and modifies *state*[*rd*], that same specification should be satisfied by two calls in a row. Indeed, it does verify in a scope where only *Rd* is imported.

Unfortunately, the implementation does not verify if *RdRep* is imported! The problem is that, in the presence of the dependencies declared in *RdRep*, the checker will issue a warning because of the possibility that the first call to *getChar* changes *buff*[*rd*] and the second call changes the contents of the new *buff*[*rd*]. Thus the net effect is inconsistent with our interpretation of *secondChar*'s specification

> **modifies** *state*[*rd*]

since this desugars to

> **modifies** *buff*[*rd*], *elems*[*buff*[*rd*]], . . .

which allows changing $elems[buff_0[rd]]$, but not $elems[buff'[rd]]$.

Reversing the tradition does not help: if *secondChar*'s specification were desugared to allow modification of *elems* at $buff'[rd]$ instead of at $buff_0[rd]$, then the checker would warn about the possibility that the first call to *getChar* changes the contents of the buffer and the second call changes the buffer pointer.

This problem is a failure of modular soundness, which can only be rectified in two ways: by changing the proof system so that *secondChar* does not verify when only *Rd* is imported, or by changing it so that *secondChar* does verify when *RdRep* is imported. Our engineering judgment is that the latter course is the right one. The best way we have found to achieve this is to impose a rather strict requirement that we call the *swinging pivots restriction*: a procedure specified to modify a pivot field is allowed to change it only to **nil** or to a value newly allocated within the procedure. This discipline is enforced formally by adding, for each pivot field b, a conjunct to the postcondition of every procedure:

$$\langle \forall s :: b_0[s] = b'[s] \ \vee \ b'[s] = \textbf{nil} \ \vee \ \neg alloc_0[b'[s]] \rangle \tag{17}$$

where $alloc_0[x]$ means the object x was allocated in the pre-state (Section 8.1 provides more details about $alloc$). With this postcondition of $getChar$, the spurious warnings will not occur, since the problematic control paths are inconsistent with the strengthened postcondition.

As we have described it, the swinging pivots restriction is too strict. For example, the restriction forbids an initialization method from assigning one of its parameters to a pivot field in the object being initialized, which is occasionally necessary. It is straightforward to revise the swinging pivots restriction to accommodate such assignments, by adding a disjunct to (17), but we won't describe it in this paper since it requires the nomenclature defined in *Wrestling with rep exposure* [7].

We can envision situations where even the revised restriction is too strict, for example, a double-buffered reader implementation in which one buffer is being filled while the other is being emptied. But the swinging pivots restriction is the best solution to the problem that we know.

8 Reasoning about types and allocation

A central issue described in this paper is the rewriting of specifications found in a modular program into specifications about which one can reason using standard techniques for verifying one-scope programs. Although those techniques have been described widely in the literature, there are some areas where we have had to innovate in order to build the Modula-3 Extended Static Checker, in particular in the areas of reasoning about types and allocation. We describe these techniques here, both because some of the techniques related to allocation are new, and because this material interacts with the modularity issues discussed in Section 9.

8.0 Reasoning about types

Conditions that the type system guarantees can be assumed by the checker without proof. We call such conditions "freeconditions". For example, in checking a procedure implementation like

$$\textbf{impl}\ P(n\colon \textbf{nat})\ \textbf{is}\ \ldots\ \textbf{end}$$

we get the free precondition $n \geq 0$. The full story is more complicated, since the value of type **nat** might be a field of some object rather than a simple parameter.

Therefore, every verification condition R in a scope D is discharged under the *background predicate* for D:

$$\mathit{BackgroundPred}_D \ \Rightarrow\ R$$

The background predicate is a conjunction of axioms formed from the declarations that are visible. This subsection gives a flavor of what the background predicate contains.

For every type T, the background predicate contains the definition of a predicate symbol $is\$T$, which asserts that its argument is of type T. In addition, for each object type T, the background predicate contains a constant $tc\$T$ representing the *typecode* of T. If T is declared to be a subtype of an object type U, the background predicate will contain the conjunct

$$\mathit{subtype}1(tc\$T, tc\$U)$$

For an object type T, $is\$T$ is defined by the conjunct

$$\langle \forall t :: \ is\$T(t) \ \equiv\ t = \textbf{nil} \ \vee\ \mathit{subtype}(\mathit{typecode}(t), tc\$T)\ \rangle$$

where $\mathit{subtype}$ is the reflexive, transitive closure of $\mathit{subtype}1$.

Data fields are treated as maps from objects to values. For every map type $T \rightarrow U$ occurring in the program, the background predicate defines a predicate symbol $\mathit{field}\$T\$U$. One of the axioms about $\mathit{field}\$T\$U$ asserts that applying a map to a value in its domain produces a value in its range:

$$\langle \forall f, t :: \ \mathit{field}\$T\$U(f) \ \wedge\ is\$T(t) \ \wedge\ t \neq \textbf{nil} \ \Rightarrow\ is\$U(f[t])\ \rangle$$

For the full details of the background predicate of a small object-oriented language, we refer the reader to the axiomatic semantics of Ecstatic [29]. The background predicate used for Modula-3 in the Extended Static Checker is similar, but more complicated, because, for example, Modula-3 has a larger variety of types.

8.1 Reasoning about allocation

Consider the following specification puzzle: A procedure P takes a filename as a parameter, opens the named file, reads four bytes, and returns their value as an integer. We would like to specify P with an empty modifies list, since P is essentially functional from the point of view of the client. However, it is impossible to implement P without side effects on allocated data. For example, if a file reader is used, its buffer will be changed.

Our solution is to make it implicit in the specification of every procedure that modifications to newly allocated state are allowed. Thus, although P's modifies list is empty, its implementation is allowed to change the fields of the file reader, since it allocates that reader (but if P used a pre-existing reader rd, it would have to mention $state[rd]$ in the modifies list, as usual). We say that by convention we allow "free modification of unused state". In fact, we have already used this convention: *BlankRd.init* modifies the contents of the buffer. This is allowed by our convention, because the buffer is newly allocated, but it would have been inconsistent with the modifies list otherwise.

We believe this convention is sound with respect to the standard operational semantics, but we have neither proved it nor noticed that anyone else has.

The convention affects the desugaring of specifications. To describe this in more detail, we must explain the semantics of allocation. Since successive calls to the storage allocator return different results, it must be that the calls have some side effect. Informally, the side effect is to extend the set of allocated objects. In the formal semantics, the side effect is to change the "allocated" property of the returned object from *false* to *true*. We model this property with the predeclared boolean object field *alloc*.

The program expression $\mathbf{new}(T)$ is sugar for

$$
\begin{array}{l}
\mathbf{var}\ x\ \mathbf{in} \\
\quad x \neq \mathbf{nil} \ \wedge\ \neg alloc[x] \ \wedge\ x \in T \\
\rightarrow \\
\quad alloc[x] := true\ ;\ \mathbf{result} := x \\
\mathbf{end}
\end{array}
$$

that is, nondeterministically choosing any non-**nil**, unallocated object of type T, and allocating and returning it.

The modifies list of every procedure implicitly contains *alloc*, and the postcondition of every procedure implicitly includes

$$\langle \forall s :: alloc[s] \Rightarrow alloc'[s] \rangle$$

that is, the procedure can allocate objects, but not deallocate them (we assume the usual fiction of garbage collected languages wherein objects are allocated but never deallocated).

Recall that, for every field g, a modifies list desugars to a conjunct in the postcondition of the form

$$\langle \forall s :: \ g[s] = g'[s] \ \vee \ s = E_0 \ \vee \ s = E_1 \ \vee \ \ldots \rangle$$

where the E's are the modification points allowed for g by the modifies list. With our allocation convention, this conjunct becomes

$$\langle \forall s :: \ g[s] = g'[s] \ \vee \ \neg alloc[s] \ \vee \ s = E_0 \ \vee \ s = E_1 \ \vee \ \ldots \rangle$$

This allows the procedure to modify g at any newly allocated object.

The specification language admits assertions that quantify over all objects of a particular type. Such assertions are considered by convention to apply to allocated objects only. For example, a universal quantification $\langle \forall x{:}\,T :: \ P(x) \ \rangle$ occurring in a specification is desugared into

$$\langle \forall x{:}\,T :: \ alloc[x] \ \Rightarrow \ P(x) \ \rangle$$

except if it occurs in a postcondition, in which case it is desugared into

$$\langle \forall x{:}\,T :: \ alloc'[x] \ \Rightarrow \ P(x) \ \rangle$$

These kinds of assertions are not common in pre- or postconditions, but they are common in program invariants, which will be discussed in Section 9.3.

Unlike the mini-language used in this paper, many programming languages allow declarations to specify *default values* for object fields. These will become important when we discuss program invariants in Section 9.3. Taking default values into account, the desugaring of **new** must be altered slightly from the version given above. Suppose, for example, that f is one of T's fields, and that the default value of f is the constant C. Then $\textbf{new}(T)$ is sugar for

var x **in**
  $x \neq \textbf{nil} \ \wedge \ \neg alloc[x] \ \wedge \ x \in T \ \wedge \ f[x] = C$
$\rightarrow$
  $alloc[x] := true$; $\textbf{result} := x$
end

This desugaring nondeterministically chooses an object whose f field has the right value. (We prefer this to an alternative desugaring which assigns $f[x] := C$

after choosing x. Our version reduces the number of assignments, which speeds mechanical checking.)

The story we have told so far about **new** is not new. For example, our story is essentially equivalent to that given by Hoare and Wirth in their classic paper on an axiomatic semantics for Pascal [19]. We were surprised to find, when applying our checker to the Modula-3 library, that the story doesn't work. The following artificial program illustrates the problem:

> **type** T, U
> **var** $f: T \to U$
> **proc** $P(t: T)$ **requires** $t \neq$ **nil**
> **impl** $P(t: T)$ **is**
>   **var** $u: U$ **in**
> $u :=$ **new**(U) ;
>   **assert** $f[t] \neq u$
>   **end**
> **end**

Procedure P, which takes an object t as a parameter and allocates a new object u, will crash if the f field of t is u. As programmers, we know this won't ever happen, but nothing we have said so far allows this procedure to be verified. We have ensured that **new** returns a previously unallocated object, but we have not ensured that all reachable objects are allocated. This problem seems to be less appreciated than the more easily solved problem of ensuring that **new** returns a previously unallocated object.

The background predicate helps, since we can arrange that it provide the assumption $alloc[t]$ for each parameter or global variable of an object type. But as the example shows, this is not sufficient, since $alloc[f[t]]$ does not follow logically from $alloc[t]$. The basic idea of our solution is to allow the checker to assume that fields of allocated objects are themselves allocated, that is, that for every declared field f whose range type is an object type, $alloc$ is closed under f. It is not enough to assume this condition once and for all in the background predicate, since both $alloc$ and f are mutable. Instead, the closure condition is an implicit pre- and postcondition of every procedure, including **new**. We will not describe the details here, since they are not particularly relevant to modular verification. Instead, we refer interested readers to the axiomatic semantics of Ecstatic [29].

9 Further challenges

Static and dynamic dependencies allow us to check many parts of the Modula-3 run-time library that we were unable to check without them. But there remain programming paradigms that are used in practice and seem sound and modular to which our approach does not apply. This section describes some of these challenges and some tentative ideas we have for addressing them.

9.0 Cyclic dependencies

Dynamic dependencies give rise to the possibility of *cyclic dependencies*, that is, an abstract variable may depend on itself indirectly, via some pivot fields. Indeed, this happens in the case of a "filter" object that "forwards" method calls to an instance of one of its supertypes. For example, consider a $DOSRd$ subtype of Rd that returns all the characters of a given child reader, but with carriage return characters filtered out:

$$
\begin{array}{l}
\textbf{unit } \mathit{DOSRd} \textbf{ import } \mathit{Rd} \\
\quad \textbf{type } T <: \mathit{Rd}.T \\
\quad \textbf{proc } \mathit{init}(\mathit{drd}\colon T,\ \mathit{rd}\colon \mathit{Rd}.T)\colon T \\
\qquad \textbf{requires } \mathit{valid}[\mathit{rd}] \\
\qquad \textbf{modifies } \mathit{valid}[\mathit{drd}] \\
\qquad \textbf{ensures } \mathit{valid}'[\mathit{drd}] \ \wedge\ \textbf{result} = \mathit{drd}
\end{array}
$$

(For simplicity, we're ignoring $state$.) The expression $\textbf{new}(\mathit{DOSRd}.T).\mathit{init}(\mathit{rd})$ allocates, initializes, and returns a new DOS reader with child reader rd. The implementation of DOS readers will need to store the child reader in some field of the DOS reader, say ch:

$$\textbf{var } \mathit{ch}\colon \mathit{DOSRd}.T \to \mathit{Rd}.T$$

The implementation will also have to give the representation of $svalid$ for DOS readers, which will include a conjunct expressing that the child is valid:

$$\textbf{rep } \mathit{svalid}[\mathit{drd}\colon \mathit{DOSRd}.T] \ \equiv\ \ldots\ \wedge\ \mathit{valid}[\mathit{ch}[\mathit{drd}]]$$

This requires the dynamic dependency

$$\textbf{depends } \mathit{svalid}[\mathit{drd}\colon \mathit{DOSRd}.T] \textbf{ on } \mathit{valid}[\mathit{ch}[\mathit{drd}]]$$

Combined with the static dependency of $valid[rd]$ on $svalid[rd]$ in $RdRep$, this produces a cycle of dependencies.

To accommodate cyclic dependencies, we make two changes to our proof system. We will describe the two changes for the case that there is exactly one pivot field involved in any cycle. This is the only case that we have implemented in ESC, although we believe that the ideas could be generalized.

The first change is in taking the closure of a modifies list. We need to make some change to prevent the closure from being infinite. We introduce two new notations allowed in closures: $f^*[t]$ and $f^{-*}[t]$. Intuitively, they represent the set of terms

$$t,\ f[t],\ f[f[t]],\ \ldots$$

and the set of terms

$$t,\ f[^{-1}t],\ f[^{-1}f[^{-1}t]],\ \ldots$$

respectively. These notations appear in the closures of modifies list, but they are fictions that are eliminated when the closures are transformed into postconditions. Since we assume only one pivot field per cycle, the infinite set of terms produced by the closure rules described previously can be summarized in a finite set of terms involving the new notations. For example, in the context of the implementation of DOS readers, the modifies list

modifies $valid[drd]$

has the closure

$$valid[ch^*[drd]],\ svalid[ch^*[drd]],$$
$$valid[ch^{-*}[drd]],\ svalid[ch^{-*}[ch^{-1}[drd]]]$$

Recall that modifies lists are closed, and then closed modifies lists are turned into modification constraints in postconditions. Thus, to eliminate our new notations, we must show how to rewrite them into modification constraints. The license to modify $a[b^*[t]]$ gives rise to the postcondition contribution

$$\langle \forall s :: a[s] = a'[s] \ \vee\ t \xrightarrow[\mathbf{nil}]{b} s \rangle$$

where the notation $t \xrightarrow[x]{b} s$, read "t reaches s via (applications of) b, not going through x", is defined by Nelson [42]. Similarly, the license to modify $a[b^{-*}[t]]$ gives rise to the postcondition contribution

$$\langle \forall s :: a[s] = a'[s] \ \vee\ s \xrightarrow[\mathbf{nil}]{b} t \rangle$$

The second change to our proof system is to the pointwise axiom for any abstract variable involved in a cycle of dependencies. We will describe the change by means of an example. To set the stage, we consider first an example with a dynamic but non-cyclic dependency, say

depends $a[t]$ **on** $e[t]$
depends $a[t]$ **on** $c[b[t]]$

The pointwise axiom for a (leaving out residues) is

$$\begin{aligned}\langle \forall\, s, e0, e1, c0, c1, b0, b1 \;::\;\; & e0[s] = e1[s] \;\;\wedge\;\; c0[b0[s]] = c1[b1[s]] \\ & \Rightarrow\;\; \mathcal{F}.a(e0, c0, b0)[s] = \mathcal{F}.a(e1, c1, b1)[s] \;\rangle\end{aligned}$$

Now let the dynamic dependency be cyclic:

depends $a[t]$ **on** $e[t]$
depends $a[t]$ **on** $a[b[t]]$

The new pointwise axiom for a (leaving out residues) is

$$\begin{aligned}&\langle \forall\, s, e0, e1, b0, b1 \;:: \\ &\qquad \langle \forall\, r \;::\; s \xrightarrow[\mathbf{nil}]{b0} r \;\Rightarrow\; e0[r] = e1[r] \;\;\wedge\;\; b0[r] = b1[r] \;\rangle \\ &\qquad \Rightarrow\;\; \mathcal{F}.a(e0, b0)[s] = \mathcal{F}.a(e1, b1)[s] \;\rangle\end{aligned}$$

That is, $a[t]$'s value depends only on the e and b fields of objects reachable from t via b.

We will illustrate this pointwise axiom by showing the verification of the *init* method of DOS readers, implemented as:

impl $init(drd\colon T,\;\; rd\colon Rd.T)\colon T$ **is**
  $ch[drd] := rd$; $lo[drd] := 0$; $\ldots$; **result** $:= drd$
end

where we assume the elided code initializes the *cur*, *hi*, and *buff* fields of *drd* to satisfy the validity requirements given in *RdRep*. The first part of this verification is showing that the assignment to the *ch* field establishes $svalid[drd]$. This is easy since the *init* method requires $valid[rd]$ as a precondition. The second part is showing that the assignment does not affect the validity of any other reader (except as allowed by the modifies list). As we have already remarked, the closure of the modifies list includes

$valid[ch^{*}[drd]],\;\; valid[ch^{-*}[drd]]$

which produces the postcondition

$$\langle \forall s :: \; valid[s] = valid'[s] \;\vee\; drd \xrightarrow[\textbf{nil}]{ch} s \;\vee\; s \xrightarrow[\textbf{nil}]{ch} drd \;\rangle$$

which is functionalized to

$$\begin{array}{l} \langle \forall s :: \; \mathcal{F}.valid(ch, lo, \ldots)[s] = \mathcal{F}.valid(ch', lo', \ldots)[s] \\ \qquad \vee\; drd \xrightarrow[\textbf{nil}]{ch} s \;\vee\; s \xrightarrow[\textbf{nil}]{ch} drd \;\rangle \end{array}$$

which follows from the pointwise axiom for *valid*, which is

$$\begin{array}{l} \langle \forall s, ch0, ch1, lo0, lo1 :: \\ \quad \langle \forall r :: \; s \xrightarrow[\textbf{nil}]{ch0} r \;\Rightarrow\; ch0[r] = ch1[r] \;\wedge\; lo0[r] = lo1[r] \;\wedge\; \ldots \rangle \\ \quad \Rightarrow \\ \quad \mathcal{F}.valid(ch0, lo0, \ldots)[s] = \mathcal{F}.valid(ch1, lo1, \ldots)[s] \;\rangle \end{array}$$

We leave the proof to the reader.

In the verification of the *init* method of DOS readers, no properties of the reachability predicate were used: it might as well have been an uninterpreted predicate. Properties of the reachability predicate come into play when verifying a non-trivial operation on the DOS reader whose implementation modifies the child reader (for example the *refill* method, which recursively invokes the *refill* method of the child).

In summary, we have described the essential ideas of a proof system for cyclic dependencies. More details are described by Rajeev Joshi [24]. At least two problems still remain: Cyclic dependencies with more than one pivot field per cycle require some generalization. Also, even with just one pivot field per cycle, our rewriting produces verification conditions that are beyond the limit of what our automatic theorem prover can handle efficiently.

9.1 Yet more dependencies

We have concentrated on static and dynamic dependencies because they play a central rôle in the patterns of abstractions in the library programs we took as test cases in the ESC project, not because we can't imagine other kinds of dependencies. In this section, we sketch what we know about other dependencies.

If a global abstract variable (not a field) depends on a global concrete variable (not a field), we call it an *entire dependency*. For example,

spec var k: **int**
var m, n: **nat**
rep $k \equiv m - n$
depends k **on** m, n

This kind of abstraction occurs frequently in papers on data refinement, but in practice we have found static and dynamic dependencies far more frequent. One place in which entire dependencies are useful is in reasoning about module initialization, which we will address in Section 9.2. We have a soundness theorem for entire dependencies, and the modularity requirements are essentially the same as those for static dependencies, that is, the dependency of a on c must be placed in the unit that declares c [28].

If an abstract field (not a global variable) depends on a global concrete variable (not a field), we have a dependency of the form

depends $a[t]$ **on** g

As an example of how this might come up, consider an abstract type whose instances contain unique id fields. Each id field is initialized from a global counter, $gcount$. This might well lead to a representation of validity of the form

rep $valid[t] \equiv \ldots \wedge id[t] < gcount$

which in turn would require a dependency of the form

depends $valid[t]$ **on** $gcount$

However, the soundness of these dependencies is problematical, and our current view is that they are not useful and should be forbidden. To specify a data type containing unique identifiers, we recommend using program invariants, as will be described in Section 9.3.

If an abstract field depends on concrete fields of the elements of an array, we have an *array dependency*. We would suggest overloading the notation for dynamic dependencies: if $b[t]$ has type **array**$[U]$ and c is a field with index type U, then

depends $a[t]$ **on** $c[b[t]]$

is an array dependency that allows $a[t]$ to depend on the sequence of values

$$c[b[t][0]],\ c[b[t][1]],\ \ldots$$

So an array dependency seems akin to a dynamic dependency, but with an array of pivots instead of just one.

As an example of an array dependency, consider a type T representing sets of elements of type E. Suppose that the implementation automatically enlarges itself when necessary, and that enlarging requires rehashing the current elements, and that rehashing an element requires that the element be valid. Then, the validity of the set will require the validity of all its elements. If the elements are held in an array, say b, then the validity of the set will have the form

$$\textbf{rep } T.valid[t\colon T] \equiv$$
$$\ldots \wedge \langle \forall i :: 0 \le i < \textbf{number}(b[t]) \Rightarrow E.valid[b[t][i]] \rangle$$

which involves the array dependency

$$\textbf{depends } T.valid[t\colon T] \textbf{ on } E.valid[b[t]]$$

We suspect that array dependencies are a straightforward generalization of dynamic dependencies, but we have not investigated them thoroughly.

One can imagine many other kinds of dependencies, for example,

$$\textbf{depends } a[t] \textbf{ on } c[b[d[t]]]$$

But we have never been able to make a strong case that such dependencies are useful.

9.2 Checking initialization order

Initializing global data is more complicated in a multi-module program than in a single-module program and is a common source of programming errors. Some of the procedures in a module require that the module's globals be initialized, but generally not all of them: for example, any procedure that is used in performing the initialization. Thus, there are two classes of procedures: those that require prior initialization of the module and those that don't. A common error is to inadvertently call a procedure of the first class before initialization is complete, either through confusion over which class a procedure is in, or because the linker initializes the modules in an unexpected order.

We suggest that abstraction and specification can help in solving this problem. The idea is to introduce into the interface of each module a boolean abstract variable, called an *init variable*, which means the module has been initialized. Procedures of the first class require the init variable as a precondition, while those of the second class do not. The purpose of a module body is to ensure an init variable as a postcondition; to achieve this, it may call other procedures that modify and ensure the init variable.

A programmer can also require one or more init variables of other modules as preconditions of the module body. The linker calls the module bodies in an order such that each body's precondition is established before it is called, or reports a cycle if this is impossible. Each module provides a **rep** declaration that connects its init variable to the globals of the module, so occurrences of init variables in specifications are desugared like any other abstract variable.

An init variable generally depends on the global variables in the module. These dependencies satisfy the modularity requirements for entire dependencies since they are placed in the same units as the declarations of the globals, and thus present no problem to modular verification. An init variable may also depend on other init variables, since

$$\mathbf{rep}\ \mathit{Minit} \ \equiv\ \mathit{Ninit} \ \wedge\ \ldots$$

where Minit and Ninit are the init variables of two modules M and N, is a simple way of giving M's procedures of the first class the right to call N's procedures of the first class. Unfortunately, the dependency of Minit on Ninit is most naturally placed in the implementation of module M, a unit that declares neither Minit nor Ninit. Thus, this dependency violates both the visibility and top-down requirements, which in general destroys soundness. We have several ideas for restoring soundness while allowing init variables to depend on one another. These ideas are based on the observation that init variables change only from *false* to *true*. But we have not proved a soundness theorem.

9.3 Invariants

In practice, almost all pivot fields are injective (one-to-one), that is, if b is a pivot field and u and v are distinct objects in the domain of b, then $b[u]$ and $b[v]$ are distinct (or they are both **nil**). The reason for this is easily seen by considering the prototypical example involving a dynamic dependency, shown in Figure 10. The call to R from P modifies $c[b[t]]$. This affects the value of $a[t]$. If the pivot field

unit *M*
  type *T*
  spec var $a\colon T \to \ldots$
  $\vdots$
  proc $P(t\colon T)$ **modifies** $a[t]$

unit *N*
  type *U*
  spec var $c\colon U \to \ldots$
  $\vdots$
  proc $R(u\colon U)$ **modifies** $c[u]$

unit *MImpl* **import** *M*, *N*
  var $b\colon T \to U$
  depends $a[t\colon T]$ **on** $c[b[t]]$
  $\vdots$
  impl $P(t\colon T)$ **is** $\ldots R(b[t]) \ldots$ **end**

Figure 10: A prototypical example involving a dynamic dependency.

b were not injective, it would also affect $a[u]$ for any u such that $b[u] = b[t]$. In general, when $a[t]$ is modified by changing part of its representation $c[b[t]]$, the only hope for showing that the modification obeys the modifies list

$$\textbf{modifies}\ a[t]$$

is to require the injectivity of b.

Note that although we find injectivity necessary to be able to verify interesting programs, we have not found injectivity to be a requirement for soundness.

By the way, it is surprisingly difficult to verify a procedure that initializes an injective field. While showing that a command like

$$b[t] := \textbf{new}(U)$$

maintains the injectivity of b is easy, a command like

$$b[t] := \mathit{NewU}()$$

does not verify, even if procedure NewU is specified to ensure $\neg \mathit{alloc}[\textbf{result}] \wedge \mathit{alloc}'[\textbf{result}]$. The checker dreams up the possibility that NewU allocates a new U object, squirrels it away into some b field, and then returns it. To cope with this problem, we enrich the specification language with the expression $\mathit{virgin}[x]$, which means that x is not, and has never been, the value of any object field or global variable. The details are found in a paper by Leino and Stata [34].

How should a programmer use the specification language to record the design decision that a field is to be injective? One might first try to include this as part of the representation of an object's validity, producing a **rep** declaration like

$$\textbf{rep}\ \mathit{valid}[t\colon T] \equiv \ \ldots \ \wedge \ (b[t] = \textbf{nil} \vee \langle \forall s\colon T \ :: \ s \neq t \ \Rightarrow \ b[s] \neq b[t] \ \rangle)$$

But this seems problematical. It makes $\mathit{valid}[t]$ depend not just on $b[t]$, but on $b[s]$ for all s of the appropriate type. It seems perverse to think of this unbounded collection of $b[s]$'s to be part of the "representation" of $\mathit{valid}[t]$.

A simpler and better strategy is to extend the specification language with the notion of a *program invariant*: a declaration of the form

$$\textbf{invariant}\ J$$

records the intention that the predicate J hold at every procedure call and return. For example, to specify the injectivity of b, the following program invariant can be used:

$$\begin{aligned}\textbf{invariant}\ \langle \forall t,\ u\colon T \ :: \ & t \neq \textbf{nil} \ \wedge \ u \neq \textbf{nil} \ \wedge \ t \neq u \\ & \Rightarrow \ b[t] \neq b[u] \ \vee \ b[t] = \textbf{nil} \ \rangle\end{aligned}$$

The checker enforces program invariants with two checks. First, it checks that J is true at the "beginning of time". Second, it checks that every procedure respects J (assuming that all the procedures it calls respect J), that is, it conjoins J to the pre- and postcondition of every procedure implementation and procedure call.

The beginning-of-time test is straightforward and presents no modularity problems. It consists of the following proof obligation for each declared program invariant J:

$$\langle \forall t :: t = \mathbf{nil} \ \vee \ \neg alloc[t] \rangle \ \Rightarrow \ J$$

That is, J must hold in a state in which no non-**nil** objects have been allocated. More precisely, this proof obligation must follow from the background predicate. If J contains free variables of primitive types like integers, then it must hold regardless of their values. To enforce invariants about global variables, the initvars technique described in Section 9.2 is more useful. In our experience, we mostly use program invariants to assert universally quantified properties of objects of a certain type, like injectivity. In this case, the beginning-of-time test passes trivially.

The second test, that every procedure respects J, involves subtle modularity issues. The basic idea is simple: when in a scope D the checker desugars a specification (either in reasoning about a procedure call or in checking a procedure implementation), it adds to the pre- and postcondition all invariants whose declarations are in D. However, if the program consists of a single global scope, then the soundness of this approach is clear: the change to the pre- and postconditions is the same for reasoning about the calls as for checking the implementations. If the program consists of many scopes, then modularity requirements must be imposed to achieve soundness, by ensuring that primitive steps in a scope where the invariant is not visible cannot falsify the invariant. We will build up to the correct modularity requirements in stages. To begin with, we assume that the invariant contains concrete variables only.

The first modularity requirement for invariants that comes to mind is:

> a program invariant must be declared near all of its free variables.

Two declarations are *near* one another if they are contained in the same unit. It follows that they are visible in the same scopes.

This simple modularity requirement achieves soundness because an invariant cannot be falsified except by modifying its free variables. Thus, those procedures whose implementation lies outside the scope of the invariant preserve the invariant because they cannot mention any of its free variables. The rest of the procedures are proved to maintain the invariant.

Unfortunately, this simple requirement is too strong because of the special concrete variable $alloc$, which represents the set of allocated objects and occurs implicitly in almost all invariants: recall from Section 8.1 that a quantification

$$\langle \forall t{:}T \ :: \ \ldots \rangle$$

is desugared to

$$\langle \forall t{:}T \ :: \ alloc[t] \ \Rightarrow \ \ldots \rangle$$

Consequently, it is necessary to loosen the simple rule to allow program invariants to mention $alloc$. This introduces the danger of a procedure falsifying an invariant invisible to it by modifying $alloc$. We address this difficulty by observing that the only way a procedure can directly modify $alloc$ is by performing an allocation, and we can demand of an invariant that it be maintained by any allocation in any portion of the program in which it is not visible. To this end, we say that an invariant J passes the *blind allocation test* for a type T if J is invariant under $\mathbf{new}(T)$.

This brings us to the second version of the modularity requirement for invariants:

(0) a program invariant must be declared near all of its free concrete variables, except $alloc$, and

(1) for all types T, either (a) T is declared near the invariant, or (b) the invariant passes the blind allocation test for T, or (c) T is not mentioned in the invariant.

Here's a sketch of a justification for this version of the modularity requirement: Because of (0), the only invariant-falsifying primitive steps that we need to worry about are those that modify $alloc$, that is, expressions of the form $\mathbf{new}(T)$ for some type T. But it is impossible for the expression $\mathbf{new}(T)$ to falsify the invariant, because for such a T, neither (a) nor (b) nor (c) could hold: not (a), since if T is declared near the invariant, the invariant is visible wherever $\mathbf{new}(T)$ can be called; not (b), since the blind allocation test explicitly checks that $\mathbf{new}(T)$ maintains the invariant; and not (c), since $\mathbf{new}(T)$ cannot falsify the invariant if the invariant doesn't mention T and passes the blind allocation test for T.

In order to pass the blind allocation test, a programmer must choose appropriate default values for the fields of an object type. For example, if a pivot is specified to be injective, its default value should be **nil**.

Let us return to a problem that we touched on in Section 9.1, namely the problem of declaring a data type containing unique identifiers:

unit *U*
  type *T*
  spec var $valid: T \to \mathbf{bool}$
  proc $init(t: T): T$
    modifies $valid[t]$
    ensures $valid'[t] \ \wedge\ \mathbf{result} = t$

unit *UImpl* **import** *U*
  var $id: T \to \mathbf{int}$
  ... (other fields) ...
  rep $valid[t: T] \ \equiv\ \ldots$
  var $gcount: \mathbf{int}$
  impl $init(t: T): T$ **is**
      $id[t] := gcount \,;\, gcount := gcount + 1$
      ...
      $\mathbf{result} := t$
    end

To record the design decisions about *id* and *gcount*, one can add to *UImpl* the program invariants:

$$\mathbf{invariant}\ \langle \forall\, t: T \ ::\ t \neq \mathbf{nil} \ \wedge\ valid[t] \ \Rightarrow\ id[t] < gcount \ \rangle$$
$$\mathbf{invariant}\ \langle \forall\, t,\ u: T \ ::\ t \neq \mathbf{nil} \ \wedge\ u \neq \mathbf{nil} \ \wedge\ valid[t] \ \wedge\ valid[u] \ \wedge\ t \neq u \ \Rightarrow\ id[t] \neq id[u] \ \rangle$$

In this approach, the statements about *id* and *gcount* that were problematical to place in the **rep** declaration (see Section 9.1) have been moved into program invariants. The **rep** declaration for $valid[t]$ concerns only fields of t. This seems an improvement, but this approach still has two problems: one is giving the public *init* method the license to modify the private variable *gcount*, the other is allowing the abstract variable *valid* to appear in a program invariant.

To solve the first problem, we can introduce an abstract variable, say *istate* for internal state, in the interface U:

spec var *istate*: **any**

We then allow *init* to modify *istate*, but *istate* has no other occurrences in the interface:

proc $init(t: T): T$
  modifies $valid[t]$, *istate*
  ensures $valid'[t] \ \wedge$ **result** $= t$

Finally, we add the entire dependency of *istate* on *gcount* to the module *UImpl*:

depends *istate* **on** *gcount*

which by downward closure gives *init* the license to modify *gcount*.

The second problem is that the invariants mention $valid[t]$, but so far we have considered invariants containing concrete variables only. We cannot just eliminate the occurrences of $valid[t]$, since no default value for *id* will make the second invariant pass the blind allocation test for T. The blind allocation test is needed, since T is mentioned in the invariant but T and the invariant are not declared near one another.

One way to solve the second problem is to allow abstract variables in program invariants. We believe that it is sound to do so, provided that the invariant satisfies (0) and (1) from above, and also, for each abstract variable a appearing in the invariant:

(2) all dependencies of a are static, and

(3) either (a) the invariant is declared near a, or (b) the invariant is declared near every **rep** declaration of a and near every dependency of a.

However, this story is getting more complicated than we like. Perhaps it is best simply to forbid abstract variables from appearing in program invariants. If we do, we need some other way of dealing with the occurrences of $valid[t]$ in the program invariants in the unique identifiers example. This we can do simply by inlining them, that is, by replacing $valid[t]$ by whatever expression is given as its rep. Although awkward, this entails no loss of modularity or information hiding, since the invariants occur in a scope (*UImpl*) where the representation of $valid[t]$ is visible.

10 Implementation status

Almost everything described in this paper has been implemented in the Modula-3 Extended Static Checker. Exceptions are:

0. the checker implements only the individual residues, not the shared residue *sres* described in Section 6 beginning on page 34,

1. the checker does not enforce the dependency segregation restriction of Section 7.1 on page 45, but instead uses a more general way of computing the dynamic closure ("upward closure of dynamic predecessors"), which does not necessitate the restriction,

2. the checker does not enforce the disjoint ranges requirement of Section 7.2 (and as mentioned in that section, we leave it to the programmer to avoid abstract aliasing), and

3. the checker does not implement the initialization order checking of Section 9.2.

Our experience with the checker is described in more detail in our companion paper [8]. We have applied the checker to thousands of lines of code, both from the Modula-3 libraries and from programs that use the libraries. In specifying the libraries, we constantly used static and dynamic dependencies.

After experimenting with our Modula-3 checker, we embarked on another project to build an extended static checker for Java [13, 32]. In the ESC/Java project, we circumvented most of the difficulties described in this paper by omitting data abstraction from the annotation language. To partially make up for the omission, we provide object invariants [33] and ghost variables, but the fundamental basis of our decision was to accept less thorough checking in order to produce a simpler checker.

11 Related work

Most work on data abstraction seems to be directed at one of two goals: algorithm design or structuring large systems.

When data abstraction is used for algorithm design, the representation is "inlined" into the site of use as the refinement step of the design [5, 16, 25, 22, 39, 17, 14]. Consequently, the work on this kind of data abstraction is largely unconnected with the large system structuring problems that we are concerned with in this paper. This is not to deny that the underlying mathematics of data abstraction applies to both enterprises. Indeed, our first verification condition generator did not use explicit functionalization of abstract variables but instead used the "change of coordinates" approach common in algorithm refinement. However, we found that the result was that our theorem-prover was constantly forced to apply the "one-point rule" and that for our purposes, explicit functionalization is preferred.

Turning to data abstraction for the purpose of structuring large systems, the earliest treatments were in contexts where there was no independent information-hiding mechanism (like our units) and therefore the problems addressed in the present paper did not arise, or were ignored in the semi-formal treatments in the literature. These treatments include Milner's definition of simulation [37], Hoare's classic treatment of abstraction functions [18], and the influential work of Liskov and Guttag and the rest of the CLU community [35].

The first programming language to support information hiding in the way our units do was Mesa [38], with its definition modules and implementation modules. The Mesa designers appear to have been influenced by Parnas's classic paper on decomposing systems into modules [46]. Mesa in turn influenced Modula [50], Modula-2 [51], Modula-3 [44], Oberon-2 [40], and Ada [4]. Ernst, Hookway, and Ogden have studied the problem of specifying Modula-2 programs where the objects of a module may share some global state [12]. These authors share our concern for modular verification, but the possible scopes they consider are not rich enough to allow subclasses or the *RdRep* interface of our example.

Another, rather different, approach of hiding information is to classify declarations as public or private. This approach is used in Oberon [49], C++ [11], and Java [15]. In the course of the ESC/Java project [13, 32], we used the modularity requirements of the units approach to guide our design for visibility of invariants in the public/private approach [33].

One of the central ideas of this paper, explicit dependency declarations, were introduced in Leino's PhD thesis [28] in 1995. Between that time and this, they have been applied in a number of contexts: they played a central rôle in ESC for Modula-3 [8], and they were incorporated in the specification languages JML [27] and Larch/C++ [26] and in the programming logic of Müller and Poetzsch-Heffter [41]. Another application (or reformulation) of dependency declarations is Leino's technique of Data Groups [30].

As described in Section 7.2, our best attempt at a solution to the problem of abstract aliasing [7] is not fully satisfactory. We do find that our framework of modular soundness and dynamic dependencies has allowed us to give a more incisive definition of the problem than other approaches in the literature, such as Hogg's Islands [20], Almeida's Balloons [3], Utting's Extended Local Stores [48], the Flexible Aliasing Protection of Noble *et al.* [45], and Boyland's Alias Burying [6].

A few other researchers have employed declarations similar to our **depends** declaration connecting an abstract variable to the (more) concrete variables in its representation. Daniel Jackson's Aspect system features dependencies much like ours, but his motivation seems to be to avoid the need for reasoning about the details of the actual representation, whereas we have argued that dependency declarations are necessary even in the presence of full representation declarations [21]. The COLD specification language of Jonkers includes abstract variables (called functions) and dependency declarations between them, but COLD seems not to allow an abstract variable to appear in a modifies list, so it doesn't address many of the problems we have wrestled with [23].

12 Conclusions

We have applied precise formal methods to systems programs that are typical examples of the programming techniques used by careful and experienced contemporary programmers. We found that the formal methods described in the verification literature are inadequate to deal with the patterns of data abstraction and modularization in these programs. We have developed new formal methods to address these shortcomings.

Central to the new methods is the concept of an *abstraction dependency*, which is a kind of abstraction of an abstraction function, in the same sense that an opaque type is an abstraction of a concrete type. A dependency specifies one or more of the variables that occur in an abstraction function, but hides the detailed definition of the function. Just as an opaque type may be widely visible in a multi-module program, while the corresponding concrete type may be visible only narrowly, we discovered that it is often useful to make a dependency more widely visible than the abstraction function itself.

Different kinds of abstraction dependencies occur in different styles of design. Top-down programming leads to static dependencies, where an abstract field of an object is represented in terms of other fields of that same object. Bottom-up programming with reusable libraries leads to dynamic dependencies, where an abstract field of an object is represented in terms of fields of other objects, reachable indirectly from the first object.

We have shown how to verify programs in the presence of static and dynamic dependencies by rewriting modifies lists, preconditions, and postconditions.

For static dependencies, we have two simple *modularity requirements*, which are laws for the placement of dependency declarations in a multi-module program. The requirements do not seem to preclude any useful designs, and we have a formal proof of modular soundness for the requirements. The formal proof makes use of our identification of modular soundness with the monotonicity of verifiability with respect to scope. For dynamic dependencies, we have several modularity requirements, but no soundness theorem, nor any confidence that the list of requirements is complete.

In our experience with static checking of contemporary program libraries, we have found that we use dependencies constantly in our annotations. We have also found that dependencies provide a new perspective on old problems like the problem of encapsulation and rep exposure.

Acknowledgments

At several points in the paper, we have remarked that implementing our ideas in a realistic program checker was critical to many of our discoveries. Here we will remark that Dave Detlefs was critical: he wrote the majority of the code, and he was often the first to see the methodological implications of practical issues.

In Section 6, we attributed the subtle example program to Jim Saxe's penetrating intelligence; we also would like to thank him for his help with cyclic dependencies and other thorny problems.

Every project owes a debt to its devil's advocates, and to our ESC project, Mark Lillibridge contributed both valuable scepticism and helpful ideas.

Rajeev Joshi worked with us as a research intern, and resolved a problem in our initial approach to cyclic dependencies.

We also thank Raymie Stata, who shared his methodological wisdom in many helpful discussions.

We are indebted to Rajeev Joshi, Jim Saxe, Mark Lillibridge, Lyle Ramshaw, and Leslie Lamport for help with various parts of the proof of modular soundness presented in the Appendix.

Finally, we thank our colleagues who commented on earlier drafts of this paper: Dave Detlefs, Cynthia Hibbard, Mark Lillibridge, Raymie Stata, and Mark Vandevoorde.

Appendix: Modular soundness of static dependencies

This appendix provides a proof that the treatment of static dependencies in Sections 5 and 6 is monotonic with respect to scope, that is, that it adheres to modular soundness. In Part I, we describe the user input, that is, programs and their declarations. In Part II, we describe how a user program is transformed into verification conditions (VCs). In Part III, we state the soundness theorem and give its proof. The soundness theorem is that VC generation is monotonic with respect to scope. Our overall proof strategy is to apply semantic-preserving, syntactic transformations to a valid VC generated in a small scope, arriving at the VC generated in a larger scope. We conclude with Part IV, in which we reflect on the theorem and its proof.

Part I
User input

A0 Declarations

A *declaration* introduces a name for a type, field, or method and/or specifies properties of such entities.

Types play almost no rôle in this appendix, but for completeness, we repeat here the two kinds of (object) type declarations:

> **type** T
> **type** $T <: U$

where U names an object type. These introduce the name T for an object type about which nothing is known, except, in the second case, that T is a subtype of the object type U. In addition to object types declared in this way, we postulate a set of predeclared types (**int**, **bool**, etc.).

A concrete field declaration has the form

> **var** $c\colon T \to U$

where T is an object type and U is a type. This introduces the name c for a U-valued concrete field present in all instances of class T.

An abstract field declaration has the form

spec var $a: T \rightarrow U$

where T is an object type and U is a type. This introduces two names: the name a for a U-valued abstract field present in all instances of class T, and the name $res.a$ for a typeless field present in all instances of class T. The field $res.a$ is called an *individual residue variable*, as described in Section 6.3.

There is also a predeclared typeless field $sres$ present in all object instances. This field is called the *shared residue variable*, as described in Section 6.3.

Our only dependency declaration under consideration is a static dependency of the form

depends $a[t: T]$ **on** $f[t]$

where a is an abstract field, f is either an abstract field other than a or a concrete field, T is an object type, and t is a dummy. Given such a declaration, we say that f is a direct dependency of a.

A rep declaration has the form

rep $a[t: T] \equiv e$

where a is an abstract field, e is an expression whose value is represented by a as described in Section 3, T is an object type, and t is a dummy whose scope is e. The only fields that may occur in e are direct dependencies of a, and each such occurrence must be indexed by the dummy t. The only free scalar variable in e is t.

A method specification declaration has the form

method $m(t: T)$ **requires** p **modifies** w **ensures** q

where p and q are expressions, w is a modifies list, T is an object type, and t is a formal parameter that can be used in p, w, and q. This declaration introduces the name m for a method with the given signature and specification. For simplicity, we consider only methods with one parameter (the so-called self parameter); result values and additional parameters can be passed via the fields of the self parameter (*cf.* [0, 1, 31]).

A method implementation declaration has the form

impl $m(t: U)$ **is** C **end**

where m names a method, C is a command, U is an object type, and t is the name of the formal parameter as introduced in the declaration of m. The parameter t can be used in C, but C may not assign to t.

Actual programming languages would place further restrictions on the declarations above, including requirements of well-typedness and non-overlapping rep declarations (see, for example, Section 3). For the purposes of this appendix, however, we consider a more general input, independently of such further restrictions.

We assume that all names introduced are unique. So where actual programming languages may use scope rules to resolve certain names, we assume that all names are always fully qualified.

A1 Scopes

For a set of declarations D, we write $x \in D$ to denote that x is an abstract field, concrete field, or individual residue variable declared in D, or that x is $sres$. For fields or residue variables x and y and a set of declarations D, we write $x \ \mathbf{on}_D \ y$ to denote that x is abstract and y is $res.x$ or $sres$, or that D contains a direct dependency of x on y. We write $\mathbf{on}_D^*$ for the reflexive, transitive closure of $\mathbf{on}_D$.

A set of declarations D is *closed* when every name mentioned in D also has a declaration in D. A set of declarations D is *cycle-free* when there are no distinct names x and y in D such that $x \ \mathbf{on}_D^* \ y \ \wedge \ y \ \mathbf{on}_D^* \ x$.

A *scope* is a (finite and) closed, cycle-free set of declarations. In an actual programming language, a scope would be determined by the units and the import relation among units (see Section 3). However, for the purposes of this appendix, defining scopes by units and imports is over-specific. Therefore, we require simply that the underlying language have some way of specifying scopes and that each scope is a closed, cycle-free set of declarations.

A2 User expressions

Rep, method specification, and method implementation declarations contain commands and expressions. We call these expressions *user expressions*, in contrast to the expressions of verification conditions that will be described in Part II. We will give grammars for commands and user expressions, starting in this section with the grammar for user expressions.

We use e_D to denote a user expression that can be written in a scope D. The shape of e_D is defined by the following grammar:

$$
\begin{array}{rcll}
e_D & ::= & s & \text{scalar variable} \\
 & | & \tilde{f}[e_D] & \text{select} \\
 & | & e_D \ \mathbf{op} \ e_D & \text{any operator other than select} \\
 & | & \langle \forall s \ :: \ e_D \ \rangle & \text{quantified expression}
\end{array}
$$

A scalar variable is a parameter, local variable, or quantified scalar variable. A scalar variable is local to the enclosing declaration, command, or expression.

In the *select* expression $\tilde{f}[e_D]$, f is a field in D (not a residue variable) and $\tilde{f}$ is an *adornment* of f. Three kinds of adornments may be used in user expressions: the default (empty) adornment f, the pre-adornment $\grave{f}$, and the post-adornment $\acute{f}$. Pre- and post-adornments are allowed only in **ensures** clauses, where they are used to denote the values of fields in the pre- and post-states of the method, respectively. The default adornment is used everywhere else and may not be used in **ensures** clauses. In this appendix, we write $\tilde{f}$ and $\bar{f}$ to denote arbitrary adornments of f.

From the point of view of the meaning of an adorned variable, the adornment can be thought of simply as part of the name of the variable. But in the syntactic transformations occurring in verification condition generation and in our proof, it will be necessary to systematically change adornments within a formula.

We call the second argument of a select expression an *index expression*.

Here and throughout, we denote an arbitrary operator by the binary operator $\mathbf{op}$. Extensions to operators of other arities (including nullary operators) will always be straightforward and obvious; we omit them for brevity. We have distinguished between select and other operators in the grammar for user expressions because fields in user expressions are allowed to occur only as first arguments to select. Other than that, select is really just another operator.

The scope of the scalar variable s introduced by the quantified expression is bracketed by $\langle$ and $\rangle$.

Note that residue variables cannot be mentioned explicitly in user expressions.

We say "e is a user expression in D" to mean that e is generated by the grammar for e_D.

A3 Modifies lists

The **modifies** clause of a method specification declaration lists the designators that the method is allowed to modify. We call this list a *modifies list*.

We use w_D to denote a modifies list that can be written in a scope D. The shape of w_D is defined by the following grammar:

$$w_D \quad ::= \quad \text{list of } f[e_D] \quad \text{designator expressions}$$

where in each designator expression $f[e_D]$, f is any field in D and all scalar variables in e_D are parameters (in particular, the self parameter of the method being specified, since that's the only parameter we allow in our simple notation).

The index expression of each designator expression is interpreted as being evaluated in the pre-state of the method (*cf.* Section 7.2 under "Swinging pivots requirement").

We say "w is a modifies list in D" to mean that w is generated by the grammar for w_D.

A4 Commands

We use C_D to denote a command that can be written in a scope D. The shape of C_D is defined by the following grammar:

$$\begin{array}{lcll}
C_D & ::= & s := e_D & \text{simple assignment (to local variable)} \\
& | & c[e_D] := e_D & \text{field update} \\
& | & \textbf{assert } e_D & \\
& | & \textbf{assume } e_D & \\
& | & C_D \ ; \ C_D & \text{sequential composition} \\
& | & C_D \ \square \ C_D & \text{choice composition} \\
& | & \textbf{var } s \textbf{ in } C_D \textbf{ end} & \text{local variable introduction} \\
& | & \textbf{call } m(e_D) & \text{method call}
\end{array}$$

Simple assignment is used to update the values of local variables. (There is no command to update the values of formal parameters once they have been bound through a method call.) The field update command $c[e0] := e1$ sets the concrete field c of object $e0$ to $e1$. There is no command to directly update abstract fields (or residue variables for that matter). Changing the value of a concrete field may cause a change in the values of abstract fields that depend on the concrete field. Residue variables cannot be modified by assignment commands.

The command **assert** e has no effect on the state if e holds, and causes the computation to go wrong if e does not hold. The command **assume** e also has no effect on the state, but can be started only in states that satisfy e. Further description of these commands (other than their formal semantics, which is given below) is beyond the scope of this note (but see, for example, Nelson's *Generalization of Dijkstra's calculus* [43]).

Command $C0\ ;\ C1$ executes $C0$, then $C1$. Command $C0\ \square\ C1$ executes either $C0$ or $C1$, blindly choosing which one. Command **var** s **in** C **end** introduces for use in C new local variable s, with an arbitrary initial value. Finally, **call** $m(e)$ invokes method m with e as the actual self parameter.

Other commands, such as a **new** command that allocates a new object, can be modeled as predefined methods or written in terms of the given commands.

We say " C is a command in D " to mean that C is generated by the grammar for C_D.

Part II

Verification condition generation

A5 Verification conditions

Each method implementation gives rise to a *verification condition*, a logical formula that is valid if and only if the method implementation is correct with respect to the specification; that is, started in a state satisfying the precondition, no execution of the implementation goes wrong, and every terminating execution ends in a state that satisfies the postcondition, having modified only those designators permitted by the modifies list.

Since we are interested in modular verification, the verification condition generation is a function of the scope. For a method implementation C of a method m in a scope D, we write $VC_D(m, C)$ to denote the verification condition generated in D for C.

If m is declared with the specification

requires p **modifies** w **ensures** q

we define $VC_D(m, C)$ as

$$\begin{array}{l} BP_D \ \wedge \ Rep_D \ \wedge \ PW_D \ \wedge \ F_D(p) \ \Rightarrow \\ \langle \forall \grave{zz} :: \ \grave{zz} = zz \ \Rightarrow \\ \qquad wlp_D(C, \langle \forall \acute{zz} :: \ \acute{zz} = zz \ \Rightarrow \ F_D(q) \ \wedge \ mc_D(w) \rangle) \rangle \end{array} \tag{18}$$

where BP_D is the background predicate generated in D, Rep_D denotes the rep axioms of D, PW_D denotes the pointwise axioms of D, F_D is the meta function that functionalizes a user expression in D, zz is the list of concrete fields and residue variables in D, $\grave{zz}$ and $\acute{zz}$ are the list zz but with each field and residue variable pre-adorned and post-adorned, respectively, wlp_D is the meta function that gives the semantics of a command in D, and mc_D is the meta function that generates a modification constraint in D. All of these things will be defined in the next several sections.

For any lists of variables xx and yy of equal lengths (such as $\grave{zz}$ and zz in (18)), we write $xx = yy$ as a shorthand for $x_1 = y_1 \ \wedge \ x_2 = y_2 \ \wedge \ \ldots \ \wedge \ x_N = y_N$ where the x_i's and y_i's are the variables of the two respective lists.

A remark about the second line of the definition of VC_D is in order. This line essentially expresses the weakest precondition of command C with respect to the postcondition $F_D(q) \ \wedge \ mc_D(w)$, that is, the postcondition contributions from the **ensures** clause and **modifies** clause, respectively. However, as we shall see from their definitions, the expressions $F_D(q)$ and $mc_D(w)$ are predicates on the pre-adorned and post-adorned fields, whereas the wlp_D works on the default-adorned fields of its second argument. For any predicate Q, the expression

$$\langle \forall \acute{zz} :: \ \acute{zz} = zz \ \Rightarrow \ Q \rangle$$

effectively changes the coordinates of Q: it says about zz whatever Q says about $\acute{zz}$. This change of coordinates can equivalently be written as a substitution

$$Q(\acute{zz} := zz)$$

This explains the shape of the second argument to wlp_D in the definition of VC_D. The quantification around the wlp_D, which is really also a substitution, serves the purpose of identifying the pre-adorned fields of the wlp_D expression with the default-adorned fields of the initial state.

In Section A12, we give a grammar that generates the kinds of expressions that the meta expression $VC_D(m, C)$ produces. To distinguish these expressions from user expressions, we'll call them *vanilla expressions*.

A6 Functionalization

For any scope D, we define a meta function F_D that *functionalizes* user expressions in D, that is, that turns each occurrence of an abstract field a into an application of the *abstraction function* named $\mathcal{F}.a$ to a's dependencies, as described in Section 5.0. Meta function F_D is defined inductively over the syntactic structure of user expressions, and for convenience we also define F_D on fields and residue variables:

$F_D(s)$	$=$	s	scalar variable
$F_D(\tilde{f}[e])$	$=$	$F_D(\tilde{f})[F_D(e)]$	select
$F_D(\tilde{c})$	$=$	$\tilde{c}$	concrete field
$F_D(\tilde{a})$	$=$	$\mathcal{F}.a(s\tilde{res}, re\tilde{s}.a, F_D(\tilde{f}))$	abstract field
$F_D(e0 \ \mathbf{op} \ e1)$	$=$	$F_D(e0) \ \mathbf{op} \ F_D(e1)$	other operators
$F_D(\langle \forall s :: e \rangle)$	$=$	$\langle \forall s :: F_D(e) \rangle$	quantified expressions
$F_D(\tilde{r})$	$=$	$\tilde{r}$	residue variable

In these equations, the line for select is identical to the line for $\mathbf{op}$, but we list it separately anyway. In the line for $F_D(\tilde{a})$, we have shown the definition for when a is an abstract field with exactly one direct dependency, f, in D. More general forms of this line are straightforward extensions. For example, if a has no dependencies in D, we have

$$F_D(\tilde{a}) \ = \ \mathcal{F}.a(s\tilde{res}, re\tilde{s}.a)$$

and if a has two direct dependencies, $f0$ and $f1$, in D, we have

$$F_D(\tilde{a}) \ = \ \mathcal{F}.a(s\tilde{res}, re\tilde{s}.a, F_D(\tilde{f0}), F_D(\tilde{f1}))$$

Throughout this appendix, we will usually show only the case for one direct dependency.

In ordering the arguments to an abstraction function, any order (for example, alphabetical order) can be used as long as it is used consistently.

Note, by the way, that the definition of F_D is well-founded—that is, its recursive applications will eventually terminate—since D is a scope, which is (finite and) cycle-free.

Finally, note that F_D is really just a substitution: it substitutes a function application for each abstract variable. Consequently,

$$F_D \text{ is monotonic} \tag{19}$$

that is, for boolean user expressions $e0$ and $e1$, if $e0 \ \Rightarrow \ e1$ is universally true, then so is $F_D(e0) \ \Rightarrow \ F_D(e1)$.

A7 Modification constraints

For any scope D and modifies list w in D, we define $mc_D(w)$, the *modification constraint* according to w in D, as follows:

$$mc_D(w) \;=\; \langle \bigwedge x \mid x \in D \;::\; modcon_D(w, x, x) \;\rangle$$

where $modcon_D(w, x, x)$ states that the variable x is not modified except as allowed by w. Before we give the formal definition of $modcon_D$, we introduce two new pieces of notation.

First, here and throughout, we use $\bigwedge$ and $\bigvee$ to denote meta-level quantifiers, that is, macros that generate expressions. The meta-level expression

$$\langle \bigwedge x \mid R(x) \;::\; S(x) \;\rangle$$

generates a conjunction with a conjunct $S(x)$ for every x that satisfies $R(x)$, and similar for $\bigvee$-quantifications which generate disjunctions. Another meta-level expression is set construction. It is written

$$\{ x \mid R(x) \;::\; S(x) \;\}$$

and denotes the set with an element $S(x)$ for every x that satisfies $R(x)$.

Second, recall that our overall proof strategy is to transform the small-scope VC into a large-scope VC that is equally valid. To perform this syntactic transformation, we find it necessary to introduce syntactic markers into the VC as it is constructed. These markers have no semantic significance, but will be used in our proof. To introduce them, we use the syntax $m{:}P$ to denote the expression P labeled with the marker m.

Now for the definition of $modcon_D$ and some auxiliary meta functions.

For any scope D and modifies list w in D, $cl_D(w)$ denotes the static closure of w in D and is defined as the set

$$\{ f, e, x \mid f[e] \in w \;\wedge\; f \;\mathbf{on}_D^*\; x \;::\; x[e] \;\} \tag{20}$$

A property of cl_D that we will use later is that for any abstract variable a, user expression e, and modifies list w in D,

$$a[e] \in cl_D(w) \;\equiv\; res.a[e] \in cl_D(w) \tag{21}$$

(Proof: since $a \;\mathbf{on}_D\; res.a$, both sides of (21) are equivalent to the existence of a term $f[e]$ in w such that $f \;\mathbf{on}_D^*\; a$.)

For any scope D, field or residue variable y in D, modifies list w in D, and dummy variable s, the predicate $modpoint_D(y, w, s)$ states that s is a modification point of y according to w. We define it as follows:

$$modpoint_D(y, w, s) \;=\; \langle \bigvee e \;\|\; y[e] \in cl_D(w) \;::\; s = \grave{e} \,\rangle \tag{22}$$

where e ranges over user expressions and where we have written $\grave{e}$ to denote e in which all fields have been pre-adorned. The reason for this pre-adornment is mentioned in Section 7.2 under "Swinging pivots restriction".

A consequence of the definition of $modpoint$ and the cl property (21) is: for any modifies list w and abstract field a in a scope D, and any dummy s,

$$modpoint_D(a, w, s) = modpoint_D(res.a, w, s) \tag{23}$$

Let D be any scope and w be any modifies list in D. Then, for any fields or residue variables x and y in D, the predicate $modcon_D(w, x, y)$ states that x is unchanged except possibly at the modification points of y according to w. We define it as follows:

$$\begin{aligned} &modcon_D(w, x, y) \;=\; \\ &\qquad\qquad w{:}\langle \forall s \;::\; F_D(\grave{x}[s] = \acute{x}[s] \;\vee\; modpoint_D(y, w, s)) \,\rangle \end{aligned} \tag{24}$$

where s is a fresh dummy. The "w:" in front of the quantification is a syntactic marker without semantic meaning. The predicate is usually used with the second and third arguments being equal, but we will use the extra generality in our proof.

A8 Weakest liberal preconditions

The semantics of commands is defined using *weakest liberal preconditions*. For any scope D, command C in D, and vanilla expression Q in D that contains no free occurrences of post-adorned fields or residue variables, the vanilla expression $wlp_D(C, Q)$ characterizes those pre-states ps of C such that

- no execution of C from ps goes wrong, and
- every terminating execution of C from ps ends in a post-state that satisfies Q.

Meta function wlp_D is defined inductively over the syntactic structure of commands:

$$
\begin{array}{lcl}
wlp_D(s := e,\ Q) & = & \langle \forall s' :: s' = F_D(e) \ \Rightarrow \\
 & & \qquad \langle \forall s :: s = s' \ \Rightarrow\ Q \rangle\rangle \\
wlp_D(c[e0] := e1,\ Q) & = & \langle \forall c' :: c' = store(c, F_D(e0), F_D(e1)) \ \Rightarrow \\
 & & \qquad \langle \forall c :: c = c' \ \Rightarrow\ Q \rangle\rangle \\
wlp_D(\mathbf{assert}\ e,\ Q) & = & F_D(e) \ \wedge\ Q \\
wlp_D(\mathbf{assume}\ e,\ Q) & = & F_D(e) \ \Rightarrow\ Q \\
wlp_D(C0\ ;\ C1,\ Q) & = & wlp_D(C0,\ wlp_D(C1,\ Q)) \\
wlp_D(C0 \ \square\ C1,\ Q) & = & wlp_D(C0,\ Q) \ \wedge\ wlp_D(C1,\ Q) \\
wlp_D(\mathbf{var}\ s\ \mathbf{in}\ C\ \mathbf{end},\ Q) & = & \langle \forall s :: wlp_D(C,\ Q) \rangle
\end{array}
$$

In the first two lines, s' and c' are fresh dummies. In the second line, *store* is the function associated with the select function [36] (see also axiom (43) in Section A22). The last of these lines requires that s not occur free in Q. Finally, for any method m whose specification, after renaming its parameter to a fresh dummy t, is

$$\mathbf{method}\ m(t\colon T)\ \mathbf{requires}\ p\ \mathbf{modifies}\ w\ \mathbf{ensures}\ q$$

we define

$$
\begin{array}{l}
wlp_D(\mathbf{call}\ m(e),\ Q) \ = \\
\quad \langle \forall t :: t = F_D(e) \ \Rightarrow \\
\qquad F_D(p) \ \wedge \\
\qquad \langle \forall \ddot{zz} :: \ddot{zz} = \grave{zz} \ \Rightarrow\ \langle \forall \grave{zz} :: \grave{zz} = zz \ \Rightarrow \\
\qquad\quad \langle \forall \acute{zz} :: F_D(q) \ \wedge\ mc_D(w) \ \Rightarrow \\
\qquad\qquad \langle \forall \grave{zz} :: \grave{zz} = \ddot{zz} \ \Rightarrow\ \langle \forall zz :: zz = \acute{zz} \ \Rightarrow\ Q \rangle\rangle\rangle\rangle\rangle\rangle
\end{array}
$$

where zz is the list of concrete fields and residue variables in D, and $\ddot{zz}$ denotes fresh adornments of zz. The reason for introducing $\ddot{zz}$ is to effectively avoid variable capture of $\grave{zz}$ in Q (which may be easier to see in the equivalent formulation with substitutions, next).

In the definition of wlp_D, the quantifications occurring in the simple assignment, field update, and method call commands can also be written as substitutions. The wlp_D of these commands are thus equivalently written as

$$
\begin{array}{lcl}
wlp_D(s := e,\ Q) & = & Q(s := F_D(e)) \\
wlp_D(c[e0] := e1,\ Q) & = & Q(c := store(c, F_D(e0), F_D(e1))) \\
wlp_D(\mathbf{call}\ m(e),\ Q) & = & \\
\multicolumn{3}{l}{\quad \langle \forall t :: t = F_D(e) \ \Rightarrow} \\
\multicolumn{3}{l}{\qquad F_D(p) \ \wedge\ \langle \forall \acute{zz} :: (F_D(q) \ \wedge\ mc_D(w))(\grave{zz} := zz) \ \Rightarrow\ Q(zz := \acute{zz}) \rangle\rangle}
\end{array}
$$

A9 Rep axioms

For any scope D, we define Rep_D as a conjunction with one conjunct for each rep declaration in D. For a rep declaration

$$\textbf{rep}\ a[t\colon T] \ \equiv\ e$$

the corresponding conjunct, called a *rep axiom*, is

$$\langle \forall\, t\colon T,\ res.a, f \ ::\ t \neq \textbf{nil} \ \Rightarrow\ \mathcal{F}.a(sres,\ res.a, f)[t] = e \ \rangle$$

where we have shown the case where a has exactly one direct dependency, f, in D.

Note that e is not functionalized, see Section 5.0.

This is the only place in our verification condition generation where we make use of types: we retain the type of the dummy t in the rep axiom. (This is not important for the soundness of modular verification, but it is important in order to generate a desirable verification condition. In particular, this avoids logical inconsistencies resulting from inconsistent rep axioms.)

A10 Pointwise axioms

Because occurrences of dependencies of a in a rep declaration for $a[t\colon T]$ are indexed by t, it follows that the abstract value of $a[t]$ does not change if a's dependencies change only at objects other than t. But our functionalization, which transforms $a[t]$ into an expression like $\mathcal{F}.a(sres,\ res.a, f)[t]$, seems to lose this fact. To compensate, we supply an explicit *pointwise axiom* for each abstraction function, see Section 5.0.

For any scope D, we define PW_D as a conjunction with one conjunct for each abstract field in D. For an abstract field a, the corresponding conjunct is the pointwise axiom for $\mathcal{F}.a$:

$$\begin{array}{l}\langle \forall\, t,\ \grave{sres},\ \acute{sres},\ \grave{res}.a,\ \acute{res}.a, \grave{f}, \acute{f} \ :: \\ \qquad \grave{sres}[t] = \acute{sres}[t] \ \wedge\ \grave{res}.a[t] = \acute{res}.a[t] \ \wedge\ \grave{f}[t] = \acute{f}[t] \\ \quad \Rightarrow \\ \qquad \mathcal{F}.a(\grave{sres},\ \grave{res}.a, \grave{f})[t] = \mathcal{F}.a(\acute{sres},\ \acute{res}.a, \acute{f})[t] \ \rangle\end{array} \tag{25}$$

where t is a fresh dummy, and where we have shown the case where a has exactly one direct dependency, f, in D.

A11 Background predicate

Each scope gives rise to a *background predicate*, which is a conjunction of axioms that formalize various properties of the type system. For any scope D, we use BP_D to denote the background predicate produced in D. The exact axioms placed in the background predicate are not relevant to this appendix, but we do assume that the set of axioms produced grows monotonically with the set of type declarations in a scope. That is, let BPS_D denote the set of background-predicate conjuncts produced for a scope D so that

$$BP_D = \langle \bigwedge Q \mid Q \in BPS_D :: Q \rangle$$

and let $Types(D)$ denote the set of type declarations in D; then we assume BPS to have the following property, for any scopes D and E:

$$Types(D) \subseteq Types(E) \Rightarrow BPS_D \subseteq BPS_E$$

Consequently, we have

$$D \subseteq E \Rightarrow [BP_D \Leftarrow BP_E] \tag{26}$$

Note that if $Types(D) = Types(E)$, then $BPS_D = BPS_E$. Thus, a consequence of our assumption about the background predicate is that BP_D does not depend on the field declarations in D.

A12 Vanilla expressions

Now that we have defined all of $VC_D(m, C)$, we can describe the shape of the resulting expressions. Such a description will come in handy in Part III of this appendix. We define a *vanilla expression* to be an expression that is generated by the grammar below and satisfies characteristics V0 through V5, listed below. We claim that any expression generated by $VC_D(m, C)$ is a vanilla expression.

Using Q_D to denote a typical vanilla expression in a scope D, the grammar is:

$$
\begin{array}{lll}
Q_D ::= & & \\
 & s & \text{scalar variable} \\
| & \tilde{c} & \text{concrete field} \\
| & \mathcal{F}.a(\widetilde{sres},\, \widetilde{res}.a,\, Q_D) & \text{abstraction function} \\
| & Q_D \ \mathbf{op} \ Q_D & \text{operator, possibly select or store} \\
| & \langle \forall s \ :: \ Q_D \rangle & \text{quantification over scalar} \\
| & \langle \forall \tilde{x} \ :: \ Q_D \rangle & \text{quantification over concrete field or} \\
 & & \text{residue variable} \\
| & \langle \forall \tilde{r} \ :: \ \tilde{r} = \bar{r} \ \Rightarrow \ Q_D \rangle & \text{quantification over residue variable,} \\
 & & \text{with equality antecedent} \\
| & w_D{:}\langle \forall s \ :: \ \grave{r}[s] = \acute{r}[s] \ \vee \ Q_D \rangle & \text{residue modification constraint} \\
| & \text{see formula (25), page 84} & \text{pointwise axiom} \\
| & \multicolumn{2}{l}{\langle \forall t{:}\, T, sres, res.a, f \ :: \ t \neq \mathbf{nil} \ \Rightarrow \ \mathcal{F}.a(sres, res.a, f)[t] = e_D \rangle} \\
 & & \text{rep axiom}
\end{array}
$$

As usual, we have shown only the cases where an abstract variable has exactly one dependency in D; extensions to other numbers of dependencies are straightforward and obvious. We have treated expressions with markers, like $w{:}Q$, as unary expressions where the "$w{:}$" is the operator, except in the case of residue modification constraints, whose inside of the quantified formula is otherwise not a vanilla expression.

The definition of VC_D seems not to fit this grammar, because it produces quantified expressions that quantify over all concrete fields and residue variables in D. For example, visible directly in the definition of VC_D, (18) on page 79, is a subexpression of the form

$$\langle \forall \grave{zz} \ :: \ \grave{zz} = zz \ \Rightarrow \ Q \rangle \tag{27}$$

where zz is the list of all concrete fields and residue variables in D. To fit the grammar for vanilla expressions, we must think of expression (27) as the semantically equivalent expression

$$
\begin{array}{l}
\langle \forall \grave{c}_1 \ :: \ \grave{c}_1 = c_1 \ \Rightarrow \ \langle \forall \grave{c}_2 \ :: \ \grave{c}_2 = c_2 \ \Rightarrow \ \cdots \ \langle \forall \grave{c}_N \ :: \ \grave{c}_N = c_N \ \Rightarrow \\
\quad \langle \forall \grave{sres} \ :: \ \grave{sres} = sres \ \Rightarrow \\
\qquad \langle \forall \grave{r}_1 \ :: \ \grave{r}_1 = r_1 \ \Rightarrow \ \cdots \ \langle \forall \grave{r}_K \ :: \ \grave{r}_K = r_K \ \Rightarrow \ Q \rangle \cdots \rangle\rangle\rangle \cdots \rangle\rangle
\end{array}
$$

where the c_i's denote the concrete fields in D and the r_i's denote the individual residue variables in D. A similar remark applies to the other quantification in

(18) and the quantifications in the definition of wlp_D for a method call. We will treat the two representations interchangeably and refer to the quantifications as forming a *cluster*.

We say the bound variables (adorned concrete fields and residue variables) of one cluster belong to the same *generation*. Variables of the same generation have the same adornments, but variables with the same adornments are not necessarily of the same generation.

The formulas generated by VC_D have more characteristics than are expressed by the grammar. We note the following characteristics of formulas generated by $VC_D(m, C)$:

V0. Pointwise axioms and rep axioms occur only in negative positions.

V1. For any concrete field or residue variable x, quantifications over $\tilde{x}$, with and without equality antecedents (more precisely, the quantifications occurring in the sixth and seventh lines of the grammar of Q_D), occur only in positive positions.

V2. For any concrete field or residue variable x, if there is no field update command of x in C, then any quantification over $\tilde{x}$, with or without equality antecedent, appears in $VC_D(m, C)$ in a cluster of quantifications (with or without equality antecedents, respectively) over all the variables in $\tilde{zz}$, where zz is the list of concrete fields and residue variables in D.

V3. Modification constraints, and in particular shared-residue modification constraints, are generated only by the *modcon* meta function. Thus, a residue modification constraint

$$w{:}\langle\, \forall s \;::\; \grave{sres}[s] = \acute{sres}[s] \;\vee\; Q \,\rangle \tag{28}$$

occurring in $VC_D(m, C)$ has the form dictated by

$$modcon_D(w, sres, sres)$$

In other words, the Q in (28) necessarily has the form

$$F_D(modpoint_D(sres, w, s))$$

V4. Each occurrence of an abstraction function $\mathcal{F}.a$ is generated either by the meta function F_D or as part of a pointwise axiom or rep axiom. By inspecting these definitions, one finds that the arguments to $\mathcal{F}.a$ are $sres$, $res.a$, and the direct dependencies of a, all adorned in the same way.

V5. For any subexpression Q in $VC_D(m, C)$ that does not break any cluster of quantifications (that is, Q does not contain part of a cluster without containing all of it), the free occurrences in Q of concrete fields and residue variables with the same adornments belong to the same generation, except possibly for those concrete fields for which C contains a field update command. Consequently, and using V4, the arguments of any given application of an abstraction function $\mathcal{F}.a$ are all of the same generation, except possibly those concrete-field arguments for which C contains a field update command.

We define a vanilla expression to be any expression generated by the grammar above and satisfying the characteristics V0 through V5.

So much for verification condition generation.

Part III

Modular soundness

A13 The theorem of soundness of modular verification

Soundness of modular verification is what justifies that one can prove the correctness of a method implementation without needing the entire program. This is important because it allows modules to be proved separately, without a need to re-prove them as they are linked together. Our theorem of soundness of modular verification states that to prove the VC generated in a larger scope for some method implementation, it suffices to prove the VC generated in a smaller scope for that method implementation.

We use the notation $[P]$ (pronounced "everywhere P") to say that a formula P is true in all infinite models. Since the theory with respect to which we verify programs includes the integers, of which there are infinitely many, "$[P]$" is for our purposes as good as saying that "P is valid", that is, that P is semantically equivalent to *true* [10]. Nevertheless, for technical reasons that we will describe later, we must define our "everywhere brackets" to ignore finite models. Our everywhere brackets satisfy $[P] \equiv [\langle \forall x :: P \rangle]$ for x an individual variable or function symbol, and therefore we may think of them as universally quantifying all free variables and function symbols over some infinite domain.

Equipped with everywhere brackets, we can now state our soundness theorem. As discussed in Section 6, the property

$$D \subseteq E \;\wedge\; [VC_D(m, C)] \;\Rightarrow\; [VC_E(m, C)]$$

where D and E are scopes containing an implementation C for a method m, does not hold for all scopes D and E. This discussion then led us to the introduction of the visibility and top-down requirements for static dependencies. Here, we formalize those requirements as a relation on two scopes D and E, where $D \subseteq E$:

$$\begin{array}{lcl} \mathit{Vis}(D, E) & = & \langle \forall x, y :: x \in D \;\wedge\; y \in D \;\wedge\; x \;\mathbf{on}_E\; y \;\Rightarrow\; x \;\mathbf{on}_D\; y \rangle \\ \mathit{TopDown}(D, E) & = & \langle \forall x, y :: y \in D \;\wedge\; x \;\mathbf{on}_E\; y \;\Rightarrow\; x \in D \rangle \end{array}$$

The theorem of soundness of modular verification is:

Soundness Theorem. For any scopes D and E, containing an implementation C for a method m,

$$D \subseteq E \;\wedge\; \mathit{Vis}(D, E) \;\wedge\; \mathit{TopDown}(D, E) \;\wedge\; [VC_D(m, C)] \;\Rightarrow\; [VC_E(m, C)]$$

The theorem essentially states that VC generation is monotonic with respect to scope.

Note that the theorem is not about the soundness of the VC generation with respect to some operational semantics of the command language. In fact, the theorem says nothing about the utility of the meta functions VC_D and VC_E — for all we know, these meta functions might generate formulas that are totally irrelevant to any question of program correctness. Our theorem simply states that if the VC generated by VC_D is valid, then so is the VC generated by VC_E.

But of course we do claim that the theorem is useful, because we claim, without proof, that VC_E generates a VC that is appropriate for an entire program E, modulo the issues discussed in Part IV of this appendix.

A14 Proof strategy

The soundness theorem states that if the modularity requirements are respected, then adding declarations to a scope does not invalidate method implementations in the scope. For some kinds of added declarations, the proof is quite easy; for others, it is harder. We begin with two lemmas (Soundness Lemmas A and B, in Section A15) that handle all the easy cases (namely: additional type declarations, rep declarations, method declarations, and method implementation declarations). We then proceed to the difficult case, which is where the added declarations consist of a new field together with some number of dependencies on the field. Soundness Lemma C (Section A16) applies to this case if the new field is concrete, and Soundness Lemma D (also in Section A16) applies if the new field is abstract. The proof of Soundness Lemma D relies on Soundness Lemma C, so the heart of the soundness proof is Soundness Lemma C. Soundness Lemma E (Section A16) combines Soundness Lemmas C and D in a straightforward way. Soundness Lemma F (Section A17) uses induction and Soundness Lemma E to prove the soundness of extensions by multiple fields. Finally, Section A18 shows that these soundness lemmas add up to a proof of the Soundness Theorem.

In Section A16, we state Soundness Lemma C and sketch an informal argument for its correctness, but the formal proof (Sections A19–A32) is deferred for the convenience of any readers who may prefer to skim it.

Our journey will be long. To make it as readable as possible, we make heavy use of the calculational proof format suggested by Wim H. J. Feijen [10]. We hope its explicit hints will make each of the numerous little proof steps manageable to check.

A15 Type, rep, and method declaration discrepancies

In this section and the next two, we consider specializations of the Soundness Theorem. We start in this section by stating and proving two lemmas. The first lemma considers scopes D and E that differ only in their type declarations. The second lemma considers scopes that differ only in their rep and method declarations.

Soundness Lemma A. For any scopes D and E, containing an implementation C for a method m, if D and E differ only in their type declarations (that is, if the sets D and E minus their type declarations are equal), then

$$D \subseteq E \ \wedge\ Vis(D,E) \ \wedge\ TopDown(D,E) \ \wedge\ [VC_D(m,C)] \ \Rightarrow\ [VC_E(m,C)]$$

Proof. Let zz denote the list of concrete fields and residue variables in D, and suppose the specification of m is

requires p **modifies** w **ensures** q

We then calculate,

$$
\begin{array}{ll}
 & [VC_D(m,C)] \\
= & \qquad \{\ VC_D\ \} \\
 & [BP_D \ \wedge\ Rep_D \ \wedge\ PW_D \ \wedge\ F_D(p) \ \Rightarrow\ \langle \forall\, \grave{z}\grave{z} :: \grave{z}\grave{z} = zz \ \Rightarrow \\
 & \qquad wlp_D(C, \langle \forall\, \acute{z}\acute{z} :: \acute{z}\acute{z} = zz \ \Rightarrow\ F_D(q) \ \wedge\ mc_D(w)\ \rangle)\ \rangle] \\
\Rightarrow & \qquad \{\ \text{by } D \subseteq E \text{ and (26), we have } [BP_D \ \Leftarrow\ BP_E]\ \} \\
 & [BP_E \ \wedge\ Rep_D \ \wedge\ PW_D \ \wedge\ F_D(p) \ \Rightarrow\ \langle \forall\, \grave{z}\grave{z} :: \grave{z}\grave{z} = zz \ \Rightarrow \\
 & \qquad wlp_D(C, \langle \forall\, \acute{z}\acute{z} :: \acute{z}\acute{z} = zz \ \Rightarrow\ F_D(q) \ \wedge\ mc_D(w)\ \rangle)\ \rangle] \\
= & \qquad \{\ D \text{ and } E \text{ are equal except for their type declarations, so} \\
 & \qquad\quad Rep_E = Rep_D,\ PW_E = PW_D,\ F_E = F_D,\ wlp_E = wlp_D, \text{ and} \\
 & \qquad\quad mc_E = mc_D\ \} \\
 & [BP_E \ \wedge\ Rep_E \ \wedge\ PW_E \ \wedge\ F_E(p) \ \Rightarrow\ \langle \forall\, \grave{z}\grave{z} :: \grave{z}\grave{z} = zz \ \Rightarrow \\
 & \qquad wlp_E(C, \langle \forall\, \acute{z}\acute{z} :: \acute{z}\acute{z} = zz \ \Rightarrow\ F_E(q) \ \wedge\ mc_E(w)\ \rangle)\ \rangle] \\
= & \qquad \{\ VC_E, \text{ since } zz \text{ is the list of concrete fields and residue variables} \\
 & \qquad\quad \text{in } E\ \} \\
 & [VC_E(m,C)]
\end{array}
$$

■

Soundness Lemma B. For any scopes D and E, containing an implementation C for a method m, if D and E differ only in their rep, method specification, and method implementation declarations, then

$$D \subseteq E \ \wedge\ Vis(D,E) \ \wedge\ TopDown(D,E) \ \wedge\ [VC_D(m,C)] \ \Rightarrow\ [VC_E(m,C)]$$

Proof. Let zz denote the list of concrete fields and residue variables in D (hence also in E), and suppose the specification of m is

requires p **modifies** w **ensures** q

We calculate,

$$
\begin{array}{ll}
 & [VC_D(m, C)] \\
= & \quad \{ \quad VC_D \quad \} \\
 & [BP_D \;\wedge\; Rep_D \;\wedge\; PW_D \;\wedge\; F_D(p) \;\Rightarrow\; \langle \forall \grave{z}z :: \grave{z}z = zz \;\Rightarrow \\
 & \quad wlp_D(C, \langle \forall \acute{z}z :: \acute{z}z = zz \;\Rightarrow\; F_D(q) \;\wedge\; mc_D(w) \,\rangle) \,\rangle] \\
\Rightarrow & \quad \{ \quad \text{by } D \subseteq E \text{ and (26), we have } [BP_D \;\Leftarrow\; BP_E] \quad \} \\
 & [BP_E \;\wedge\; Rep_D \;\wedge\; PW_D \;\wedge\; F_D(p) \;\Rightarrow\; \langle \forall \grave{z}z :: \grave{z}z = zz \;\Rightarrow \\
 & \quad wlp_D(C, \langle \forall \acute{z}z :: \acute{z}z = zz \;\Rightarrow\; F_D(q) \;\wedge\; mc_D(w) \,\rangle) \,\rangle] \\
\Rightarrow & \quad \{ \quad \text{by } D \subseteq E\text{, the fact that } D \text{ and } E \text{ coincide in their field and} \\
 & \quad\quad \text{dependency declarations, and the definition of rep axioms, we} \\
 & \quad\quad \text{have } [Rep_E \;\Rightarrow\; Rep_D] \quad \} \\
 & [BP_E \;\wedge\; Rep_E \;\wedge\; PW_D \;\wedge\; F_D(p) \;\Rightarrow\; \langle \forall \grave{z}z :: \grave{z}z = zz \;\Rightarrow \\
 & \quad wlp_D(C, \langle \forall \acute{z}z :: \acute{z}z = zz \;\Rightarrow\; F_D(q) \;\wedge\; mc_D(w) \,\rangle) \,\rangle] \\
= & \quad \{ \quad D \text{ and } E \text{ are equal except for their rep, method specification,} \\
 & \quad\quad \text{and method implementation declarations, so } PW_D = PW_E\text{,} \\
 & \quad\quad F_D = F_E\text{, } wlp_D = wlp_E\text{, and } mc_D = mc_E \quad \} \\
 & [BP_E \;\wedge\; Rep_E \;\wedge\; PW_E \;\wedge\; F_E(p) \;\Rightarrow\; \langle \forall \grave{z}z :: \grave{z}z = zz \;\Rightarrow \\
 & \quad wlp_E(C, \langle \forall \acute{z}z :: \acute{z}z = zz \;\Rightarrow\; F_E(q) \;\wedge\; mc_E(w) \,\rangle) \,\rangle] \\
= & \quad \{ \quad VC_E\text{, since } zz \text{ is the list of concrete fields and residue variables} \\
 & \quad\quad \text{in } E \quad \} \\
 & [VC_E(m, C)]
\end{array}
$$

■

A16 Single field-declaration discrepancies

In this section, we state three lemmas as specializations of the Soundness Theorem. We prove the second and third, but defer the proof of the first lemma until Section A19 and beyond.

We define a property *Extend*. For any D, ε, and E, $Extend(D, \varepsilon, E)$ says that D and E are scopes that differ only in that E additionally contains the declaration of a concrete field ε and some number of dependencies on ε. Note that

$$Extend(D, \varepsilon, E) \;\Rightarrow\; D \subseteq E \;\wedge\; Vis(D, E) \;\wedge\; TopDown(D, E)$$

Here's the first of the three lemmas of this section, a lemma that captures the essence of the Soundness Theorem.

Soundness Lemma C. For any scopes D and E, containing an implementation C for a method m, and any concrete field ε,

$$Extend(D, \varepsilon, E) \;\wedge\; [VC_D(m, C)] \;\Rightarrow\; [VC_E(m, C)]$$

The formal proof of Soundness Lemma C is presented in Sections A19 through A32. In the meantime, here is an informal sketch of an argument for the correctness of the lemma.

Proof sketch. Given $Extend(D, \varepsilon, E)$ and that $VC_D(m, C)$ is valid, we must prove that $VC_E(m, C)$ is also valid. There are two differences between $VC_D(m, C)$ and $VC_E(m, C)$.

The first difference is that, for any abstract variable a that depends on ε (in E), occurrences of a are functionalized differently in D and in E. Wherever we have in D a subexpression like $\mathcal{F}.a(sres, \ldots)$, we have instead in E a subexpression like $\mathcal{F}.a(sres, \ldots, \varepsilon)$.

The second and more difficult difference is that modifies lists are desugared differently in D and in E: the modification constraints in E mention ε, but the corresponding modification constraints in D do not mention ε.

The first difference is resolved using shared residues: the universal validity of the formula $VC_D(m, C)$ implies the validity of $\langle \forall\, sres :: \; VC_D(m, C) \; \rangle$, which means that for the D-scope $sres$ we can substitute the ordered pair of ε and the E-scope $sres$. Thus by a validity-preserving transformation, the D-subexpression $\mathcal{F}.a(sres, \ldots)$ becomes $\mathcal{F}.a((sres, \varepsilon), \ldots)$. The universal validity also allows us to substitute any function for the uninterpreted function $\mathcal{F}.a$. So for the D-scope $\mathcal{F}.a$ we can then substitute a function that picks apart the ordered pair and applies the E-scope $\mathcal{F}.a$ to $sres$, ε, and any other arguments, which is exactly the E-scope expression to which occurrences of a are functionalized. Thus by composing these two validity-preserving transformations, the differences between VC_D and VC_E due to differently functionalized abstract variables disappear.

The second difference is resolved using individual residues: Since m's implementation C contains no occurrences of ε, we must prove that m's modification constraint for ε follows from the various modification constraints for ε arising for each method call in C. For any abstract variable a, the VC in the small scope asserts that m's modification constraint for $res.a$ follows from the various modification constraints for $res.a$ arising for each method call in C, because there are no occurrences of residue variables in C. Since the modification points of ε equals the union of the modification points of the abstract variables that depend on

ε, which coincides with the modification points of the residues of those variables, the small-scope proof for the residues implies the large-scope proof for ε. ■

The next lemma is like Soundness Lemma C, except that ε is abstract. We will not define another flavor of *Extend* for abstract fields, so the lemma repeats the relevant properties of *Extend*.

Soundness Lemma D. Let D and E be scopes containing an implementation C for a method m. If D and E differ only in that E additionally contains the declaration of an abstract field ε and some number of dependencies on ε, then

$$[VC_D(m, C)] \;\Rightarrow\; [VC_E(m, C)]$$

Proof. We prove the lemma by constructing two new scopes G and H, applying Soundness Lemma C to the pairs of scopes (D, G) and (G, H), respectively, and then obtaining $[VC_E(m, C)]$ by massaging the formula $[VC_H(m, C)]$.

Let D and E satisfy the antecedent of the lemma, and let G be the set E but where E contains the declaration

$$\textbf{spec var}\ \varepsilon\colon T \to U$$

G instead contains the declaration

$$\textbf{var}\ \varepsilon\colon T \to U$$

We have that D and G now fit the description of D and E in the antecedent of Soundness Lemma C, from which we then conclude $[VC_G(m, C)]$.

Let H be the set G but where G contains the declaration

$$\textbf{var}\ \varepsilon\colon T \to U$$

H additionally contains the declaration

$$\textbf{var}\ \delta\colon T \to U$$

and where G contains a declaration

$$\textbf{depends}\ a[v\colon V]\ \textbf{on}\ \varepsilon[v]$$

for any a, v, and V, set H additionally contains the declaration

$$\textbf{depends}\ a[v\colon V]\ \textbf{on}\ \delta[v]$$

We have that G and H fit the description of D and E in the antecedent of Soundness Lemma C, from which we conclude $[VC_H(m, C)]$.

Let us now consider how the formula $VC_H(m, C)$ differs in structure from the formula $VC_E(m, C)$, so that we can take validity-preserving steps to reconcile the differences.

Let zz denote the concrete fields and residue variables in D, and suppose the specification of m is

requires p **modifies** w **ensures** q

Then $VC_H(m, C)$ is the expression

$$\begin{array}{l} BP_H \ \wedge \ Rep_H \ \wedge \ PW_H \ \wedge \ F_H(p) \ \Rightarrow \\ \quad \langle \forall \grave{zz}, \grave{\varepsilon}, \grave{\delta} \ :: \ \grave{zz} = zz \ \wedge \ \grave{\varepsilon} = \varepsilon \ \wedge \ \grave{\delta} = \delta \ \Rightarrow \\ \qquad wlp_H(C, \langle \forall \acute{zz}, \acute{\varepsilon}, \acute{\delta} \ :: \ \acute{zz} = zz \ \wedge \ \acute{\varepsilon} = \varepsilon \ \wedge \ \acute{\delta} = \delta \ \Rightarrow \\ \qquad\qquad\qquad F_H(q) \ \wedge \ mc_H(w) \ \rangle) \ \rangle \end{array}$$

Scope H contains the concrete fields ε and δ, whereas scope E contains the abstract field ε. Since the user expressions that play a part in the generation of the verification conditions $VC_H(m, C)$ and $VC_E(m, C)$ are all in the smaller scope D, there is no direct mention of ε or δ in the command C or in the specifications of the methods called from C. But ε and δ may still appear in $VC_H(m, C)$ and $VC_E(m, C)$, as follows:

- For every abstract variable a that depends on ε in E, and thus on ε and δ in H, the abstraction function $\mathcal{F}.a$ in H has arguments corresponding to both ε and δ, whereas the function $\mathcal{F}.a$ in E has an argument corresponding to ε but not to δ. The extra argument is found in all occurrences of $\mathcal{F}.a$ in $VC_H(m, C)$, that is, in functionalized user expressions, in rep axioms for a, and in the pointwise axiom for $\mathcal{F}.a$.

- For every modifies list v, $mc_H(v)$ contains the conjuncts

$$\begin{array}{l} v{:}\langle \forall s \ :: \ \grave{\varepsilon}[s] = \acute{\varepsilon}[s] \ \vee \ F_H(modpoint_H(\varepsilon, v, s)) \ \rangle \ \wedge \\ v{:}\langle \forall s \ :: \ \grave{\delta}[s] = \acute{\delta}[s] \ \vee \ F_H(modpoint_H(\delta, v, s)) \ \rangle \end{array}$$

  whereas $mc_E(v)$ instead contains the conjuncts

$$\begin{array}{l} v{:}\langle \forall s \ :: \ \mathcal{F}.\varepsilon(s\grave{res}, \grave{\varepsilon})[s] = \mathcal{F}.\varepsilon(s\acute{res}, \acute{\varepsilon})[s] \ \vee \\ \qquad\qquad\qquad F_E(modpoint_E(\varepsilon, v, s)) \ \rangle \ \wedge \\ v{:}\langle \forall s \ :: \ \grave{res}.\varepsilon[s] = \acute{res}.\varepsilon[s] \ \vee \ F_E(modpoint_E(res.\varepsilon, v, s)) \ \rangle \end{array}$$

Note that since v does not directly mention ε or δ, and since an abstract field depends on ε in H exactly when it also depends on δ in H, we have

$$\langle \forall e :: \varepsilon[e] \in cl_H(v) \equiv \delta[e] \in cl_H(v) \rangle$$

and hence $modpoint_H(\varepsilon, v, s) = modpoint_H(\delta, v, s)$. Moreover, due to (23), we have $modpoint_E(\varepsilon, v, s) = modpoint_E(res.\varepsilon, v, s)$. Finally, since an abstract field depends on ε in H exactly when it depends on ε in E, we have

$$\langle \forall e :: \varepsilon[e] \in cl_H(v) \equiv \varepsilon[e] \in cl_E(v) \rangle$$

and hence $modpoint_H(\varepsilon, v, s) = modpoint_E(\varepsilon, v, s)$. Thus, the expansions of the four *modpoint* expressions above are the same. Their functionalizations differ as described in the previous bullet.

- Since ε is abstract in E, $VC_E(m, C)$ contains a pointwise axiom for $\mathcal{F}.\varepsilon$. There is no such axiom in $VC_H(m, C)$.
- Where the verification conditions contain quantifications over all concrete fields and residue variables, with or without equality antecedents, the quantifications in $VC_H(m, C)$ are over $\tilde{\varepsilon}$ and $\tilde{\delta}$, possibly with the equality antecedent $\tilde{\varepsilon} = \bar{\varepsilon} \land \tilde{\delta} = \bar{\delta}$, whereas the quantifications in $VC_E(m, C)$ are over $r\tilde{e}s.\varepsilon$, with the respective antecedent $r\tilde{e}s.\varepsilon = r\bar{e}s.\varepsilon$.

The verification conditions $VC_H(m, C)$ and $VC_E(m, C)$ are vanilla expressions as defined by the grammar and characteristics V0 through V5 in Section A12. Figure 11 classifies the differences between the two VCs according to the vanilla grammar. In particular, it shows all the ways that $\tilde{\varepsilon}$, $\tilde{\delta}$, $\mathcal{F}.\varepsilon$, and $r\tilde{e}s.\varepsilon$ may occur. Using characteristic V2 (page 87), the figure shows the quantifications over $\tilde{\varepsilon}$ and $\tilde{\delta}$ together with the neighboring quantifications over $s\tilde{r}es$. Squinting at this formula, we see that variables ε and δ in $VC_H(m, C)$ play rôles that are similar but not identical to $\mathcal{F}.\varepsilon(sres, res.\varepsilon)$ and $res.\varepsilon$, respectively, in $VC_E(m, C)$. We will have to work on the differences.

We are now starting the process of transforming $VC_H(m, C)$ into $VC_E(m, C)$. We do so by a series of transformations to *the working formula*, which starts off as the formula $VC_H(m, C)$ and ends up as $VC_E(m, C)$. Our series of transformations are guided by the differences shown in Figure 11. To help us keep track of the effect of the transformations applied so far, we will provide sketches of the

$VC_H(m, C)$	$VC_E(m, C)$
A0. Functionalized occurrences of abstract variables that depend on ε:	
$\mathcal{F}.a(\widetilde{sres}, \ldots, \tilde{\varepsilon}, \tilde{\delta})$	$\mathcal{F}.a(\widetilde{sres}, \ldots, \mathcal{F}.\varepsilon(\widetilde{sres}, \widetilde{res}.\varepsilon))$
A1. Quantifications over concrete fields and residue variables:	
$\langle \forall \widetilde{sres}, \tilde{\delta}, \tilde{\varepsilon} :: \ \ldots \rangle$	$\langle \forall \widetilde{sres}, \widetilde{res}.\varepsilon :: \ \ldots \rangle$
A2. Quantifications over concrete fields and residue variables, with antecedents:	
$\langle \forall \widetilde{sres}, \tilde{\delta}, \tilde{\varepsilon} :: \ \widetilde{sres} = \overline{sres} \ \wedge \ \tilde{\delta} = \bar{\delta} \ \wedge \ \tilde{\varepsilon} = \bar{\varepsilon} \ \Rightarrow \ \ldots \rangle$	$\langle \forall \widetilde{sres}, \widetilde{res}.\varepsilon :: \ \widetilde{sres} = \overline{sres} \ \wedge \ \widetilde{res}.\varepsilon = \overline{res}.\varepsilon \ \Rightarrow \ \ldots \rangle$
A3. Modification constraints of ε:	
$v{:}\langle \forall s :: \ \grave{\varepsilon}[s] = \acute{\varepsilon}[s] \ \vee \ \ldots \rangle$	$v{:}\langle \forall s :: \ \mathcal{F}.\varepsilon(\grave{sres}, \grave{res}.\varepsilon)[s] = \mathcal{F}.\varepsilon(\acute{sres}, \acute{res}.\varepsilon)[s] \ \vee \ \ldots \rangle$
A4. Modification constraints of δ and $res.\varepsilon$:	
$v{:}\langle \forall s :: \ \grave{\delta}[s] = \acute{\delta}[s] \ \vee \ \ldots \rangle$	$v{:}\langle \forall s :: \ \grave{res}.\varepsilon[s] = \acute{res}.\varepsilon[s] \ \vee \ \ldots \rangle$
A5. Pointwise axiom for $\mathcal{F}.\varepsilon$:	
no pointwise axiom for $\mathcal{F}.\varepsilon$	pointwise axiom for $\mathcal{F}.\varepsilon$
A6. Pointwise axiom for $\mathcal{F}.a$:	
includes δ argument to $\mathcal{F}.a$	does not includes δ argument to $\mathcal{F}.a$
A7. Rep axioms for a:	
includes δ argument to $\mathcal{F}.a$	does not includes δ argument to $\mathcal{F}.a$

Figure 11: An illustration of the syntactic differences between $VC_H(m, C)$ and $VC_E(m, C)$.

$$
\begin{array}{ll}
\text{A0:} & \mathcal{F}.a(s\tilde{r}es, \ldots, \tilde{\varepsilon}, \tilde{\delta}) \\
\text{A1:} & \langle \forall\, s\tilde{r}es, \tilde{\delta}, \tilde{\varepsilon} \ :: \ \ldots \rangle \\
\text{A2:} & \langle \forall\, s\tilde{r}es, \tilde{\delta}, \tilde{\varepsilon} \ :: \ s\tilde{r}es = s\bar{r}es \ \wedge \ \tilde{\delta} = \bar{\delta} \ \wedge \ \tilde{\varepsilon} = \bar{\varepsilon} \ \Rightarrow \ \ldots \rangle \\
\text{A3:} & v{:}\langle \forall\, s \ :: \ \grave{\varepsilon}[s] = \acute{\varepsilon}[s] \ \vee \ \ldots \rangle \\
\text{A4:} & v{:}\langle \forall\, s \ :: \ \grave{\delta}[s] = \acute{\delta}[s] \ \vee \ \ldots \rangle \\
\text{A5:} & \text{no pointwise axiom for } \mathcal{F}.\varepsilon \\
\text{A6:} & \langle \forall\, t, s\grave{r}es, s\acute{r}es, \ldots, \grave{\varepsilon}, \acute{\varepsilon}, \grave{\delta}, \acute{\delta} \ :: \\
 & \qquad\qquad s\grave{r}es[t] = s\acute{r}es[t] \ \wedge \ \ldots \ \wedge \ \grave{\varepsilon}[t] = \acute{\varepsilon}[t] \ \wedge \ \grave{\delta}[t] = \acute{\delta}[t] \ \Rightarrow \\
 & \qquad\qquad \mathcal{F}.a(s\grave{r}es, \ldots, \grave{\varepsilon}, \grave{\delta})[t] = \mathcal{F}.a(s\acute{r}es, \ldots, \acute{\varepsilon}, \acute{\delta})[t] \,\rangle \\
\text{A7:} & \langle \forall\, t{:}\, T, sres, \ldots, \varepsilon, \delta \ :: \ t \neq \mathbf{nil} \ \Rightarrow \ \mathcal{F}.a(sres, \ldots, \varepsilon, \delta)[t] = \ldots \rangle
\end{array}
$$

Figure 12: A sketch of the initial working formula, $VC_H(m, C)$.

working formula in Figures 12 through 16. Figure 12 contains our first sketch of the discrepancy formulas from the left column of Figure 11.

(Alternatively, but more verbosely, instead of showing a sketch of each working formula, we could give a grammar that describes the formula. Or even more verbosely, we could give the full recipe for generating the formula, more or less duplicating Part II for each working formula. We find our sketches to convey the essential information more concisely.)

Step 0: From Figure 12 to Figure 13. In our first transformation of the working formula, we replace $\mathcal{F}.a$, for every a that depends on ε , by a function that doesn't actually make use of the δ argument. This transformation is justified by the fact that everywhere brackets quantify over function symbols like $\mathcal{F}.a$: Let Q denote the current working formula. We consider the case where abstract variable a has exactly three dependencies, f , ε , and δ ; other cases are straightforward and omitted. We calculate,

$$
\begin{array}{ll}
 & [Q] \\
\Rightarrow & \qquad \{ \ \text{instantiate the universally quantified } \mathcal{F}.a \ \} \\
 & [Q(\mathcal{F}.a := \langle \lambda\, sres, res.a, f, \varepsilon, \delta \ :: \ \mathcal{F}.a(sres, res.a, f, \varepsilon) \rangle)]
\end{array}
$$

After applying this calculation to the working formula for every a that directly depends on ε and then applying β-conversion (unfold) on the λ-expressions (that is, applying the λ-expressions to their arguments), the working formula will have the form sketched in Figure 13.

$$
\begin{array}{ll}
\text{A0:} & \mathcal{F}.a(s\widetilde{re}s, \ldots, \tilde{\varepsilon}) \\
\text{A1:} & \langle \forall\, s\widetilde{re}s, \tilde{\delta}, \tilde{\varepsilon} \ :: \ \ldots \rangle \\
\text{A2:} & \langle \forall\, s\widetilde{re}s, \tilde{\delta}, \tilde{\varepsilon} \ :: \ s\widetilde{re}s = s\bar{re}s \ \wedge \ \tilde{\delta} = \bar{\delta} \ \wedge \ \tilde{\varepsilon} = \bar{\varepsilon} \ \Rightarrow \ \ldots \rangle \\
\text{A3:} & v{:}\langle \forall\, s \ :: \ \grave{\varepsilon}[s] = \acute{\varepsilon}[s] \ \vee \ \ldots \rangle \\
\text{A4:} & v{:}\langle \forall\, s \ :: \ \grave{\delta}[s] = \acute{\delta}[s] \ \vee \ \ldots \rangle \\
\text{A5:} & \text{no pointwise axiom for } \mathcal{F}.\varepsilon \\
\text{A6:} & \langle \forall\, t, s\grave{re}s, s\acute{re}s, \ldots, \grave{\varepsilon}, \acute{\varepsilon}, \grave{\delta}, \acute{\delta} \ :: \\
& \qquad\qquad s\grave{re}s[t] = s\acute{re}s[t] \ \wedge \ \ldots \ \wedge \ \grave{\varepsilon}[t] = \acute{\varepsilon}[t] \ \wedge \ \grave{\delta}[t] = \acute{\delta}[t] \ \Rightarrow \\
& \qquad\qquad \mathcal{F}.a(s\grave{re}s, \ldots, \grave{\varepsilon})[t] = \mathcal{F}.a(s\acute{re}s, \ldots, \acute{\varepsilon})[t] \ \rangle \\
\text{A7:} & \langle \forall\, t{:}\, T, sres, \ldots, \varepsilon, \delta \ :: \ t \neq \mathbf{nil} \ \Rightarrow \ \mathcal{F}.a(sres, \ldots, \varepsilon)[t] = \ldots \rangle
\end{array}
$$

Figure 13: A sketch of the working formula after substituting new functions $\mathcal{F}.a$ for those in Figure 12, which affects lines A0, A6, and A7.

Step 1: From Figure 13 to Figure 14. It is now easy to tidy up any rep axioms for a and the pointwise axiom for $\mathcal{F}.a$. Let's do each in turn.

Consider a rep axiom for any a that depends on ε. If a depends on f, ε, and δ in H, the rep axiom has the following shape in the current working formula:

$$\langle \forall\, t{:}\, T, sres, res.a, f, \varepsilon, \delta \ :: \ t \neq \mathbf{nil} \ \Rightarrow \ \mathcal{F}.a(sres, res.a, f, \varepsilon)[t] = e \ \rangle$$

Since e is a user expression in D, it does not mention δ. So the quantification over δ can be dropped, producing the equivalent formula:

$$\langle \forall\, t{:}\, T, sres, res.a, f, \varepsilon \ :: \ t \neq \mathbf{nil} \ \Rightarrow \ \mathcal{F}.a(sres, res.a, f, \varepsilon)[t] = e \ \rangle$$

Cases where a has a different set of dependencies in H are similar and omitted.

Similarly, for any a that depends on f, ε, and δ in H, we calculate from the pointwise axiom for $\mathcal{F}.a$ in the working formula,

$$
\begin{array}{ll}
& \langle \forall\, t, s\grave{re}s, s\acute{re}s, re\grave{s}.a, re\acute{s}.a, \grave{f}, \acute{f}, \grave{\varepsilon}, \acute{\varepsilon}, \grave{\delta}, \acute{\delta} \ :: \\
& \quad s\grave{re}s[t] = s\acute{re}s[t] \ \wedge \ re\grave{s}.a[t] = re\acute{s}.a[t] \ \wedge \\
& \quad \grave{f}[t] = \acute{f}[t] \ \wedge \ \grave{\varepsilon}[t] = \acute{\varepsilon}[t] \ \wedge \ \grave{\delta}[t] = \acute{\delta}[t] \ \Rightarrow \\
& \quad \mathcal{F}.a(s\grave{re}s, re\grave{s}.a, \grave{f}, \grave{\varepsilon})[t] = \mathcal{F}.a(s\acute{re}s, re\acute{s}.a, \acute{f}, \acute{\varepsilon})[t] \ \rangle \\
\Leftarrow & \quad \{ \ \text{weaken antecedent} \ \} \\
& \langle \forall\, t, s\grave{re}s, s\acute{re}s, re\grave{s}.a, re\acute{s}.a, \grave{f}, \acute{f}, \grave{\varepsilon}, \acute{\varepsilon}, \grave{\delta}, \acute{\delta} \ :: \\
& \quad s\grave{re}s[t] = s\acute{re}s[t] \ \wedge \ re\grave{s}.a[t] = re\acute{s}.a[t] \ \wedge \ \grave{f}[t] = \acute{f}[t] \ \wedge \ \grave{\varepsilon}[t] = \acute{\varepsilon}[t] \ \Rightarrow \\
& \quad \mathcal{F}.a(s\grave{re}s, re\grave{s}.a, \grave{f}, \grave{\varepsilon})[t] = \mathcal{F}.a(s\acute{re}s, re\acute{s}.a, \acute{f}, \acute{\varepsilon})[t] \ \rangle \\
= & \quad \{ \ \grave{\delta} \text{ and } \acute{\delta} \text{ do not occur} \ \}
\end{array}
$$

$$
\begin{array}{ll}
\text{A0:} & \mathcal{F}.a(s\tilde{r}es, \ldots, \tilde{\varepsilon}) \\
\text{A1:} & \langle \forall\, s\tilde{r}es, re\tilde{s}.\varepsilon, \tilde{\varepsilon} \ :: \ \ldots \rangle \\
\text{A2:} & \langle \forall\, s\tilde{r}es, re\tilde{s}.\varepsilon, \tilde{\varepsilon} \ :: \ s\tilde{r}es = s\bar{r}es \ \wedge \ re\tilde{s}.\varepsilon = re\bar{s}.\varepsilon \ \wedge \ \tilde{\varepsilon} = \bar{\varepsilon} \ \Rightarrow \ \ldots \rangle \\
\text{A3:} & v{:}\langle \forall\, s \ :: \ \grave{\varepsilon}[s] = \acute{\varepsilon}[s] \ \vee \ \ldots \rangle \\
\text{A4:} & v{:}\langle \forall\, s \ :: \ re\grave{s}.\varepsilon[s] = re\acute{s}.\varepsilon[s] \ \vee \ \ldots \rangle \\
\text{A5:} & \text{no pointwise axiom for } \mathcal{F}.\varepsilon \\
\text{A6:} & \langle \forall\, t, s\grave{r}es, s\acute{r}es, \ldots, \grave{\varepsilon}, \acute{\varepsilon} \ :: \\
& \qquad\qquad s\grave{r}es[t] = s\acute{r}es[t] \ \wedge \ \ldots \ \wedge \ \grave{\varepsilon}[t] = \acute{\varepsilon}[t] \ \Rightarrow \\
& \qquad\qquad \mathcal{F}.a(s\grave{r}es, \ldots, \grave{\varepsilon})[t] = \mathcal{F}.a(s\acute{r}es, \ldots, \acute{\varepsilon})[t] \ \rangle \\
\text{A7:} & \langle \forall\, t{:}\, T, sres, \ldots, \varepsilon \ :: \ t \neq \mathbf{nil} \ \Rightarrow \ \mathcal{F}.a(sres, \ldots, \varepsilon)[t] = \ldots \rangle
\end{array}
$$

Figure 14: A sketch of the working formula after tidying up the rep and pointwise axioms and renaming $\tilde{\delta}$ to $re\tilde{s}.\varepsilon$, which affects A1, A2, A4, A6, and A7.

$$
\begin{array}{l}
\langle \forall\, t, s\grave{r}es, s\acute{r}es, re\grave{s}.a, re\acute{s}.a, \grave{f}, \acute{f}, \grave{\varepsilon}, \acute{\varepsilon} \ :: \\
\quad s\grave{r}es[t] = s\acute{r}es[t] \ \wedge \ re\grave{s}.a[t] = re\acute{s}.a[t] \ \wedge \ \grave{f}[t] = \acute{f}[t] \ \wedge \ \grave{\varepsilon}[t] = \acute{\varepsilon}[t] \ \Rightarrow \\
\quad \mathcal{F}.a(s\grave{r}es, re\grave{s}.a, \grave{f}, \grave{\varepsilon})[t] = \mathcal{F}.a(s\acute{r}es, re\acute{s}.a, \acute{f}, \acute{\varepsilon})[t] \ \rangle
\end{array}
$$

Cases where a has a different set of dependencies in H are similar and omitted.

Since pointwise axioms appear only in negative positions (see characteristic V0, page 87), applying these transformations to rep and pointwise axioms in the working formula sketched in Figure 13 preserves its validity.

Having removed occurrences of $\tilde{\delta}$ from rep and pointwise axioms, we now rename each remaining occurrence of $\tilde{\delta}$ to $re\tilde{s}.a$ The resulting formula is shown in Figure 14.

Step 2: From Figure 14 to Figure 15. We are now done with the transformation of δ into $res.\varepsilon$ and so we turn to the transformation of ε into $\mathcal{F}.\varepsilon(sres, res.\varepsilon)$. To this end, we first add the pointwise axiom for $\mathcal{F}.\varepsilon$. Since pointwise axioms occur in negative positions (by characteristic V0, page 87), this transformation weakens the working formula, preserving its validity. Next, we calculate for any predicate Q , possibly including equality antecedents,

$$
\begin{array}{ll}
& \langle \forall\, s\tilde{r}es, re\tilde{s}.\varepsilon, \tilde{\varepsilon} \ :: \ Q \ \rangle \\
\Rightarrow & \qquad \{ \ \text{instantiate } \tilde{\varepsilon} := \mathcal{F}.\varepsilon(s\tilde{r}es, re\tilde{s}.\varepsilon) \quad \} \\
& \langle \forall\, s\tilde{r}es, re\tilde{s}.\varepsilon \ :: \ Q(\tilde{\varepsilon} := \mathcal{F}.\varepsilon(s\tilde{r}es, re\tilde{s}.\varepsilon)) \ \rangle
\end{array}
$$

$$
\begin{array}{ll}
\text{A0:} & \mathcal{F}.a(\widetilde{sres}, \ldots, \mathcal{F}.\varepsilon(\widetilde{sres}, \widetilde{res}.\varepsilon)) \\
\text{A1:} & \langle \forall\, \widetilde{sres}, \widetilde{res}.\varepsilon \ :: \ \ldots \rangle \\
\text{A2:} & \langle \forall\, \widetilde{sres}, \widetilde{res}.\varepsilon \ :: \ \widetilde{sres} = \overline{sres} \ \wedge \ \widetilde{res}.\varepsilon = \overline{res}.\varepsilon \ \wedge \\
& \qquad\qquad\qquad \mathcal{F}.\varepsilon(\widetilde{sres}, \widetilde{res}.\varepsilon) = \mathcal{F}.\varepsilon(\overline{sres}, \overline{res}.\varepsilon) \ \Rightarrow \ \ldots \rangle \\
\text{A3:} & v{:}\langle \forall\, s \ :: \ \mathcal{F}.\varepsilon(\grave{sres}, \grave{res}.\varepsilon)[s] = \mathcal{F}.\varepsilon(\acute{sres}, \acute{res}.\varepsilon)[s] \ \vee \ \ldots \rangle \\
\text{A4:} & v{:}\langle \forall\, s \ :: \ \grave{res}.\varepsilon[s] = \acute{res}.\varepsilon[s] \ \vee \ \ldots \rangle \\
\text{A5:} & \langle \forall\, t, \grave{sres}, \acute{sres}, \grave{res}.\varepsilon, \acute{res}.\varepsilon \ :: \\
& \qquad\qquad \grave{sres}[t] = \acute{sres}[t] \ \wedge \ \grave{res}.\varepsilon[t] = \acute{res}.\varepsilon[t] \ \Rightarrow \\
& \qquad\qquad \mathcal{F}.\varepsilon(\grave{sres}, \grave{res}.\varepsilon)[t] = \mathcal{F}.\varepsilon(\acute{sres}, \acute{res}.\varepsilon)[t] \ \rangle \\
\text{A6:} & \langle \forall\, t, \grave{sres}, \acute{sres}, \ldots, \grave{\varepsilon}, \acute{\varepsilon} \ :: \\
& \qquad\qquad \grave{sres}[t] = \acute{sres}[t] \ \wedge \ \ldots \ \wedge \ \grave{\varepsilon}[t] = \acute{\varepsilon}[t] \ \Rightarrow \\
& \qquad\qquad \mathcal{F}.a(\grave{sres}, \ldots, \grave{\varepsilon})[t] = \mathcal{F}.a(\acute{sres}, \ldots, \acute{\varepsilon})[t] \ \rangle \\
\text{A7:} & \langle \forall\, t{:}\, T, sres, \ldots, \varepsilon \ :: \ t \neq \mathbf{nil} \ \Rightarrow \ \mathcal{F}.a(sres, \ldots, \varepsilon)[t] = \ldots \rangle
\end{array}
$$

Figure 15: A sketch of the working formula after introducing the pointwise axiom for $\mathcal{F}.\varepsilon$ and instantiating the concrete ε with the functionalized abstract ε. These transformations affect A0, A1, A2, A3, and A5.

Since such quantifications appear only in positive positions (by characteristic V1, page 87), applying this transformation to all quantifications, with or without equality antecedents, preserves the validity of the working formula. The resulting working formula is sketched in Figure 15. Because of characteristic V5 (page 88), the arguments we have supplied to $\mathcal{F}.\varepsilon$ are indeed the ones supplied to $\mathcal{F}.\varepsilon$ in $VC_E(m, C)$.

Step 3: From Figure 15 to Figure 16. Almost there. Now we just need to get rid of the conjunct

$$\mathcal{F}.\varepsilon(\widetilde{sres}, \widetilde{res}.\varepsilon) = \mathcal{F}.\varepsilon(\overline{sres}, \overline{res}.\varepsilon)$$

from quantifications with equality antecedents. We calculate,

$$
\begin{array}{ll}
& \widetilde{sres} = \overline{sres} \ \wedge \ \widetilde{res}.\varepsilon = \overline{res}.\varepsilon \ \wedge \ \mathcal{F}.\varepsilon(\widetilde{sres}, \widetilde{res}.\varepsilon) = \mathcal{F}.\varepsilon(\overline{sres}, \overline{res}.\varepsilon) \\
= & \qquad \{ \ \text{third conjunct follows from the first two} \ \} \\
& \widetilde{sres} = \overline{sres} \ \wedge \ \widetilde{res}.\varepsilon = \overline{res}.\varepsilon
\end{array}
$$

Applying this transformation to the working formula from Figure 15, we arrive at the formula sketched in Figure 16, which is exactly the formula $VC_E(m, C)$

$$
\begin{array}{ll}
\text{A0:} & \mathcal{F}.a(\widetilde{sres}, \ldots, \mathcal{F}.\varepsilon(\widetilde{sres}, \widetilde{res}.\varepsilon)) \\
\text{A1:} & \langle \forall\, \widetilde{sres}, \widetilde{res}.\varepsilon \ :: \ \ldots \rangle \\
\text{A2:} & \langle \forall\, \widetilde{sres}, \widetilde{res}.\varepsilon \ :: \ \widetilde{sres} = \overline{sres} \ \wedge \ \widetilde{res}.\varepsilon = \overline{res}.\varepsilon \ \Rightarrow \ \ldots \rangle \\
\text{A3:} & v{:}\langle \forall\, s \ :: \ \mathcal{F}.\varepsilon(\grave{sres}, \grave{res}.\varepsilon)[s] = \mathcal{F}.\varepsilon(\acute{sres}, \acute{res}.\varepsilon)[s] \ \vee \ \ldots \rangle \\
\text{A4:} & v{:}\langle \forall\, s \ :: \ \grave{res}.\varepsilon[s] = \acute{res}.\varepsilon[s] \ \vee \ \ldots \rangle \\
\text{A5:} & \langle \forall\, t, \grave{sres}, \acute{sres}, \grave{res}.\varepsilon, \acute{res}.\varepsilon \ :: \\
& \qquad\qquad \grave{sres}[t] = \acute{sres}[t] \ \wedge \ \grave{res}.\varepsilon[t] = \acute{res}.\varepsilon[t] \ \Rightarrow \\
& \qquad\qquad \mathcal{F}.\varepsilon(\grave{sres}, \grave{res}.\varepsilon)[t] = \mathcal{F}.\varepsilon(\acute{sres}, \acute{res}.\varepsilon)[t] \ \rangle \\
\text{A6:} & \langle \forall\, t, \grave{sres}, \acute{sres}, \ldots, \grave{\varepsilon}, \acute{\varepsilon} \ :: \\
& \qquad\qquad \grave{sres}[t] = \acute{sres}[t] \ \wedge \ \ldots \ \wedge \ \grave{\varepsilon}[t] = \acute{\varepsilon}[t] \ \Rightarrow \\
& \qquad\qquad \mathcal{F}.a(\grave{sres}, \ldots, \grave{\varepsilon})[t] = \mathcal{F}.a(\acute{sres}, \ldots, \acute{\varepsilon})[t] \ \rangle \\
\text{A7:} & \langle \forall\, t{:}\, T, sres, \ldots, \varepsilon \ :: \ t \neq \mathbf{nil} \ \Rightarrow \ \mathcal{F}.a(sres, \ldots, \varepsilon)[t] = \ldots \rangle \ldots
\end{array}
$$

Figure 16: A sketch of the final working formula, obtained from Figure 15 by applying the pointwise axiom for $\mathcal{F}.\varepsilon$, which affects A2.

sketched in the right column of Figure 11. So in summary, we have transformed formula $VC_H(m, C)$ into formula $VC_E(m, C)$ by applying validity-preserving steps, thereby completing the proof of Soundness Lemma D. ∎

The third lemma of this section is a simple corollary of the other two, allowing the field ε to be either concrete or abstract.

Soundness Lemma E. Let D and E be scopes containing an implementation C for a method m. If D and E differ only in that E additionally contains a declaration of a (concrete or abstract) field ε and some number of dependencies on ε, then

$$[VC_D(m, C)] \ \Rightarrow \ [VC_E(m, C)]$$

Proof. Follows directly from Soundness Lemmas C and D. ∎

A17 Multiple field-declaration discrepancies

In this section, we present the last of the specializations of the Soundness Theorem. The specialization concerns scopes that differ only in their field and dependency declarations. We prove this specialization from Soundness Lemma E and induction.

Soundness Lemma F. For any scopes D and E, containing an implementation C for a method m, if D and E differ only in their field and dependency declarations, then

$$D \subseteq E \;\wedge\; \mathit{Vis}(D, E) \;\wedge\; \mathit{TopDown}(D, E) \;\wedge\; [\mathit{VC}_D(m, C)] \;\Rightarrow\; [\mathit{VC}_E(m, C)]$$

Proof. Let D and E be scopes satisfying the antecedent. Using Soundness Lemma E, we now prove the lemma by induction over the number of fields declared in E and not in D. We consider two cases.

CASE D and E coincide in their field declarations: Due to $\mathit{Vis}(D, E)$, we have that D and E coincide also in their dependency declarations. Hence, D and E are equal, so the lemma follows trivially.

CASE E contains some field declaration not in D: Since D and E differ only in their field and dependency declarations, the only uses in E of a field not in D are in dependency declarations (since D is a scope, which is closed). For any declaration

depends $a[t\colon T]$ **on** $f[t]$

in E where $a \in E \setminus D$, $\mathit{TopDown}(D, E)$ implies that $f \in E \setminus D$. So, fields in $E \setminus D$ can depend only on other fields in $E \setminus D$. Since E is a scope, it is cycle-free, and thus there is some field among those in $E \setminus D$, call it ε, such that there is no field f for which $\varepsilon\ \mathbf{on}_E\ f$.

Let H be the set E minus the declaration of ε and minus any declaration of the form

depends $a[t\colon T]$ **on** $\varepsilon[t]$

for any a, t, and T. We have that H is a scope, that $D \subseteq H$ and $H \subseteq E$, and $\mathit{Vis}(D, H)$, $\mathit{TopDown}(D, H)$, $\mathit{Vis}(H, E)$, and $\mathit{TopDown}(H, E)$. We calculate,

$$\begin{array}{ll}
 & [\mathit{VC}_D(m, C)] \\
\Rightarrow & \qquad \{\ \text{induction hypothesis, with } D, E := D, H\ \} \\
 & [\mathit{VC}_H(m, C)] \\
\Rightarrow & \qquad \{\ \text{Soundness Lemma E, with } D, E := H, E\ \} \\
 & [\mathit{VC}_E(m, C)]
\end{array}$$

Since under our premise $D \subseteq E$, the two cases we have considered are exhaustive, we have now proved the lemma. ∎

A18 Proving the Soundness Theorem

In this section, we prove the Soundness Theorem from the Soundness Lemmas of the previous three sections.

Proof of Soundness Theorem. Let D and E be scopes satisfying the antecedent of the theorem. We define sets of declarations G and H such that G is D union the type declarations of E and H is G union the field and dependency declarations of E. Sets H and E hence differ only in their rep, method specification, and method implementation declarations, and we have

$$D \subseteq G \subseteq H \subseteq E$$

Furthermore, G and H are scopes, and the following properties hold: $Vis(D, G)$, $TopDown(D, G)$, $Vis(G, H)$, $TopDown(G, H)$, $Vis(H, E)$, and $TopDown(H, E)$. We calculate,

$$
\begin{array}{ll}
 & [VC_D(m, C)] \\
\Rightarrow & \quad \{ \text{ Soundness Lemma A with } D, E := D, G \ \} \\
 & [VC_G(m, C)] \\
\Rightarrow & \quad \{ \text{ Soundness Lemma F with } D, E := G, H \ \} \\
 & [VC_H(m, C)] \\
\Rightarrow & \quad \{ \text{ Soundness Lemma B with } D, E := H, E \ \} \\
 & [VC_E(m, C)]
\end{array}
$$

∎

So, we have proved the Soundness Theorem from the Soundness Lemmas, but we're not done yet, because we have deferred the proof of Soundness Lemma C. In the remaining sections of Part III of this appendix, we prove Soundness Lemma C.

A19 Consequences of the modularity requirements

In this section, we prove three lemmas that follow from the definitions of Vis and $TopDown$.

Lemma. Let scopes D and E satisfy $D \subseteq E$, $Vis(D, E)$, and $TopDown(D, E)$. Then,

$$\langle \forall x, y :: x \in D \ \wedge \ y \in D \ \wedge \ x \ \mathbf{on}_E^* \ y \ \Rightarrow \ x \ \mathbf{on}_D^* \ y \rangle \tag{29}$$

That is, the larger scope introduces no new dependencies of old variables.

Proof. To prove this lemma, we first formalize "reflexive, transitive closure of $\mathbf{on}_D$": for any scope D, any x and y, and any natural number n, we define

$$\begin{array}{lll} x\ \mathbf{on}_D^0\ y & \equiv & x \in D\ \wedge\ x = y \\ x\ \mathbf{on}_D^{n+1}\ y & \equiv & \langle\, \exists\, a\ ::\ x\ \mathbf{on}_D^n\ a\ \wedge\ a\ \mathbf{on}_D\ y\,\rangle \end{array}$$

We thus have

$$\langle\, \forall\, D, x, y\ ::\ x\ \mathbf{on}_D^*\ y\ \equiv\ \langle\, \exists\, n\ ::\ x\ \mathbf{on}_D^n\ y\,\rangle\rangle \tag{30}$$

and

$$\langle\, \forall\, D, x, a, y\ ::\ x\ \mathbf{on}_D^*\ a\ \wedge\ a\ \mathbf{on}_D\ y\ \Rightarrow\ x\ \mathbf{on}_D^*\ y\,\rangle \tag{31}$$

Using the new notation, we rewrite the lemma as follows:

$$\begin{array}{ll} & \langle\, \forall\, x, y\ ::\ x \in D\ \wedge\ y \in D\ \wedge\ x\ \mathbf{on}_E^*\ y\ \Rightarrow\ x\ \mathbf{on}_D^*\ y\,\rangle \\ = & \quad \{\ (30)\ \} \\ & \langle\, \forall\, x, y\ ::\ x \in D\ \wedge\ y \in D\ \wedge\ \langle\, \exists\, n\ ::\ x\ \mathbf{on}_E^n\ y\,\rangle\ \Rightarrow\ x\ \mathbf{on}_D^*\ y\,\rangle \\ = & \quad \{\ \text{predicate calculus}\ \} \\ & \langle\, \forall\, x, y, n\ ::\ x \in D\ \wedge\ y \in D\ \wedge\ x\ \mathbf{on}_E^n\ y\ \Rightarrow\ x\ \mathbf{on}_D^*\ y\,\rangle \end{array}$$

We prove this formula by induction on n.
CASE $n = 0$:

$$\begin{array}{ll} & x \in D\ \wedge\ y \in D\ \wedge\ x\ \mathbf{on}_E^0\ y \\ \Rightarrow & \quad \{\ \ \mathbf{on}_E^0\ \ \} \\ & x \in D\ \wedge\ x = y \\ \Rightarrow & \quad \{\ \ \mathbf{on}_D^0\ \text{ and }\ \mathbf{on}_D^*\ \ \} \\ & x\ \mathbf{on}_D^*\ y \end{array}$$

CASE $n = k + 1$:

$$\begin{array}{ll} & x \in D\ \wedge\ y \in D\ \wedge\ x\ \mathbf{on}_E^{k+1}\ y \\ = & \quad \{\ \ \mathbf{on}_E^{k+1}\ \ \} \\ & x \in D\ \wedge\ y \in D\ \wedge\ \langle\, \exists\, a\ ::\ x\ \mathbf{on}_E^k\ a\ \wedge\ a\ \mathbf{on}_E\ y\,\rangle \\ \Rightarrow & \quad \{\ \text{by } \mathit{TopDown}(D, E),\ y \in D\ \wedge\ a\ \mathbf{on}_E\ y\ \Rightarrow\ a \in D\ \ \} \\ & x \in D\ \wedge\ y \in D\ \wedge\ \langle\, \exists\, a\ ::\ a \in D\ \wedge\ x\ \mathbf{on}_E^k\ a\ \wedge\ a\ \mathbf{on}_E\ y\,\rangle \\ \Rightarrow & \quad \{\ \text{by } \mathit{Vis}(D, E),\ a \in D\ \wedge\ y \in D\ \wedge\ a\ \mathbf{on}_E\ y\ \Rightarrow\ a\ \mathbf{on}_D\ y\ \ \} \\ & x \in D\ \wedge\ y \in D\ \wedge\ \langle\, \exists\, a\ ::\ a \in D\ \wedge\ x\ \mathbf{on}_E^k\ a\ \wedge\ a\ \mathbf{on}_D\ y\,\rangle \\ \Rightarrow & \quad \{\ \text{by induction hypothesis, } x \in D \wedge a \in D \wedge x\ \mathbf{on}_E^k\ a\ \Rightarrow\ x\ \mathbf{on}_D^*\ a\ \ \} \end{array}$$

$$
\begin{array}{ll}
& x \in D \ \wedge \ y \in D \ \wedge \ \langle \exists a \ :: \ a \in D \ \wedge \ x \ \mathbf{on}_D^* \ a \ \wedge \ a \ \mathbf{on}_D \ y \ \rangle \\
\Rightarrow & \qquad \{ \ \ (31)\text{:} \ x \ \mathbf{on}_D^* \ a \ \wedge \ a \ \mathbf{on}_D \ y \ \Rightarrow \ x \ \mathbf{on}_D^* \ y \ \ \} \\
& x \in D \ \wedge \ y \in D \ \wedge \ \langle \exists a \ :: \ a \in D \ \wedge \ x \ \mathbf{on}_D^* \ y \ \rangle \\
\Rightarrow & \qquad \{ \ \ \text{predicate calculus} \ \ \} \\
& x \ \mathbf{on}_D^* \ y
\end{array}
$$

That proves lemma (29). ∎

Lemma. Let scopes D and E satisfy $D \subseteq E$, $\mathit{Vis}(D, E)$, and $\mathit{TopDown}(D, E)$. Then, for any modifies list w in D and any field or residue variable x in D,

$$
\langle \forall e \ :: \ x[e] \in cl_D(w) \ \equiv \ x[e] \in cl_E(w) \ \rangle \tag{32}
$$

Proof. The proof is by mutual implication. One direction is that cl_D is monotonic with respect to D, which follows from that $\mathbf{on}_D$, and thus also $\mathbf{on}_D^*$, is monotonic in D:

$$
\begin{array}{ll}
& x[e] \in cl_D(w) \\
= & \qquad \{ \ \ (20)\text{: definition of } cl_D \ \ \} \\
& \langle \exists f \ :: \ f[e] \in w \ \wedge \ f \ \mathbf{on}_D^* \ x \ \rangle \\
\Rightarrow & \qquad \{ \ \ D \subseteq E \ \wedge \ f \ \mathbf{on}_D^* \ x \ \Rightarrow \ f \ \mathbf{on}_E^* \ x \ \ \} \\
& \langle \exists f \ :: \ f[e] \in w \ \wedge \ f \ \mathbf{on}_E^* \ x \ \rangle \\
= & \qquad \{ \ \ (20)\text{: definition of } cl_E \ \ \} \\
& x[e] \in cl_E(w)
\end{array}
$$

For the other direction,

$$
\begin{array}{ll}
& x[e] \in cl_E(w) \\
= & \qquad \{ \ \ (20)\text{: definition of } cl_E \ \ \} \\
& \langle \exists f \ :: \ f[e] \in w \ \wedge \ f \ \mathbf{on}_E^* \ x \ \rangle \\
\Rightarrow & \qquad \{ \ \ f[e] \in w \ \Rightarrow \ f \in D\text{, and} \\
& \qquad \quad \text{by lemma (29), } f \in D \ \wedge \ x \in D \ \wedge \ f \ \mathbf{on}_E^* \ x \ \Rightarrow \ f \ \mathbf{on}_D^* \ x \ \ \} \\
& \langle \exists f \ :: \ f[e] \in w \ \wedge \ f \ \mathbf{on}_D^* \ x \ \rangle \\
= & \qquad \{ \ \ (20)\text{: definition of } cl_D \ \ \} \\
& x[e] \in cl_D(w)
\end{array}
$$

That proves lemma (32). ∎

The third lemma is a simple corollary of the second lemma:

Lemma. Let scopes D and E satisfy $D \subseteq E$, $Vis(D, E)$, and $TopDown(D, E)$. Then, for any modifies list w in D, any field or residue variable x in D, and any dummy s,

$$modpoint_D(x, w, s) \;=\; modpoint_E(x, w, s) \tag{33}$$

Proof. We calculate,

$$\begin{array}{ll}
 & modpoint_D(x, w, s) \\
= & \qquad \{ \;\; (22)\text{: definition of } modpoint_D \;\; \} \\
 & \langle \bigvee e \;\|\; x[e] \in cl_D(w) \;::\; s = \grave{e} \,\rangle \\
= & \qquad \{ \;\; \text{lemma (32)} \;\; \} \\
 & \langle \bigvee e \;\|\; x[e] \in cl_E(w) \;::\; s = \grave{e} \,\rangle \\
= & \qquad \{ \;\; (22)\text{: definition of } modpoint_E \;\; \} \\
 & modpoint_E(x, w, s)
\end{array}$$

■

A20 A property about modification points

In this section, we prove a lemma about modification points.

Lemma. For any scope D and modifies list w in D, let f be a field not listed in w (that is, for all e, $f[e] \notin w$). Then,

$$[modcon_D(w, f, f) \;\Rightarrow\; modcon_D(w, f, sres)] \tag{34}$$

Proof. For any user expression e, we calculate,

$$\begin{array}{ll}
 & f[e] \in cl_D(w) \\
= & \qquad \{ \;\; (20)\text{: definition of } cl_D \;\; \} \\
 & \langle \exists g \;::\; g[e] \in w \;\wedge\; g \;\mathbf{on}^*_D\, f \,\rangle \\
= & \qquad \{ \;\; f \text{ is not listed in } w\text{, so } f[e] \notin w \;\; \} \\
 & \langle \exists g \;::\; g[e] \in w \;\wedge\; g \;\mathbf{on}^*_D\, f \;\wedge\; g \neq f \,\rangle \\
\Rightarrow & \qquad \{ \;\; \text{definitions of } \mathbf{on}^*_D \text{ and } \mathbf{on}_D \;\; \} \\
 & \langle \exists g \;::\; g[e] \in w \;\wedge\; g \text{ is abstract} \,\rangle \\
\Rightarrow & \qquad \{ \;\; \text{definitions of } \mathbf{on}_D \text{ and } \mathbf{on}^*_D \;\; \} \\
 & \langle \exists g \;::\; g[e] \in w \;\wedge\; g \;\mathbf{on}^*_D\, sres \,\rangle \\
= & \qquad \{ \;\; (20)\text{: definition of } cl_D \;\; \} \\
 & sres[e] \in cl_D(w)
\end{array}$$

Therefore,

$$
\begin{array}{ll}
 & modpoint_D(f, w, s) \\
= & \quad \{ \ \text{(22): definition of } modpoint_D \ \} \\
 & \langle \bigvee e \mid f[e] \in cl_D(w) \ :: \ s = \grave{e} \ \rangle \\
\Rightarrow & \quad \{ \ \text{calculation above} \ \} \\
 & \langle \bigvee e \mid sres[e] \in cl_D(w) \ :: \ s = \grave{e} \ \rangle \\
= & \quad \{ \ \text{(22): definition of } modpoint_D \ \} \\
 & modpoint_D(sres, w, s)
\end{array}
$$

Thus, we prove lemma (34) as follows:

$$
\begin{array}{ll}
 & modcon_D(w, f, f) \\
= & \quad \{ \ \text{(24): definition of } modcon_D \ \} \\
 & w{:}\langle \forall s \ :: \ F_D(\grave{f}[s] = \acute{f}[s] \ \vee \ modpoint_D(f, w, s)) \ \rangle \\
= & \quad \{ \ \text{calculation above, and (19): monotonicity of } F_D \ \} \\
 & w{:}\langle \forall s \ :: \ F_D(\grave{f}[s] = \acute{f}[s] \ \vee \ modpoint_D(sres, w, s)) \ \rangle \\
= & \quad \{ \ \text{(24): definition of } modcon_D \ \} \\
 & modcon_D(w, f, sres)
\end{array}
$$

■

A21 Properties of liberal preconditions

We now start a three-section journey that culminates in a lemma at the heart of our soundness proof, regarding liberal preconditions, individual residue variables, and variables not in scope.

In this section, we define a variation of weakest liberal preconditions that ignores the possibility of commands going wrong, *ultra-liberal preconditions*, written ulp, and develop some properties of wlp and ulp.

For any scope D, command C in D, and vanilla expression Q in D that contains no free occurrences of post-adorned fields or residue variables, the vanilla expression $ulp_D(C, Q)$ characterizes those pre-states ps of C such that

- every terminating execution of C from ps ends in a post-state that satisfies Q.

Note that unlike $wlp_D(C, Q)$, $ulp_D(C, Q)$ does not guarantee that executions do not go wrong (*cf.* Section A8). The fact that this is the only difference between wlp_D and ulp_D is captured by the following relation: for any D, C, and Q,

$$[wlp_D(C, Q) \ \equiv \ wlp_D(C, true) \ \wedge \ ulp_D(C, Q)] \tag{35}$$

This relation is in fact the same relation as between Dijkstra's weakest preconditions, which account for non-termination, and weakest liberal preconditions, which ignore non-termination [9].

We define $ulp_D(C, Q)$ inductively over the syntactic structure of C. Except for the assert command and the method call command, ulp_D is defined like wlp_D in Section A8, but with occurrences of wlp_D replaced by ulp_D. For the assert command, we define

$$ulp_D(\textbf{assert}\ e,\ Q) \ = \ true \ \wedge \ Q$$

And for any method m whose specification, after renaming its parameter to a fresh dummy t, is

$$\textbf{method}\ m(t\colon T)\ \textbf{requires}\ p\ \textbf{modifies}\ w\ \textbf{ensures}\ q$$

we define

$$\begin{array}{l}
ulp_D(\textbf{call}\ m(e),\ Q) \ = \\
\quad \langle \forall\, t \ :: \ t = F_D(e) \ \Rightarrow \\
\qquad true \wedge \\
\qquad \langle \forall\, \ddot{zz} \ :: \ \ddot{zz} = \grave{zz} \ \Rightarrow \ \langle \forall\, \grave{zz} \ :: \ \grave{zz} = zz \ \Rightarrow \\
\qquad\quad \langle \forall\, \acute{zz} \ :: \ F_D(q) \ \wedge \ mc_D(w) \ \Rightarrow \\
\qquad\qquad \langle \forall\, \grave{zz} \ :: \ \grave{zz} = \ddot{zz} \ \Rightarrow \ \langle \forall\, zz \ :: \ zz = \acute{zz} \ \Rightarrow \ Q \,\rangle\rangle\rangle\rangle\rangle\rangle
\end{array}$$

where zz is the list of concrete fields and residue variables in D, and $\ddot{zz}$ denotes fresh adornments of zz.

From the definitions of ulp_D and wlp_D, one can easily prove property (35). A simple consequence of (35) is: for any D, C, and Q,

$$[wlp_D(C, Q) \ \Rightarrow \ ulp_D(C, Q)] \tag{36}$$

An important property of ulp_D is that it is conjunctive in its second argument: for any scope D, command C in D, and predicates $Q0$ and $Q1$,

$$[ulp_D(C, Q0 \ \wedge \ Q1) \ \equiv \ ulp_D(C, Q0) \ \wedge \ ulp_D(C, Q1)] \tag{37}$$

This property can easily be proved from the definition of ulp_D.

Similarly, one can prove from the definition of wlp_D that wlp_D also is conjunctive. Furthermore, conjunctivity implies monotonicity [10], so we have that wlp_D is monotonic in its second argument. That is, for any scope D, command C in D, and predicates $Q0$ and $Q1$,

$$[Q0 \ \Rightarrow \ Q1] \ \Rightarrow \ [wlp_D(C, Q0) \ \Rightarrow \ wlp_D(C, Q1)] \tag{38}$$

Another property that follows from (35) and (37) is: for any scope D, command C, and predicates $Q0$ and $Q1$,

$$[wlp_D(C, Q0 \ \wedge \ Q1) \ \equiv \ wlp_D(C, Q0) \ \wedge \ ulp_D(C, Q1)] \tag{39}$$

We end this section by proving one more lemma about ulp. This lemma, which we will use a couple of sections from now, lets us restrict our attention to $\square$-free commands in certain cases.

Lemma. For any scope D, command C in D, and sufficiently large set of fresh variables ss, there exists a non-negative integer n and a set of $\square$-free commands $\{ j \mid 0 \le j < n :: C_j \}$ such that for any predicate Q that has no free occurrences of any variable in ss,

$$[ulp_D(C, Q) \ \equiv \ \langle \bigwedge j \mid 0 \le j < n \ :: \ \langle \forall ss \ :: \ ulp_D(C_j, Q) \rangle \rangle] \tag{40}$$

Remark: more precisely, the set ss is "sufficiently large" when it contains at least one fresh variable for every **var** ... **end** command in C.

Proof. We prove the lemma by induction over the structure of commands. We consider four cases.

CASE $\square$-free commands: For commands that are already $\square$-free (including the inherently $\square$-free commands simple assignment, field update, assert, assume, and method call), the lemma follows trivially (with $n, C_0 := 1, C$).

CASE **var** s **in** C **end**: Let t be one of the fresh variables in ss, and let tt denote the rest of the variables. We calculate,

$$\begin{array}{ll}
 & ulp_D(\mathbf{var}\ s\ \mathbf{in}\ C\ \mathbf{end}, Q) \\
= & \qquad \{ \ \text{rename } s \text{ to } t \text{ in } C \text{ and call the result } C' \ \} \\
 & ulp_D(\mathbf{var}\ t\ \mathbf{in}\ C'\ \mathbf{end}, Q) \\
= & \qquad \{ \ ulp_D \ \} \\
 & \langle \forall t \ :: \ ulp_D(C', Q) \rangle \\
= & \qquad \{ \ \text{induction hypothesis, with } C, ss := C', tt \ \} \\
 & \langle \forall t \ :: \ \langle \bigwedge j \mid 0 \le j < n \ :: \ \langle \forall tt \ :: \ ulp_D(C_j, Q) \rangle \rangle \rangle \\
= & \qquad \{ \ \text{predicate calculus} \ \} \\
 & \langle \bigwedge j \mid 0 \le j < n \ :: \ \langle \forall t, tt \ :: \ ulp_D(C_j, Q) \rangle \rangle
\end{array}$$

CASE $C0 \ \square \ C1$: Divide ss into two sufficiently large parts, tt and uu. Then,

$$\begin{array}{ll}
 & ulp_D(C0 \ \square \ C1, Q) \\
= & \qquad \{ \ ulp_D \ \}
\end{array}$$

$$
\begin{array}{ll}
& ulp_D(C0, Q) \ \wedge \ ulp_D(C1, Q) \\
= & \quad \{ \ \text{induction hypothesis, with } C, ss := C0, tt \text{ and with} \\
& \quad\quad C, ss := C1, uu \ \} \\
& \langle \bigwedge j \mid 0 \le j < m :: \langle \forall tt :: ulp_D(C_j, Q) \rangle\rangle \wedge \\
& \langle \bigwedge j \mid 0 \le j < n :: \langle \forall uu :: ulp_D(C'_j, Q) \rangle\rangle \\
= & \quad \{ \ \text{for } j\text{: } 0 \le j < n \text{, let } C_{m+j} := C'_j \ \} \\
& \langle \bigwedge j \mid 0 \le j < m :: \langle \forall tt :: ulp_D(C_j, Q) \rangle\rangle \wedge \\
& \langle \bigwedge j \mid 0 \le j < n :: \langle \forall uu :: ulp_D(C_{m+j}, Q) \rangle\rangle \\
= & \quad \{ \ \text{predicate calculus} \ \} \\
& \langle \bigwedge j \mid 0 \le j < m+n :: \langle \forall tt, uu :: ulp_D(C_j, Q) \rangle\rangle
\end{array}
$$

CASE $B\,;\,C$: Divide ss into two sufficiently large parts, tt and uu. Then,

$$
\begin{array}{ll}
& ulp_D(B\,;\,C, Q) \\
= & \quad \{ \ ulp_D \ \} \\
& ulp_D(B, ulp_D(C, Q)) \\
= & \quad \{ \ \text{induction hypothesis, with } C, ss, Q := B, tt, ulp_D(C, Q) \ \} \\
& \langle \bigwedge i \mid 0 \le i < m :: \langle \forall tt :: ulp_D(B_i, ulp_D(C, Q)) \rangle\rangle \\
= & \quad \{ \ \text{induction hypothesis, with } ss := uu \ \} \\
& \langle \bigwedge i \mid 0 \le i < m :: \langle \forall tt :: ulp_D(B_i, \\
& \quad\quad \langle \bigwedge j \mid 0 \le j < n :: \langle \forall uu :: ulp_D(C_j, Q) \rangle\rangle) \rangle\rangle \\
= & \quad \{ \ (37)\text{: } ulp_D \text{ is conjunctive} \ \} \\
& \langle \bigwedge i \mid 0 \le i < m :: \langle \forall tt :: \langle \bigwedge j \mid 0 \le j < n :: \langle \forall uu :: \\
& \quad\quad ulp_D(B_i, ulp_D(C_j, Q)) \rangle\rangle\rangle\rangle \\
= & \quad \{ \ \text{predicate calculus} \ \} \\
& \langle \bigwedge i,j \mid 0 \le i < m \ \wedge \ 0 \le j < n :: \langle \forall tt, uu :: ulp_D(B_i, ulp_D(C_j, Q)) \rangle\rangle \\
= & \quad \{ \ ulp_D \ \} \\
& \langle \bigwedge i,j \mid 0 \le i < m \ \wedge \ 0 \le j < n :: \langle \forall tt, uu :: ulp_D(B_i\,;\,C_j, Q) \rangle\rangle
\end{array}
$$

Since we have now considered an exhaustive set of cases, we have proved the lemma. ■

A22 Chain of Equalities Lemma

In the informal proof sketch for Soundness Lemma C (page 93), we argued that in the large scope, m's body C establishes $modcon_E(w, \varepsilon, \varepsilon)$, where w is the modifies list of m, because in the small scope, C establishes $modcon_D(w, res.a, res.a)$

for each abstract variable a that depends on ε. In the next three lemmas, we make this argument formal.

Each occurrence in C of a method call leads to a modification constraint in the VC of the form

$$\langle \forall s :: res.a_j[s] = res.a_{j+1}[s] \ \vee \ modpoint_D(res.a, v, s) \rangle \tag{41}$$

which is assumed about the call (where v is the modifies list of the called method, and the subscripts j and $j+1$ indicate whatever adornments the VC generator used when it processed that method call). The VC also contains the modification constraint

$$\langle \forall s :: res.a_0[s] = res.a_n[s] \ \vee \ modpoint_D(res.a, w, s) \rangle \tag{42}$$

as a proof obligation (where $res.a_0$ and $res.a_n$ are the initial and final values of $res.a$). The validity of the VC implies that (42) follows from the various instances of (41). An obvious way in which (42) can be proved to follow from the various instances of (41) is to show, for each instance, that $\neg modpoint_D(res.a, v, t)$ holds (in the context of the VC) for each t such that $\neg modpoint_D(res.a, w, t)$. In fact, we claim that this is the *only* circumstance in which (42) follows from the various instances of (41). This claim is the formal version of the fact that there is no way to undo a side effect to a residue variable. The claim is useful because, when combined with the connection between the modification points of ε and those of the abstract variables that depend on ε, it proves that $modcon_E(w, \varepsilon, \varepsilon)$ also follows from the same instances of (41). The essence of the proof of this claim is the Chain of Equalities Lemma, which follows next.

To understand this lemma, it may help to first state a simpler property and then explain in which ways the Chain of Equalities Lemma is more complicated. The simpler property is:

> Let a, b, and c be variables, and let P, K, and L be predicates independent of a, b, and c (that is, the predicates contain no free occurrences of these variables). Then,
>
> $$\begin{array}{l} [\, P \ \wedge \ (a = b \ \vee \ K) \ \wedge \ (b = c \ \vee \ L) \ \Rightarrow \ a = c \,] \\ \Rightarrow \\ [\, P \ \wedge \ (a = b \ \vee \ K) \ \wedge \ (b = c \ \vee \ L) \ \Rightarrow \ \neg K \ \wedge \ \neg L \,] \end{array}$$

This simpler property says that if the equality $a = c$ follows from the antecedent "$P \ \wedge \ \ldots$", then so does $\neg K \ \wedge \ \neg L$. Informally (and somewhat imprecisely), the reason the simpler property is true is that the antecedent contains no information about the variables a, b, and c, and thus the only way to conclude $a = c$ is to first conclude the chain of equalities $a = b$ and $b = c$, which can be achieved only by establishing $\neg K \wedge \neg L$.

The simpler property is true, and it resembles the Chain of Equalities Lemma. But the latter lemma is more complicated in four ways.

First, instead of three variables a, b, and c and two predicates K and L, we have a sequence of $n+1$ variables $a_0, \ldots, a_n$ and n predicates $K_0, \ldots, K_{n-1}$.

Second, P and the K_j's are not entirely independent of the variables, but only "almost independent", a notion that involves a given uninterpreted function f (corresponding to an abstraction function).

Third, instead of equalities between entire variables, the Chain of Equalities Lemma concerns equalities between indexed map variables.

Fourth, the predicate P is replaced by $Q \wedge P$, where Q provides properties of f (corresponding to rep axioms and pointwise axioms).

Here's the definition of almost-independence: for any uninterpreted function symbol f and set of variables aa, we say an expression P is *almost independent* of aa with respect to f when

- the only free occurrences of the aa variables in P appear as the first argument to f, and
- every application of f has a free occurrence of one of the aa variables as its first argument.

We use the following properties of the select and store functions:

$$\begin{array}{l} \langle \forall m, i, j, v :: i = j \Rightarrow store(m, i, v)[j] = v \rangle \wedge \\ \langle \forall m, i, j, v :: i \neq j \Rightarrow store(m, i, v)[j] = m[j] \rangle \end{array} \tag{43}$$

$$\langle \forall m, n :: m = n \equiv \langle \forall s :: m[s] = n[s] \rangle\rangle \tag{44}$$

(Advanced remark: Part IV of this appendix has more to say about property (44).)

Chain of Equalities Lemma. Let f be a 2-argument uninterpreted function symbol, let n be a non-negative integer, and let aa denote a set of $n+1$ variables $\{j \mid 0 \leq j \leq n :: a_j\}$. Define Q as:

$$\begin{array}{l} \langle \forall x, y, z, s :: M(y, z, s) \Rightarrow f(x, y)[s] = z \rangle \wedge \\ \langle \forall w, x, y, z, s :: w[s] = x[s] \wedge N(y, z, s) \Rightarrow f(w, y)[s] = f(x, z)[s] \rangle \end{array}$$

where $M(y, z, s)$ and $N(y, z, s)$ are predicates independent of f and of the variables in aa. Let P be a predicate almost independent of aa with respect to f. For any $j\colon 0 \le j < n$ and any dummy variable s, let $K_j(s)$ be a predicate almost independent of aa (with respect to f). Let S denote

$$Q \ \wedge\ P \ \wedge\ \langle \bigwedge j \mid 0 \le j < n \ :: \ \langle \forall s \ :: \ a_j[s] = a_{j+1}[s] \ \vee\ K_j(s) \rangle \rangle$$

Then, for any $h\colon 0 \le h < n$ and any t not in aa,

$$[\, S \ \Rightarrow\ a_0[t] = a_n[t] \,] \ \Rightarrow\ [\, S \ \Rightarrow\ \neg K_h(t) \,] \tag{45}$$

(End of Lemma.)

Proof. The everywhere brackets say a given formula is valid. Elsewhere in this appendix, we take this to mean that the everywhere brackets implicitly quantify over all variables and uninterpreted function symbols. Equivalently, it means that the formula holds in all models.

We prove the lemma by proving its contrapositive: from a model ϕ that is a counterexample to "$S \Rightarrow \neg K_h(t)$", we show a model ψ that is a counterexample to "$S \Rightarrow a_0[t] = a_n[t]$".

Let ϕ be a counterexample to "$S \Rightarrow \neg K_h(t)$". That is, we have the following properties of ϕ:

$$\begin{aligned} &\phi(Q) \ \wedge\ \phi(P) \ \wedge \\ &\langle \bigwedge j \mid 0 \le j < n \ :: \ \phi(\langle \forall s \ :: \ a_j[s] = a_{j+1}[s] \ \vee\ K_j(s) \rangle) \rangle \ \wedge \\ &\phi(K_h(t)) \end{aligned} \tag{46}$$

Let $F = \phi(f)$, $A_j = \phi(a_j)$ for $j\colon 0 \le j \le n$, and $T = \phi(t)$.

We will now construct a model ψ that falsifies the antecedent of (45). The model ψ will interpret the first $h+1$ variables in aa as the $h+1$ maps B_j, $j\colon 0 \le j \le h$. The most important part of each B_j is its value $B_j[T]$. These values $B_j[T]$ are defined to be any V_j, $j\colon 0 \le j \le h$, satisfying

$$\{ j \mid 0 \le j \le h \ :: \ V_j \} \cap \{ j \mid h < j \le n \ :: \ A_j[T] \} \ = \ \emptyset \tag{47}$$

$$\langle \forall i, j \mid 0 \le i \le j \le h \ :: \ V_i = V_j \ \equiv\ A_i[T] = A_j[T] \rangle \tag{48}$$

Condition (47) requires that the V's be disjoint from the $A_j[T]$'s, and condition (48) requires that the V's be partitioned by equality in the same way as the corresponding $A_j[T]$'s. (Part IV of this appendix argues that V's satisfying (47) and (48) exist in all of the models that matter.)

Having chosen the V's, we can now define

$$B_j = \mathit{store}(A_j, T, V_j)$$

for $j\colon 0 \le j \le h$.

Model ψ will interpret f as a function G, defined by

$$G \;=\; \langle\, \lambda\, x, y \;::\; \textbf{if } x[T] = V_j \text{ for some } j\colon 0 \le j \le h \textbf{ then } F(\mathit{store}(x, T, A_j[T]), y) \textbf{ else } F(x, y) \textbf{ end } \rangle$$

Note that (48) implies that G is well-defined.

For any $j\colon 0 \le j \le h$ and any s,

$$\begin{array}{ll}
 & \mathit{store}(B_j, T, A_j[T])[s] \\
= & \quad \{\ \text{(43): select and store}\ \} \\
 & \textbf{if } s = T \textbf{ then } A_j[T] \textbf{ else } B_j[s] \textbf{ end} \\
= & \quad \{\ B_j = \mathit{store}(A_j, T, V_j)\text{, and (43): select and store with } s \ne T\ \} \\
 & \textbf{if } s = T \textbf{ then } A_j[T] \textbf{ else } A_j[s] \textbf{ end} \\
= & \quad \{\ \text{cases of } \textbf{if} \ldots \textbf{end} \text{ are the same}\ \} \\
 & A_j[s]
\end{array}$$

and thus by select property (44),

$$\mathit{store}(B_j, T, A_j[T]) = A_j \tag{49}$$

Consequently, for any $j\colon 0 \le j \le h$ and any Y,

$$\begin{array}{ll}
 & G(B_j, Y) \\
= & \quad \{\ G \text{ since } B_j[T] = V_j\ \} \\
 & F(\mathit{store}(B_j, T, A_j[T]), Y) \\
= & \quad \{\ (49)\ \} \\
 & F(A_j, Y)
\end{array}$$

which shows

$$G(B_j, Y) = F(A_j, Y) \tag{50}$$

Finally, we define ψ to be the following model:

$$\psi \;=\; \langle\, \lambda\, v \;::\; \textbf{if } v = \text{“}a_j\text{” for some } j\colon 0 \le j \le h \textbf{ then } B_j \textbf{ elsif } v = \text{“}f\text{”} \textbf{ then } G \textbf{ else } \phi(v) \textbf{ end } \rangle$$

To prove the theorem, we need to prove that ψ is a counterexample to "$S \Rightarrow a_0[t] = a_n[t]$".

We continue by proving the following lemmita: for any expression e almost independent of aa,

$$\psi(e) = \phi(e) \tag{51}$$

We prove this lemmita by induction on the structure of e. The only difference between ϕ and ψ is how they interpret the function f and the variables aa. Almost-independence tells us that every occurrence of an aa variable appears as the first argument to some application of f. Hence, to prove the lemmita, it suffices to consider applications of f; all other cases are trivial. Furthermore, almost-independence tells us that each application of f has one of the aa variables as its first argument, so we only need to consider expressions of the form $f(a_j, e)$ for every $j\colon 0 \le j \le n$. We consider two cases. First, for $j\colon 0 \le j \le h$ and any e almost independent of aa,

$$\begin{array}{ll}
 & \psi(f(a_j, e)) \\
= & \qquad \{ \ \psi \ \} \\
 & G(B_j, \psi(e)) \\
= & \qquad \{ \ (50) \text{ with } Y := \psi(e) \ \} \\
 & F(A_j, \psi(e)) \\
= & \qquad \{ \ \text{induction hypothesis} \ \} \\
 & F(A_j, \phi(e)) \\
= & \qquad \{ \ \phi \ \} \\
 & \phi(f(a_j, e))
\end{array}$$

Second, for $j\colon h < j \le n$ and any e almost independent of aa,

$$\begin{array}{ll}
 & \psi(f(a_j, e)) \\
= & \qquad \{ \ \psi \text{ and } A_j = \phi(a_j) \ \} \\
 & G(A_j, \psi(e)) \\
= & \qquad \{ \ (47) \text{ and } G \ \} \\
 & F(A_j, \psi(e)) \\
= & \qquad \{ \ \text{induction hypothesis} \ \} \\
 & F(A_j, \phi(e)) \\
= & \qquad \{ \ \phi \ \} \\
 & \phi(f(a_j, e))
\end{array}$$

This proves lemmita (51).

From this lemmita, we have

$$\begin{aligned}&\psi(P) = \phi(P) \wedge \\ &\langle \bigwedge j \;\rule{2pt}{8pt}\; 0 \le j < n \;::\; \langle \forall s \;::\; \psi(K_j(s)) = \phi(K_j(s)) \,\rangle\rangle\end{aligned} \tag{52}$$

From (46), we then also have

$$\psi(P) \tag{53}$$

We continue by proving another lemmita: for any $j\colon 0 \le j < n$ and any dummy variable s,

$$j \ne h \;\vee\; s \ne T \;\Rightarrow\; \psi(a_j[s] = a_{j+1}[s]) = \phi(a_j[s] = a_{j+1}[s]) \tag{54}$$

Technically, ϕ and ψ do not have any dummy variables in their domains. Thus, the proper way to write the consequence of this lemmita would be

$$\langle \forall Y \;::\; \psi(\text{“}s\text{”} \mapsto Y)(a_j[s] = a_{j+1}[s]) = \phi(\text{“}s\text{”} \mapsto Y)(a_j[s] = a_{j+1}[s]) \,\rangle$$

where $\psi(\text{“}s\text{”} \mapsto Y)$ denotes function ψ extended to map “s” to Y. However, for brevity, and we claim without actual precision, we take ϕ and ψ to map dummy variables to themselves, implicitly denoting any value. This allows us to state the lemmita the way we did and lets us avoid unnecessary clutter in our proof. Actually, this comment also applies to lemmita (51), where we shamelessly swept this issue under the rug.

We prove lemmita (54) by considering four cases:

$$\begin{aligned}&h < j \\ &j < h \;\wedge\; s \ne T \\ &j < h \;\wedge\; s = T \\ &j = h \;\wedge\; s \ne T\end{aligned}$$

First, if $h < j$, then

$$\begin{array}{ll} & \psi(a_j[s] = a_{j+1}[s]) \\ = & \qquad \{ \quad \psi \quad \} \\ & \phi(a_j)[s] = \phi(a_{j+1})[s] \\ = & \qquad \{ \quad \phi \quad \} \\ & \phi(a_j[s] = a_{j+1}[s]) \end{array}$$

Second, if $j < h \;\wedge\; s \ne T$, then

$$\psi(a_j[s] = a_{j+1}[s])$$
$$= \quad \{ \quad \psi \quad \}$$
$$B_j[s] = B_{j+1}[s]$$
$$= \quad \{ \quad B_j\ ,\ B_{j+1}\ , \text{ and (43): select and store, since } s \neq T \quad \}$$
$$A_j[s] = A_{j+1}[s]$$
$$= \quad \{ \quad \phi \quad \}$$
$$\phi(a_j[s] = a_{j+1}[s])$$

Third, if $j < h \ \wedge \ s = T$, then

$$\psi(a_j[s] = a_{j+1}[s])$$
$$= \quad \{ \quad \psi \quad \}$$
$$B_j[s] = B_{j+1}[s]$$
$$= \quad \{ \quad B_j\ ,\ B_{j+1}\ , \text{ and (43): select and store, since } s = T \quad \}$$
$$V_j = V_{j+1}$$
$$= \quad \{ \quad (48) \quad \}$$
$$A_j[T] = A_{j+1}[T]$$
$$= \quad \{ \quad s = T \text{ and } \phi \quad \}$$
$$\phi(a_j[s] = a_{j+1}[s])$$

Fourth, if $j = h \ \wedge \ s \neq T$, then

$$\psi(a_h[s] = a_{h+1}[s])$$
$$= \quad \{ \quad \psi \quad \}$$
$$B_h[s] = \phi(a_{h+1})[s]$$
$$= \quad \{ \quad B_h \text{ and (43): select and store, since } s \neq T \quad \}$$
$$A_h[s] = \phi(a_{h+1})[s]$$
$$= \quad \{ \quad \phi \quad \}$$
$$\phi(A_h[s] = a_{h+1}[s])$$

This proves lemmita (54).

We can now prove that ψ satisfies

$$\langle \bigwedge j \mid 0 \leq j < n \ :: \ \psi(\langle \forall s \ :: \ a_j[s] = a_{j+1}[s] \ \vee \ K_j(s) \rangle) \rangle \tag{55}$$

We consider two cases. First, if $j \neq h$, then

$$\psi(\langle \forall s \ :: \ a_j[s] = a_{j+1}[s] \ \vee \ K_j(s) \rangle)$$
$$= \quad \{ \quad \psi \quad \}$$
$$\langle \forall s \ :: \ \psi(a_j[s] = a_{j+1}[s]) \ \vee \ \psi(K_j(s)) \rangle$$
$$= \quad \{ \quad \text{(52), and lemmita (54) since } j \neq h \quad \}$$

$$
\begin{array}{ll}
 & \langle \forall s :: \phi(a_j[s] = a_{j+1}[s]) \ \vee \ \phi(K_j(s)) \rangle \\
= & \quad \{ \ \phi \ \} \\
 & \phi(\langle \forall s :: a_j[s] = a_{j+1}[s] \ \vee \ K_j(s) \rangle)
\end{array}
$$

Second, if $j = h$, then

$$
\begin{array}{ll}
 & \psi(\langle \forall s :: a_h[s] = a_{h+1}[s] \ \vee \ K_h(s) \rangle) \\
= & \quad \{ \ \psi \ \} \\
 & \langle \forall s :: \psi(a_h[s] = a_{h+1}[s]) \ \vee \ \psi(K_h(s)) \rangle \\
= & \quad \{ \ \text{split range} \ \} \\
 & \langle \forall s :: s \neq T \ \Rightarrow \ \psi(a_h[s] = a_{h+1}[s]) \ \vee \ \psi(K_h(s)) \rangle \ \wedge \\
 & \langle \forall s :: s = T \ \Rightarrow \ \psi(a_h[s] = a_{h+1}[s]) \ \vee \ \psi(K_h(s)) \rangle \\
= & \quad \{ \ \text{(52), and lemmita (54) since } s \neq T \ \} \\
 & \langle \forall s :: s \neq T \ \Rightarrow \ \phi(a_h[s] = a_{h+1}[s]) \ \vee \ \phi(K_h(s)) \rangle \ \wedge \\
 & \langle \forall s :: s = T \ \Rightarrow \ \psi(a_h[s] = a_{h+1}[s]) \ \vee \ \phi(K_h(s)) \rangle \\
= & \quad \{ \ T = \phi(t)\text{, and (46): } \phi(K_h(t))\text{, hence } s = T \ \Rightarrow \ \phi(K_h(s)) \ \} \\
 & \langle \forall s :: s \neq T \ \Rightarrow \ \phi(a_h[s] = a_{h+1}[s]) \ \vee \ \phi(K_h(s)) \rangle \ \wedge \\
 & \langle \forall s :: s = T \ \Rightarrow \ \phi(K_h(s)) \rangle \\
= & \quad \{ \ T = \phi(t)\text{, and (46): } \phi(K_h(t))\text{, hence } s = T \ \Rightarrow \ \phi(K_h(s)) \ \} \\
 & \langle \forall s :: s \neq T \ \Rightarrow \ \phi(a_h[s] = a_{h+1}[s]) \ \vee \ \phi(K_h(s)) \rangle \ \wedge \\
 & \langle \forall s :: s = T \ \Rightarrow \ \phi(a_h[s] = a_{h+1}[s]) \ \vee \ \phi(K_h(s)) \rangle \\
= & \quad \{ \ \text{combine range} \ \} \\
 & \langle \forall s :: \phi(a_h[s] = a_{h+1}[s]) \ \vee \ \phi(K_h(s)) \rangle \\
= & \quad \{ \ \phi \ \} \\
 & \phi(\langle \forall s :: a_h[s] = a_{h+1}[s] \ \vee \ K_h(s) \rangle)
\end{array}
$$

That proves (55).

We have two things left to prove before we can conclude that ψ is a counterexample to "$S \ \Rightarrow \ a_0[t] = a_n[t]$": checking that Q holds under ψ and checking that the consequent does not hold under ψ. We continue by working on the latter, and thus prove:

$$
\begin{array}{ll}
 & \psi(a_0[t] \neq a_n[t]) \\
= & \quad \{ \ \psi\text{, since } 0 \leq h < n \ \} \\
 & B_0[T] \neq A_n[T] \\
= & \quad \{ \ B_0 = \mathit{store}(A_0, T, V_0)\text{, and (43): select and store} \ \} \\
 & V_0 \neq A_n[T] \\
= & \quad \{ \ \text{(47), since } 0 \leq h < n \ \} \\
 & \mathit{true}
\end{array}
\tag{56}
$$

Finally, using the fact that applying ϕ in the conjunct $\phi(Q)$ of (46) yields

$$\langle \forall x, y, z, s :: \phi(M(y, z, s)) \Rightarrow F(x, y)[s] = z \rangle \wedge \tag{57}$$

$$\begin{aligned}\langle \forall w, x, y, z, s :: \; & w[s] = x[s] \wedge \phi(N(y, z, s)) \Rightarrow \\ & F(w, y)[s] = F(x, z)[s] \rangle\end{aligned} \tag{58}$$

we show:

$$\psi(Q) \tag{59}$$

For the first conjunct of Q, we calculate,

$$\begin{array}{ll} & \psi(\langle \forall x, y, z, s :: M(y, z, s) \Rightarrow f(x, y)[s] = z \rangle) \\ = & \quad \{ \; \psi \; \} \\ & \langle \forall x, y, z, s :: \psi(M(y, z, s)) \Rightarrow G(x, y)[s] = z \rangle \\ = & \quad \{ \; M(y, z, s) \text{ is independent of } f \text{ and } aa \; \} \\ & \langle \forall x, y, z, s :: \phi(M(y, z, s)) \Rightarrow G(x, y)[s] = z \rangle \end{array}$$

We consider two cases. First, if $x[T]$ is not among the V's (that is, if there is no $j\!: 0 \le j \le h$ such that $x[T] = V_j$), we have

$$\begin{array}{ll} & G(x, y)[s] = z \\ = & \quad \{ \; G, \text{ since } x[T] \text{ is not among the } V\text{'s} \; \} \\ & F(x, y)[s] = z \\ \Leftarrow & \quad \{ \; (57) \; \} \\ & \phi(M(y, z, s)) \end{array}$$

Second, for any $j\!: 0 \le j \le h$ such that $x[T] = V_j$, we have

$$\begin{array}{ll} & G(x, y)[s] = z \\ = & \quad \{ \; G, \text{ since } x[T] = V_j \; \} \\ & F(store(x, T, A_j[T]), y)[s] = z \\ \Leftarrow & \quad \{ \; (57) \text{ with } x := store(x, T, A_j[T]) \; \} \\ & \phi(M(y, z, s)) \end{array}$$

Now for the second conjunct of Q, we calculate,

$$\begin{array}{ll} & \psi(\langle \forall w, x, y, z, s :: w[s] = x[s] \wedge N(y, z, s) \Rightarrow f(w, y)[s] = f(x, z)[s] \rangle) \\ = & \quad \{ \; \psi \; \} \\ & \langle \forall w, x, y, z, s :: w[s] = x[s] \wedge \psi(N(y, z, s)) \Rightarrow G(w, y)[s] = G(x, z)[s] \rangle \\ = & \quad \{ \; N(y, z, s) \text{ is independent of } f \text{ and } aa \; \} \\ & \langle \forall w, x, y, z, s :: w[s] = x[s] \wedge \phi(N(y, z, s)) \Rightarrow G(w, y)[s] = G(x, z)[s] \rangle \end{array}$$

We consider three cases. First, if neither $w[T]$ nor $x[T]$ is among the V's, then

$$
\begin{array}{ll}
 & G(w,y)[s] = G(x,z)[s] \\
= & \quad \{ \ \ G\text{, since neither } w[T] \text{ nor } x[T] \text{ is among the } V\text{'s} \ \ \} \\
 & F(w,y)[s] = F(x,z)[s] \\
\Leftarrow & \quad \{ \ \ (58) \ \ \} \\
 & w[s] = x[s] \ \wedge \ \phi(N(y,z,s))
\end{array}
$$

Second, if exactly one of $w[T]$ and $x[T]$ is among the V's, say $w[T] = V_j$, then we note that $w[T] \neq x[T]$ and calculate,

$$
\begin{array}{ll}
 & G(w,y)[s] = G(x,z)[s] \\
= & \quad \{ \ \ G\text{, given assumptions about } w[T] \text{ and } x[T] \ \ \} \\
 & F(\mathit{store}(w,T,A_j[T]),y)[s] = F(x,z)[s] \\
\Leftarrow & \quad \{ \ \ (58) \text{ with } w := \mathit{store}(w,T,A_j[T]) \ \ \} \\
 & \mathit{store}(w,T,A_j[T])[s] = x[s] \ \wedge \ \phi(N(y,z,s)) \\
= & \quad \{ \ \ (43)\text{: select and store} \ \ \} \\
 & (s = T \ \Rightarrow \ A_j[T] = x[T]) \ \wedge \ (s \neq T \ \Rightarrow \ w[s] = x[s]) \ \wedge \ \phi(N(y,z,s)) \\
\Leftarrow & \quad \{ \ \ w[T] \neq x[T]\text{, and thus vacuously } w[T] = x[T] \ \Rightarrow \ A_j[T] = x[T] \ \ \} \\
 & (s = T \ \Rightarrow \ w[T] = x[T]) \ \wedge \ (s \neq T \ \Rightarrow \ w[s] = x[s]) \ \wedge \ \phi(N(y,z,s)) \\
= & \quad \{ \ \ \text{combine cases} \ \ \} \\
 & w[s] = x[s] \ \wedge \ \phi(N(y,z,s))
\end{array}
$$

Third, if both $w[T]$ and $x[T]$ are among the V's, say $w[T] = V_i$ and $x[T] = V_j$, then

$$
\begin{array}{ll}
 & G(w,y)[s] = G(x,z)[s] \\
= & \quad \{ \ \ G\text{, since } w[T] = V_i \text{ and } x[T] = V_j \ \ \} \\
 & F(\mathit{store}(w,T,A_i[T]),y)[s] = F(\mathit{store}(x,T,A_j[T]),z)[s] \\
\Leftarrow & \quad \{ \ \ (58) \text{ with } w,x := \mathit{store}(w,T,A_i[T]), \mathit{store}(x,T,A_j[T]) \ \ \} \\
 & \mathit{store}(w,T,A_i[T])[s] = \mathit{store}(x,T,A_j[T])[s] \ \wedge \ \phi(N(y,z,s)) \\
= & \quad \{ \ \ (43)\text{: select and store} \ \ \} \\
 & (s = T \ \Rightarrow \ A_i[T] = A_j[T]) \ \wedge \ (s \neq T \ \Rightarrow \ w[s] = x[s]) \ \wedge \ \phi(N(y,z,s)) \\
= & \quad \{ \ \ (48)\text{, and } w[T] = V_i \text{ and } x[T] = V_j \ \ \} \\
 & (s = T \ \Rightarrow \ w[T] = x[T]) \ \wedge \ (s \neq T \ \Rightarrow \ w[s] = x[s]) \ \wedge \ \phi(N(y,z,s)) \\
= & \quad \{ \ \ \text{combine cases} \ \ \} \\
 & w[s] = x[s] \ \wedge \ \phi(N(y,z,s))
\end{array}
$$

With that, we have proved (59).

Formulas (59), (53), (55), and (56) show that ψ is a counterexample to "$S \Rightarrow a_0[t] = a_n[t]$", which completes the proof of the Chain of Equalities Lemma. ∎

A23 Chain Rewriting Lemma

Recall the simple argument that we are formalizing: $VC_D(m, C)$ is valid, hence C respects the modification constraint for $res.a$, for each a that depends on ε. Hence each procedure call in C respects these constraints, from which it can be concluded that C itself respects the modification constraint for ε. The Chain of Equalities Lemma is our essential tool for formalizing this argument, but there is still an argument required to show that $VC_D(m, C)$ can be written in a form to which the Chain of Equalities Lemma applies. This argument is a structural induction on the command C. We show by induction that the verification condition is equivalent to an implication in which the antecedent is a conjunction to which the Chain of Equalities Lemma applies. The exact inductive assertion is a lemma that we call the Chain Rewriting Lemma.

The setting of this lemma is as follows. Let E be any scope, ε a concrete field in E, and zz the list of concrete fields and residue variables in E. Let xx denote the list consisting of ε and the residue variable $res.a$ for each a such that $a\ \mathbf{on}_E\ \varepsilon$.

The verification conditions that are the subject of our inductive proof can contain many adorned variables, so in this lemma, we will adorn variables with numerical subscripts instead of with accents. We will even use negative subscripts, so that, for example, x_0, x_{-3}, and x_{17} are three adornments of x. The relevant adornments of each variable are always a linear sequence, whose subscripts range from $-2k$ to $3k+1$ for some natural number k. The first adornment, x_{-2k}, represents the input value $\grave{x}$ and the last adornment, x_{3k+1}, represents the current value (default adornment) x. Therefore, for any k, we define a k*-replacement* to be a substitution θ such that for any x in xx,

$$\grave{x}\,\theta = x_{-2k} \qquad \text{and} \qquad x\,\theta = x_{3k+1}$$

A k-replacement may also arbitrarily rename local-variable scalars.

We next define a predicate *chain*. For any sequence α of modifies lists in E, any x in xx, and any integer k, $chain_x(\alpha, k)$ asserts a relation between the adornments of x from x_{-2k} to x_{3k+1}: pairs of adjacent x's are either equal or are related as allowed by one of the modifies lists in α. In particular, one fifth of the pairs are constrained by a modifies list in α, and the other pairs are constrained to be equal. (The reason for this strange definition is that the *wlp* equation for method call avoids variable capture by introducing five dummies, one of which is constrained by a modifies list.) Formally, we define $chain_x(\alpha, k)$ as:

$$\langle \bigwedge j \mid -2k \le j < 3k+1 :: \langle \forall s :: x_j[s] = x_{j+1}[s] \ \vee\ K_j(x, \alpha, s) \rangle\rangle$$

where

$$
\begin{aligned}
K_j(x, \alpha, s) \;=\; & \textbf{if } 0 \le j \;\wedge\; i = (j-2)/3 \text{ for some integer } i \textbf{ then} \\
& \qquad F_E(\mathit{modpoint}_E(x, \alpha_i, s))(\grave{z}z := zz_j) \\
& \textbf{else} \\
& \qquad \mathit{false} \\
& \textbf{end}
\end{aligned}
$$

The definition implies equality between four out of five adjacent adornments, because due to select property (44),

$$x_j = x_{j+1} \;\equiv\; \langle \forall s \;::\; x_j[s] = x_{j+1}[s] \;\vee\; \mathit{false} \,\rangle$$

Note that

$$\mathit{chain}_x(\alpha, 0) \;\equiv\; x_0 = x_1$$

and that

$$
\begin{aligned}
& \mathit{chain}_x(\alpha, k+1) \;\equiv \\
& \quad \mathit{chain}_x(\alpha, k) \;\wedge \\
& \quad x_{-2k-2} = x_{-2k-1} \;\wedge\; x_{-2k-1} = x_{-2k} \;\wedge\; x_{3k+1} = x_{3k+2} \;\wedge \\
& \quad \langle \forall s \;::\; x_{3k+2}[s] = x_{3k+3}[s] \;\vee\; F_E(\mathit{modpoint}_E(x, \alpha_k, s))(\grave{z}z := zz_{3k+2}) \,\rangle \;\wedge \\
& \quad x_{3k+3} = x_{3k+4}
\end{aligned}
\tag{60}
$$

From the definitions of $\mathit{modpoint}_E$ and F_E, we observe, for any x, w, and s,

$$F_E(\mathit{modpoint}_E(x, w, s)) \text{ is independent of } \acute{z}z \tag{61}$$

Finally, we define

$$\mathit{chain}(\alpha, k) \;=\; \langle \textstyle\bigwedge x \;\|\; x \in xx \;::\; \mathit{chain}_x(\alpha, k) \,\rangle$$

In this setting, we now state the lemma:

Chain Rewriting Lemma. For any □-free command C in E that does not assign directly to ε, any sequence α of modifies lists in E, and any k-replacement θ, there exist

- a predicate P such that, for every individual residue variable $res.a$ in xx for an abstract field a, P is almost independent (page 113) of $res.a$ with respect to function $\mathcal{F}.a$,
- a sequence β of modifies lists in E,
- an integer ℓ, and
- an ℓ-replacement σ,

such that for any predicate Q and for any predicate R whose free variables are not post-adorned,

$$\begin{array}{l}[Q \;\wedge\; chain(\alpha, k) \;\Rightarrow\; ulp_E(C, R)\,\theta] \;\equiv \\ [Q \;\wedge\; P \;\wedge\; chain(\beta, \ell) \;\Rightarrow\; R\,\sigma]\end{array} \tag{62}$$

(End of Lemma.)

Proof. We prove the lemma by induction over the structure of C. We consider a number of cases, each of which is proved by a calculation that ends with suitable β, ℓ, σ, and P.

CASE $s := e$:

$$\begin{array}{ll}
& [Q \;\wedge\; chain(\alpha, k) \;\Rightarrow\; ulp_E(s := e, R)\,\theta] \\
= & \quad \{ \;\; ulp_E \;\; \} \\
& [Q \;\wedge\; chain(\alpha, k) \;\Rightarrow\; \langle \forall s' \;::\; s' = F_E(e) \;\Rightarrow\; \langle \forall s \;::\; s = s' \;\Rightarrow\; R \,\rangle\rangle\,\theta] \\
= & \quad \{ \;\; \text{distribute quantification over } s' \text{ out (since } s' \text{ is fresh), then absorb} \\
& \quad\quad \text{it into the everywhere brackets; predicate calculus} \;\; \} \\
& [Q \;\wedge\; chain(\alpha, k) \;\wedge\; s' = F_E(e)\,\theta \;\Rightarrow\; \langle \forall s \;::\; s = s' \;\Rightarrow\; R \,\rangle\,\theta] \\
= & \quad \{ \;\; \text{rewrite quantification as a substitution} \;\; \} \\
& [Q \;\wedge\; chain(\alpha, k) \;\wedge\; s' = F_E(e)\,\theta \;\Rightarrow\; R(s := s')\,\theta]
\end{array}$$

which is the right side of (62) with $\beta := \alpha$, $\ell := k$, $\sigma := (s := s')\,\theta$, and $P := s' = F_E(e)\,\theta$.

CASE $c[e0] := e1$:

$$
\begin{array}{ll}
 & [Q \ \wedge \ chain(\alpha, k) \ \Rightarrow \ ulp_E(c[e0] := e1, R)\,\theta] \\
= & \qquad \{ \quad ulp_E \quad \} \\
 & [Q \ \wedge \ chain(\alpha, k) \ \Rightarrow \\
 & \langle \forall c' \ :: \ c' = store(c, F_E(e0), F_E(e1)) \ \Rightarrow \ \langle \forall c \ :: \ c = c' \ \Rightarrow \ R \,\rangle\rangle\,\theta] \\
= & \qquad \{ \quad \text{predicate calculus, since } c' \text{ is fresh} \quad \} \\
 & [Q \ \wedge \ chain(\alpha, k) \ \wedge \ c' = store(c, F_E(e0), F_E(e1))\,\theta \ \Rightarrow \\
 & \langle \forall c \ :: \ c = c' \ \Rightarrow \ R \,\rangle\,\theta] \\
= & \qquad \{ \quad \text{rewrite quantification as a substitution} \quad \} \\
 & [Q \ \wedge \ chain(\alpha, k) \ \wedge \ c' = store(c, F_E(e0), F_E(e1))\,\theta \ \Rightarrow \ R(c := c')\,\theta]
\end{array}
$$

CASE **assert** e :

$$
\begin{array}{ll}
 & [Q \ \wedge \ chain(\alpha, k) \ \Rightarrow \ ulp_E(\textbf{assert}\ e, R)\,\theta] \\
= & \qquad \{ \quad ulp_E \quad \} \\
 & [Q \ \wedge \ chain(\alpha, k) \ \Rightarrow \ R\,\theta]
\end{array}
$$

CASE **assume** e :

$$
\begin{array}{ll}
 & [Q \ \wedge \ chain(\alpha, k) \ \Rightarrow \ ulp_E(\textbf{assume}\ e, R)\,\theta] \\
= & \qquad \{ \quad ulp_E \quad \} \\
 & [Q \ \wedge \ chain(\alpha, k) \ \Rightarrow \ (F_E(e) \ \Rightarrow \ R)\,\theta] \\
= & \qquad \{ \quad \text{predicate calculus} \quad \} \\
 & [Q \ \wedge \ chain(\alpha, k) \ \wedge \ F_E(e)\,\theta \ \Rightarrow \ R\,\theta]
\end{array}
$$

CASE $C0\ ;\ C1$:

$$
\begin{array}{ll}
 & [Q \ \wedge \ chain(\alpha, k) \ \Rightarrow \ ulp_E(C0\ ;\ C1, R)\,\theta] \\
= & \qquad \{ \quad ulp_E \quad \} \\
 & [Q \ \wedge \ chain(\alpha, k) \ \Rightarrow \ ulp_E(C0, ulp_E(C1, R))\,\theta] \\
= & \qquad \{ \quad \text{induction hypothesis, with } C, R := C0, ulp_E(C1, R) \quad \} \\
 & [Q \ \wedge \ P \ \wedge \ chain(\beta, \ell) \ \Rightarrow \ ulp_E(C1, R)\,\sigma] \\
= & \qquad \{ \quad \text{induction hypothesis, with } Q, k, C, \theta := Q \wedge P, \ell, C1, \sigma \quad \} \\
 & [Q \ \wedge \ P \ \wedge \ P' \ \wedge \ chain(\gamma, m) \ \Rightarrow \ R\,\tau]
\end{array}
$$

CASE **var** s **in** C **end** : We assume s is a fresh name.

$$
\begin{array}{ll}
 & [Q \ \wedge \ chain(\alpha, k) \ \Rightarrow \ ulp_E(\textbf{var}\ s\ \textbf{in}\ C\ \textbf{end}, R)\,\theta] \\
= & \qquad \{ \quad ulp_E \quad \} \\
 & [Q \ \wedge \ chain(\alpha, k) \ \Rightarrow \ \langle \forall s \ :: \ ulp_E(C, R) \,\rangle\,\theta] \\
= & \qquad \{ \quad \text{predicate calculus, since } s \text{ is fresh} \quad \}
\end{array}
$$

$$[Q \ \wedge\ chain(\alpha, k) \ \Rightarrow\ ulp_E(C, R)\,\theta]$$

$$= \qquad \{ \ \text{induction hypothesis} \ \}$$

$$[Q \ \wedge\ P \ \wedge\ chain(\beta, \ell) \ \Rightarrow\ R\,\sigma]$$

CASE **call** $m(e)$: Assume the specification of m , after renaming its parameter to a fresh dummy t , is

method $m(t\colon T)$ **requires** p **modifies** w **ensures** q

We calculate,

$$
\begin{array}{ll}
 & [Q \ \wedge\ chain(\alpha, k) \ \Rightarrow\ ulp_E(\mathbf{call}\ m(e), R)\,\theta] \\
= & \qquad \{ \ \ ulp_E \ \ \} \\
 & [Q \ \wedge\ chain(\alpha, k) \ \Rightarrow\ \langle \forall\, t \ :: \ t = F_E(e) \ \Rightarrow \\
 & \quad \langle \forall\, \ddot{z}z \ :: \ \ddot{z}z = \grave{z}z \ \Rightarrow\ \langle \forall\, \grave{z}z \ :: \ \grave{z}z = zz \ \Rightarrow \\
 & \qquad \langle \forall\, \acute{z}z \ :: \ F_E(q) \ \wedge\ mc_E(w) \ \Rightarrow \\
 & \qquad\quad \langle \forall\, \grave{z}z \ :: \ \grave{z}z = \ddot{z}z \ \Rightarrow\ \langle \forall\, zz \ :: \ zz = \acute{z}z \ \Rightarrow\ R\, \rangle\rangle\rangle\rangle\rangle\rangle\,\theta] \\
= & \qquad \{ \ \text{predicate calculus, and rename } \ddot{z}z, \grave{z}z, \acute{z}z, \grave{z}z, zz \text{ to} \\
 & \qquad\quad zz_{-2k-1}, zz_{3k+2}, zz_{3k+3}, zz_{-2k-2}, zz_{3k+4} \ \ \} \\
 & [Q \ \wedge\ chain(\alpha, k) \ \wedge\ t = F_E(e)\,\theta \ \Rightarrow \\
 & \quad \langle \forall\, zz_{-2k-1} \ :: \ zz_{-2k-1} = \grave{z}z \ \Rightarrow\ \langle \forall\, zz_{3k+2} \ :: \ zz_{3k+2} = zz \ \Rightarrow \\
 & \qquad \langle \forall\, zz_{3k+3} \ :: \ (F_E(q) \ \wedge\ mc_E(w))(\grave{z}z, \acute{z}z := zz_{3k+2}, zz_{3k+3}) \ \Rightarrow \\
 & \qquad\quad \langle \forall\, zz_{-2k-2} \ :: \ zz_{-2k-2} = zz_{-2k-1} \ \Rightarrow\ \langle \forall\, zz_{3k+4} \ :: \ zz_{3k+4} = zz_{3k+3} \ \Rightarrow \\
 & \qquad\qquad R(\grave{z}z, zz := zz_{-2k-2}, zz_{3k+4})\ \rangle\rangle\rangle\rangle\rangle\,\theta] \\
= & \qquad \{ \ \text{predicate calculus} \ \} \\
 & [Q \ \wedge\ chain(\alpha, k) \ \wedge\ t = F_E(e)\,\theta \ \wedge\ zz_{-2k-1} = \grave{z}z\,\theta \ \wedge\ zz_{3k+2} = zz\,\theta \ \wedge \\
 & (F_E(q) \ \wedge\ mc_E(w))(\grave{z}z, \acute{z}z := zz_{3k+2}, zz_{3k+3}) \ \wedge\ zz_{-2k-2} = zz_{-2k-1} \ \wedge \\
 & zz_{3k+4} = zz_{3k+3} \\
 & \Rightarrow\ R(\grave{z}z, zz := zz_{-2k-2}, zz_{3k+4})\,\theta] \\
= & \qquad \{ \ \text{let } yy \text{ denote the list of variables in } zz \text{ but not in } xx, \\
 & \qquad\quad \text{let } mc_E^{xx}(w) = \langle \bigwedge x \ \mid\ x \in xx \ :: \ modcon_E(w, x, x) \rangle \text{, and} \\
 & \qquad\quad \text{let } mc_E^{yy}(w) = \langle \bigwedge y \ \mid\ y \in yy \ :: \ modcon_E(w, y, y) \rangle \ \ \} \\
 & [Q \ \wedge\ chain(\alpha, k) \ \wedge\ t = F_E(e)\,\theta \ \wedge \\
 & xx_{-2k-1} = \grave{x}x\,\theta \ \wedge\ yy_{-2k-1} = \grave{y}y\,\theta \ \wedge\ xx_{3k+2} = xx\,\theta \ \wedge\ yy_{3k+2} = yy\,\theta \ \wedge \\
 & (F_E(q) \wedge mc_E^{yy}(w))(\grave{z}z, \acute{z}z := zz_{3k+2}, zz_{3k+3}) \ \wedge \\
 & mc_E^{xx}(w)(\grave{z}z, \acute{z}z := zz_{3k+2}, zz_{3k+3}) \ \wedge \\
 & xx_{-2k-2} = xx_{-2k-1} \ \wedge\ yy_{-2k-2} = yy_{-2k-1} \ \wedge \\
 & xx_{3k+4} = xx_{3k+3} \ \wedge\ yy_{3k+4} = yy_{3k+3} \\
 & \Rightarrow\ R(\grave{z}z, zz := zz_{-2k-2}, zz_{3k+4})\,\theta] \\
= & \qquad \{ \ \ \theta \text{ is a } k\text{-replacement, hence } \grave{x}x\,\theta = xx_{-2k} \text{ and } xx\,\theta = xx_{3k+1} \ \ \}
\end{array}
$$

$$
\begin{array}{ll}
& [Q \ \wedge \ chain(\alpha, k) \ \wedge \ t = F_E(e)\,\theta \ \wedge \\
& xx_{-2k-1} = xx_{-2k} \ \wedge \ yy_{-2k-1} = \grave{yy}\,\theta \ \wedge \ xx_{3k+2} = xx_{3k+1} \ \wedge \ yy_{3k+2} = yy\,\theta \ \wedge \\
& (F_E(q) \wedge mc_E^{yy}(w))(\grave{zz}, \acute{zz} := zz_{3k+2}, zz_{3k+3}) \ \wedge \\
& mc_E^{xx}(w)(\grave{zz}, \acute{zz} := zz_{3k+2}, zz_{3k+3}) \ \wedge \\
& xx_{-2k-2} = xx_{-2k-1} \ \wedge \ yy_{-2k-2} = yy_{-2k-1} \ \wedge \\
& xx_{3k+4} = xx_{3k+3} \ \wedge \ yy_{3k+4} = yy_{3k+3} \\
& \Rightarrow \ R(\grave{zz}, zz := zz_{-2k-2}, zz_{3k+4})\,\theta] \\
= & \qquad \{ \ \text{let } \beta \text{ be the sequence } \alpha \text{ but with } \beta_k = w \text{, and apply (60) and} \\
& \qquad\quad \text{(61)} \ \} \\
& [Q \ \wedge \ chain(\beta, k+1) \ \wedge \ t = F_E(e)\,\theta \ \wedge \\
& yy_{-2k-1} = \grave{yy}\,\theta \ \wedge \ yy_{3k+2} = yy\,\theta \ \wedge \\
& (F_E(q) \ \wedge \ mc_E^{yy}(w))(\grave{zz}, \acute{zz} := zz_{3k+2}, zz_{3k+3}) \ \wedge \\
& yy_{-2k-2} = yy_{-2k-1} \ \wedge \ yy_{3k+4} = yy_{3k+3} \\
& \Rightarrow \ R(\grave{zz}, zz := zz_{-2k-2}, zz_{3k+4})\,\theta]
\end{array}
$$

Note that $(\grave{zz}, zz := zz_{-2k-2}, zz_{3k+4})\,\theta$ is a $(k+1)$-replacement, which completes the case for method calls.

We have now exhausted all cases (since C is □-free), and thus we have proved the Chain Rewriting Lemma. ■

We define a meta function mcr representing modification constraints for relevant residue variables. For any scopes D and E such that $D \subseteq E$, any modifies list w in D, and any concrete field ε in E but not in D, we define $mcr_E(w)$ to be the following predicate:

$$
\begin{array}{l}
\langle \bigwedge a \ \| \ a \text{ is an abstract field in } E \ \wedge \ a \ \mathbf{on}_E \ \varepsilon \ :: \\
\qquad\qquad modcon_E(w, res.a, res.a) \ \rangle
\end{array}
\tag{63}
$$

We now wrap up the Chain of Equalities Lemma and the Chain Rewriting Lemma into a lemma at the heart of the proof of Soundness Lemma C. The lemma says that if C respects the modification constraints for the relevant residue variables, then C respects the modification constraint for ε.

Chain of Equalities Corollary. For any scopes D and E such that $D \subseteq E$, any user expression p, modifies list w, and command C in D, and any concrete field ε in E but not in D,

$$
\begin{array}{l}
[BP_E \ \wedge \ Rep_E \ \wedge \ PW_E \ \wedge \ F_E(p) \ \Rightarrow \ \langle \forall\, \grave{zz} \ :: \ \grave{zz} = zz \ \Rightarrow \\
\qquad ulp_E(C, \langle \forall\, \acute{zz} \ :: \ \acute{zz} = zz \ \Rightarrow \ mcr_E(w) \ \rangle) \ \rangle] \\
\ \Rightarrow \\
[BP_E \ \wedge \ Rep_E \ \wedge \ PW_E \ \wedge \ F_E(p) \ \Rightarrow \ \langle \forall\, \grave{zz} \ :: \ \grave{zz} = zz \ \Rightarrow \\
\qquad ulp_E(C, \langle \forall\, \acute{zz} \ :: \ \acute{zz} = zz \ \Rightarrow \ modcon_E(w, \varepsilon, \varepsilon) \ \rangle) \ \rangle]
\end{array}
$$

where zz denotes all concrete fields and residue variables in E.

Proof. We begin by showing that it suffices to consider □-free commands. Let S abbreviate the antecedent $BP_E \wedge Rep_E \wedge PW_E \wedge F_E(p)$, let ss be a sufficiently large set of fresh variables, and let n and $\{j \mid 0 \le j < n :: C_j\}$ be like in lemma (40). Then,

$$
\begin{array}{ll}
 & [S \Rightarrow \langle \forall \grave{zz} :: \grave{zz} = zz \Rightarrow ulp_E(C, \langle \forall \acute{zz} :: \acute{zz} = zz \Rightarrow mcr_E(w) \rangle) \rangle] \\
= & \quad \{ \text{ lemma (40) } \} \\
 & [S \Rightarrow \langle \forall \grave{zz} :: \grave{zz} = zz \Rightarrow \langle \bigwedge j \mid 0 \le j < n :: \langle \forall ss :: \\
 & \qquad\qquad ulp_E(C_j, \langle \forall \acute{zz} :: \acute{zz} = zz \Rightarrow mcr_E(w) \rangle) \rangle\rangle\rangle] \\
= & \quad \{ \text{ predicate calculus } \} \\
 & \langle \bigwedge j \mid 0 \le j < n :: \langle \forall ss :: [S \Rightarrow \langle \forall \grave{zz} :: \grave{zz} = zz \Rightarrow \\
 & \qquad\qquad ulp_E(C_j, \langle \forall \acute{zz} :: \acute{zz} = zz \Rightarrow mcr_E(w) \rangle) \rangle] \rangle\rangle \\
\Rightarrow & \quad \{ \text{ a version of the Chain of Equalities Corollary for □-free} \\
 & \quad\;\;\, \text{commands, to be proved } \} \\
 & \langle \bigwedge j \mid 0 \le j < n :: \langle \forall ss :: [S \Rightarrow \langle \forall \grave{zz} :: \grave{zz} = zz \Rightarrow \\
 & \qquad\qquad ulp_E(C_j, \langle \forall \acute{zz} :: \acute{zz} = zz \Rightarrow modcon_E(w, \varepsilon, \varepsilon) \rangle) \rangle] \rangle\rangle \\
= & \quad \{ \text{ predicate calculus } \} \\
 & [S \Rightarrow \langle \forall \grave{zz} :: \grave{zz} = zz \Rightarrow \langle \bigwedge j \mid 0 \le j < n :: \langle \forall ss :: \\
 & \qquad\qquad ulp_E(C_j, \langle \forall \acute{zz} :: \acute{zz} = zz \Rightarrow modcon_E(w, \varepsilon, \varepsilon) \rangle) \rangle\rangle\rangle] \\
= & \quad \{ \text{ lemma (40) } \} \\
 & [S \Rightarrow \langle \forall \grave{zz} :: \grave{zz} = zz \Rightarrow \\
 & \qquad\qquad ulp_E(C, \langle \forall \acute{zz} :: \acute{zz} = zz \Rightarrow modcon_E(w, \varepsilon, \varepsilon) \rangle) \rangle]
\end{array}
$$

Thus, from now on, we assume C to be □-free.

Since w is in D but ε is not, ε is not mentioned in w, and so with a ranging over the abstract variables in E,

$$[modpoint_E(\varepsilon, w, t) \equiv \langle \bigvee a \mid a \; \mathbf{on}_E \; \varepsilon :: modpoint_E(a, w, t) \rangle]$$

From (23), we then have, with $res.a$ denoting the residue variable of a,

$$[modpoint_E(\varepsilon, w, t) \equiv \langle \bigvee a \mid a \; \mathbf{on}_E \; \varepsilon :: modpoint_E(res.a, w, t) \rangle] \qquad (64)$$

For any abstract variable a, let QR_a be the conjunction of all rep axioms for a in E. Note that QR_a can be written in the form

$$\langle \forall res.a, y, z, s :: M(y, z, s) \Rightarrow \mathcal{F}.a(res.a, y)[s] = z \rangle$$

for some suitable M and suitable list y of variables. For any abstract variable a, let QP_a be the pointwise axiom for $\mathcal{F}.a$ in E. Note that QP_a has the form

$$\langle \forall\, r\grave{e}s.a, r\acute{e}s.a, y, z, s \;::\; r\grave{e}s.a[s] = r\acute{e}s.a[s] \;\wedge\; N(y, z, s) \;\Rightarrow \\ \mathcal{F}.a(r\grave{e}s.a, y)[s] = \mathcal{F}.a(r\acute{e}s.a, z)[s] \,\rangle$$

for some suitable N (essentially $y[s] = z[s]$) and suitable lists of variables y and z. Therefore, $QR_a \;\wedge\; QP_a$ has the shape prescribed for the predicate Q in the Chain of Equalities Lemma (page 113).

Here, let Q be the conjunction

$$\langle \textstyle\bigwedge a \mid a\ \mathbf{on}_E\ \varepsilon \;::\; QR_a \;\wedge\; QP_a \,\rangle$$

Note that Q has no free variables: it is a closed statement about the various abstraction functions. Let R denote the conjuncts of

$$BP_E \;\wedge\; Rep_E \;\wedge\; PW_E \;\wedge\; F_E(p)$$

that are not in Q. So, we have

$$[Q \;\wedge\; R \;\equiv\; BP_E \;\wedge\; Rep_E \;\wedge\; PW_E \;\wedge\; F_E(p)] \tag{65}$$

Also, for any abstract variable a, let Q'_a be the conjunction

$$\langle \textstyle\bigwedge b \mid b\ \mathbf{on}_E\ \varepsilon \;\wedge\; a \neq b \;::\; QR_b \;\wedge\; QP_b \,\rangle$$

So, we have for any a,

$$[Q \;\equiv\; QR_a \;\wedge\; QP_a \;\wedge\; Q'_a] \tag{66}$$

Let xx denote the list consisting of ε and the residue variable $res.a$ for each a such that $a\ \mathbf{on}_E\ \varepsilon$, and let yy denote the list of variables in zz but not in xx.

For any sequence α of modifies lists in E, any x in xx, and any integer k, we define $chain'_x(\alpha, k)$ to be the conjunction

$$\langle \textstyle\bigwedge v \mid v \in xx \;\wedge\; x \neq v \;::\; chain_v(\alpha, k) \,\rangle$$

where $chain$ is defined as in the setting of the Chain Rewriting Lemma (page 122). So, we have for any x in xx and any α and k,

$$[chain(\alpha, k) \;\equiv\; chain_x(\alpha, k) \;\wedge\; chain'_x(\alpha, k)] \tag{67}$$

We calculate,

$$[BP_E \ \wedge\ Rep_E \ \wedge\ PW_E \ \wedge\ F_E(p) \ \Rightarrow\ \langle \forall\, \grave{zz} :: \grave{zz} = zz \ \Rightarrow$$
$$ulp_E(C, \langle \forall\, \acute{zz} :: \acute{zz} = zz \ \Rightarrow\ mcr_E(w) \,\rangle) \,\rangle\,]$$

$=$ { (65) }

$$[Q \ \wedge\ R \ \Rightarrow\ \langle \forall\, \grave{zz} :: \grave{zz} = zz \ \Rightarrow$$
$$ulp_E(C, \langle \forall\, \acute{zz} :: \acute{zz} = zz \ \Rightarrow\ mcr_E(w) \,\rangle) \,\rangle\,]$$

$=$ { predicate calculus, since Q and R have no free occurrences of $\grave{zz}$ }

$$[Q \ \wedge\ R \ \wedge\ \grave{zz} = zz \ \Rightarrow\ ulp_E(C, \langle \forall\, \acute{zz} :: \acute{zz} = zz \ \Rightarrow\ mcr_E(w) \,\rangle)\,]$$

$=$ { rename $\grave{zz}, zz$ to fresh zz_0, zz_1, using the fact that Q has no free variables; and $zz = (xx, yy)$ }

$$[Q \ \wedge\ R(\grave{zz}, zz := zz_0, zz_1) \ \wedge\ xx_0 = xx_1 \ \wedge\ yy_0 = yy_1 \ \Rightarrow$$
$$ulp_E(C, \langle \forall\, \acute{zz} :: \acute{zz} = zz \ \Rightarrow\ mcr_E(w) \,\rangle)(\grave{zz}, zz := zz_0, zz_1)\,]$$

$=$ { let T abbreviate $R(\grave{zz}, zz := zz_0, zz_1) \ \wedge\ yy_0 = yy_1$, and definition of *chain* (page A23) for any list α }

$$[Q \ \wedge\ T \ \wedge\ chain(\alpha, 0) \ \Rightarrow$$
$$ulp_E(C, \langle \forall\, \acute{zz} :: \acute{zz} = zz \ \Rightarrow\ mcr_E(w) \,\rangle)(\grave{zz}, zz := zz_0, zz_1)\,]$$

$=$ { Chain Rewriting Lemma with $Q, k, \theta := Q \wedge T, 0, (\grave{zz}, zz := zz_0, zz_1)$ }

$$[Q \ \wedge\ T \ \wedge\ P \ \wedge\ chain(\beta, \ell) \ \Rightarrow\ \langle \forall\, \acute{zz} :: \acute{zz} = zz \ \Rightarrow\ mcr_E(w) \,\rangle\, \sigma\,]$$

$=$ { rewrite quantification as a substitution }

$$[Q \ \wedge\ T \ \wedge\ P \ \wedge\ chain(\beta, \ell) \ \Rightarrow\ mcr_E(w)(\acute{zz} := zz)\, \sigma\,]$$

$=$ { (63): mcr_E, and predicate calculus, leaving the range of a implicit }

$$\langle \textstyle\bigwedge a :: [Q \ \wedge\ T \ \wedge\ P \ \wedge\ chain(\beta, \ell) \ \Rightarrow$$
$$modcon_E(w, res.a, res.a)(\acute{zz} := zz)\, \sigma\,] \,\rangle$$

$=$ { $modcon_E$, using a fresh t }

$$\langle \textstyle\bigwedge a :: [Q \ \wedge\ T \ \wedge\ P \ \wedge\ chain(\beta, \ell) \ \Rightarrow$$
$$\langle \forall\, t :: \grave{res}.a[t] = \acute{res}.a[t] \ \vee$$
$$modpoint_E(res.a, w, t) \,\rangle(\acute{zz} := zz)\, \sigma\,] \,\rangle$$

$=$ { predicate calculus, since t does not occur free in antecedent }

$$\langle \textstyle\bigwedge a :: [Q \ \wedge\ T \ \wedge\ P \ \wedge\ chain(\beta, \ell) \ \wedge$$
$$\neg modpoint_E(res.a, w, t)(\acute{zz} := zz)\, \sigma \ \Rightarrow$$
$$(\grave{res}.a[t] = \acute{res}.a[t])(\acute{zz} := zz)\, \sigma\,] \,\rangle$$

$=$ { σ is an ℓ-replacement, and let τ abbreviate $(\acute{zz} := zz)\, \sigma$ }

$$\langle \textstyle\bigwedge a :: [Q \ \wedge\ T \ \wedge\ P \ \wedge\ chain(\beta, \ell) \ \wedge\ \neg modpoint_E(res.a, w, t)\, \tau \ \Rightarrow$$
$$res.a_{-2\ell}[t] = res.a_{3\ell+1}[t]\,] \,\rangle$$

$=$ { (66) and (67) }

$$\langle \textstyle\bigwedge a :: [QR_a \wedge QP_a \wedge Q'_a \wedge T \wedge P \wedge chain_{res.a}(\beta, \ell) \wedge$$
$$chain'_{res.a}(\beta, \ell) \wedge \neg modpoint_E(res.a, w, t)\,\tau \Rightarrow$$
$$res.a_{-2\ell}[t] = res.a_{3\ell+1}[t]\ ]\ \rangle$$

$=$ { Chain of Equalities Lemma, since Q'_a, T, P, $chain'_{res.a}(\beta, \ell)$, $\neg modpoint_E(res.a, w, t)\,\tau$, and the K_j's in $chain'_{res.a}(\beta, \ell)$ are almost independent of $res.a$ with respect to $\mathcal{F}.a$ }

$$\langle \textstyle\bigwedge a :: [QR_a \wedge QP_a \wedge Q'_a \wedge T \wedge P \wedge chain_{res.a}(\beta, \ell) \wedge$$
$$chain'_{res.a}(\beta, \ell) \wedge \neg modpoint_E(res.a, w, t)\,\tau \Rightarrow$$
$$\langle \textstyle\bigwedge j \mid -2\ell \le j < 3\ell + 1 :: \neg K_j(res.a, \beta, t) \rangle\ ]\ \rangle$$

$=$ { (66), (67), and predicate calculus, henceforth leaving the range of j implicit }

$$\langle \textstyle\bigwedge a :: [Q \wedge T \wedge P \wedge chain(\beta, \ell) \Rightarrow$$
$$modpoint_E(res.a, w, t)\,\tau \vee \langle \textstyle\bigwedge j :: \neg K_j(res.a, \beta, t) \rangle\ ]\ \rangle$$

$\Rightarrow$ { by (64), $[modpoint_E(res.a, w, t) \Rightarrow modpoint_E(\varepsilon, w, t)]$ }

$$\langle \textstyle\bigwedge a :: [Q \wedge T \wedge P \wedge chain(\beta, \ell) \Rightarrow$$
$$modpoint_E(\varepsilon, w, t)\,\tau \vee \langle \textstyle\bigwedge j :: \neg K_j(res.a, \beta, t) \rangle\ ]\ \rangle$$

$=$ { predicate calculus }

$$[Q \wedge T \wedge P \wedge chain(\beta, \ell) \Rightarrow$$
$$modpoint_E(\varepsilon, w, t)\,\tau \vee \langle \textstyle\bigwedge j :: \langle \textstyle\bigwedge a :: \neg K_j(res.a, \beta, t) \rangle\rangle\ ]$$

$=$ { by (64), the definition of K_j, and DeMorgan's law, we have $[\langle \textstyle\bigwedge a :: \neg K_j(res.a, \beta, t) \rangle \equiv \neg K_j(\varepsilon, \beta, t)]$ }

$$[Q \wedge T \wedge P \wedge chain(\beta, \ell) \Rightarrow$$
$$modpoint_E(\varepsilon, w, t)\,\tau \vee \langle \textstyle\bigwedge j :: \neg K_j(\varepsilon, \beta, t) \rangle\ ]$$

$\Rightarrow$ { instantiating the quantifications in $chain_\varepsilon(\beta, \ell)$ with $s := t$ yields $\langle \textstyle\bigwedge j :: \varepsilon_j[t] = \varepsilon_{j+1}[t] \vee K_j(\varepsilon, \beta, t) \rangle$ }

$$[Q \wedge T \wedge P \wedge chain(\beta, \ell) \Rightarrow$$
$$modpoint_E(\varepsilon, w, t)\,\tau \vee \langle \textstyle\bigwedge j :: \varepsilon_j[t] = \varepsilon_{j+1}[t] \rangle\ ]$$

$\Rightarrow$ { transitivity of $=$ }

$$[Q \wedge T \wedge P \wedge chain(\beta, \ell) \Rightarrow$$
$$modpoint_E(\varepsilon, w, t)\,\tau \vee \varepsilon_{-2\ell}[t] = \varepsilon_{3\ell+1}[t]\ ]$$

$=$ { predicate calculus, since t does not occur free in antecedent; and $\tau = (\acute{z}z := zz)\,\sigma$ where σ is an ℓ-replacement }

$$[Q \wedge T \wedge P \wedge chain(\beta, \ell) \Rightarrow$$
$$\langle \forall t :: modpoint_E(\varepsilon, w, t) \vee \grave{\varepsilon}[t] = \acute{\varepsilon}[t] \rangle\,\tau\ ]$$

$=$ { $modcon_E$; and τ, substitution, and quantification }

$$[Q \ \wedge\ T \ \wedge\ P \ \wedge\ chain(\beta, \ell) \ \Rightarrow$$
$$\quad \langle \forall \acute{zz} \ :: \ \acute{zz} = zz \ \Rightarrow\ modcon_E(w, \varepsilon, \varepsilon) \ \rangle\, \sigma \ ]$$

$=$ { Chain Rewriting Lemma with the same C, α, k, and θ as before, but with $R := \langle \forall \acute{zz} \ :: \ \acute{zz} = zz \ \Rightarrow\ modcon_E(w, \varepsilon, \varepsilon) \ \rangle$ }

$$[Q \ \wedge\ T \ \wedge\ chain(\alpha, 0) \ \Rightarrow$$
$$\quad ulp_E(C, \langle \forall \acute{zz} \ :: \ \acute{zz} = zz \ \Rightarrow\ modcon_E(w, \varepsilon, \varepsilon) \ \rangle)(\grave{zz}, zz := zz_0, zz_1) \ ]$$

$=$ { T, $chain(\alpha, 0)$, yy, xx }

$$[Q \ \wedge\ R(\grave{zz}, zz := zz_0, zz_1) \ \wedge\ zz_0 = zz_1 \ \Rightarrow$$
$$\quad ulp_E(C, \langle \forall \acute{zz} \ :: \ \acute{zz} = zz \ \Rightarrow\ modcon_E(w, \varepsilon, \varepsilon) \ \rangle)(\grave{zz}, zz := zz_0, zz_1) \ ]$$

$\Rightarrow$ { rename zz_0, zz_1 back to $\grave{zz}, zz$, since zz_0, zz_1 were chosen to be fresh }

$$[Q \ \wedge\ R \ \wedge\ \grave{zz} = zz \ \Rightarrow$$
$$\quad ulp_E(C, \langle \forall \acute{zz} \ :: \ \acute{zz} = zz \ \Rightarrow\ modcon_E(w, \varepsilon, \varepsilon) \ \rangle) \ ]$$

$=$ { predicate calculus, since Q has no free variables and R has no free occurrences of $\grave{zz}$ }

$$[Q \ \wedge\ R \ \Rightarrow\ \langle \forall \grave{zz} \ :: \ \grave{zz} = zz \ \Rightarrow$$
$$\quad ulp_E(C, \langle \forall \acute{zz} \ :: \ \acute{zz} = zz \ \Rightarrow\ modcon_E(w, \varepsilon, \varepsilon) \ \rangle) \ \rangle \ ]$$

$=$ { (65) }

$$[BP_E \ \wedge\ Rep_E \ \wedge\ PW_E \ \wedge\ F_E(p) \ \Rightarrow\ \langle \forall \grave{zz} \ :: \ \grave{zz} = zz \ \Rightarrow$$
$$\quad ulp_E(C, \langle \forall \acute{zz} \ :: \ \acute{zz} = zz \ \Rightarrow\ modcon_E(w, \varepsilon, \varepsilon) \ \rangle) \ \rangle \ ]$$

With that, we have proved the Chain of Equalities Corollary. ■

A24 Refunctionalization

The formula $VC_D(m, C)$ has been generated using declarations in D. How do we get from it to $VC_E(m, C)$, that is, how can we even begin to relate $VC_D(m, C)$ to $VC_E(m, C)$? For example, an occurrence of a variable a that depends on ε in E will be functionalized in VC_E to an expression of the form $\mathcal{F}.a(\ldots, \varepsilon)$; while in VC_D the corresponding occurrence of a will be functionalized to $\mathcal{F}.a(\ldots)$. We deal with these kinds of discrepancies by *refunctionalizing* $VC_D(m, C)$ into a formula that looks closer to $VC_E(m, C)$ than $VC_D(m, C)$ does. Refunctionalization is driven by the syntactic form of the given formula. Here is where it will be useful that we left some markers in the VC generated by $VC_D(m, C)$, to give us guidance in refunctionalization.

We introduce refunctionalization as a meta function $X_{D,E}$: if Q is a vanilla expression in D, then $X_{D,E}(Q)$ approximates a vanilla expression in E. To have mercy on our readers, and on ourselves, we drop the subscripts of X and will write simply X for $X_{D,E}$ from now on.

For any D, ε, and E such that $Extend(D, \varepsilon, E)$ (defined in Section A16), we define X in Figure 17. The definition follows the cases in the grammar for vanilla expressions in Section A12, but with more special cases added. Since scalars (line 0) and concrete fields (line 1) are unchanged by functionalization, they are also unchanged by refunctionalization. An abstract variable a (line 2) with one direct dependency, say f, in D and one additional dependency, ε, in E is refunctionalized by recursively refunctionalizing the arguments to $\mathcal{F}.a$ (the residue arguments remain unchanged) and adding ε as an extra argument adorned in the same way as the residue arguments. Since the argument order of an abstraction function is arbitrary, we order ε last among a's dependencies. Cases where the abstract variable has different numbers of dependencies are straightforward and omitted from Figure 17 (for example, if a does not depend on ε in E, then refunctionalization does not add the extra argument $\tilde{\varepsilon}$).

X distributes over operators. Line 3 shows the equation for an arbitrary binary operator; operators with other arities are similar. Except as stated below, X also distributes over quantifications over scalars (line 4), concrete fields (line 5), and individual residue variables, with (line 8) and without (line 6) equality antecedents. Quantifications over ε and equality antecedents for ε are introduced during the refunctionalization of quantifications over shared residues (lines 7 and 9).

For quantifications arising from individual-residue modification constraints (line 10), X applies recursively to the subpredicate Q. Shared-residue modification constraints (line 11) are similar, but in this case, X also introduces a special modification constraint for ε, namely $modcon_E(w, \varepsilon, sres)$, where w comes from the marker "w:" around the given modification constraint for $sres$. Note that the second and third parameters to this use of $modcon_E$ are different, whereas they always coincide when $modcon_E$ is generated in a VC. (The reason for the special modification constraint is as follows. In order to prove that X preserves the validity of VCs, see (75), refunctionalization must add some modification constraint for ε. In the main proof of Soundness Lemma C, we will get the desired $modcon_E(w, \varepsilon, \varepsilon)$ by applying the Chain of Equalities Corollary. At that time, $modcon_E(w, \varepsilon, sres)$ is absorbed, because it turns out to be weaker than $modcon_E(w, \varepsilon, \varepsilon)$.)

Refunctionalization simply replaces the pointwise axioms (line 12) and rep axioms (line 13) generated in D by those that would be generated in E.

$$
\begin{array}{rlcl}
0: & X(s) & = & s \\
1: & X(\tilde{c}) & = & \tilde{c} \\
2: & X(\mathcal{F}.a(\widetilde{sres},\, \widetilde{res}.a,\, Q)) & = & \mathcal{F}.a(\widetilde{sres},\, \widetilde{res}.a,\, X(Q),\, \tilde{\varepsilon}) \\
3: & X(Q0\ \mathbf{op}\ Q1) & = & X(Q0)\ \mathbf{op}\ X(Q1) \\
4: & X(\langle\, \forall\, s \ ::\ Q \,\rangle) & = & \langle\, \forall\, s \ ::\ X(Q) \,\rangle \\
5: & X(\langle\, \forall\, \tilde{c} \ ::\ Q \,\rangle) & = & \langle\, \forall\, \tilde{c} \ ::\ X(Q) \,\rangle \\
6: & X(\langle\, \forall\, \widetilde{res}.a \ ::\ Q \,\rangle) & = & \langle\, \forall\, \widetilde{res}.a \ ::\ X(Q) \,\rangle \\
7: & X(\langle\, \forall\, \widetilde{sres} \ ::\ Q \,\rangle) & = & \langle\, \forall\, \tilde{\varepsilon} \ ::\ \langle\, \forall\, \widetilde{sres} \ ::\ X(Q) \,\rangle\rangle \\
8: & \multicolumn{3}{l}{X(\langle\, \forall\, \widetilde{res}.a \ ::\ \widetilde{res}.a = \overline{res}.a \ \Rightarrow\ Q \,\rangle) \ =} \\
 & & & \langle\, \forall\, \widetilde{res}.a \ ::\ \widetilde{res}.a = \overline{res}.a \ \Rightarrow\ X(Q) \,\rangle \\
9: & \multicolumn{3}{l}{X(\langle\, \forall\, \widetilde{sres} \ ::\ \widetilde{sres} = \overline{sres} \ \Rightarrow\ Q \,\rangle) \ =} \\
 & \multicolumn{3}{r}{\langle\, \forall\, \tilde{\varepsilon} \ ::\ \tilde{\varepsilon} = \bar{\varepsilon} \ \Rightarrow\ \langle\, \forall\, \widetilde{sres} \ ::\ \widetilde{sres} = \overline{sres} \ \Rightarrow\ X(Q) \,\rangle\rangle} \\
10: & \multicolumn{3}{l}{X(w{:}\langle\, \forall\, s \ ::\ \grave{res}.a[s] = \acute{res}.a[s] \ \vee\ Q \,\rangle) \ =} \\
 & & & w{:}\langle\, \forall\, s \ ::\ \grave{res}.a[s] = \acute{res}.a[s] \ \vee\ X(Q) \,\rangle) \\
11: & \multicolumn{3}{l}{X(w{:}\langle\, \forall\, s \ ::\ \grave{sres}[s] = \acute{sres}[s] \ \vee\ Q \,\rangle) \ =} \\
 & & & w{:}\langle\, \forall\, s \ ::\ \grave{sres}[s] = \acute{sres}[s] \ \vee\ X(Q) \,\rangle) \ \wedge \\
 & & & modcon_E(w,\, \varepsilon,\, sres) \\
12: & X(\text{ formula (25), page 84 }) & = & \langle\, \forall\, t,\, \grave{sres},\, \acute{sres},\, \grave{res}.a,\, \acute{res}.a,\, \grave{f},\, \acute{f},\, \grave{\varepsilon},\, \acute{\varepsilon} \ :: \\
 & & & \qquad \grave{sres}[t] = \acute{sres}[t] \ \wedge \\
 & & & \qquad \grave{res}.a[t] = \acute{res}.a[t] \ \wedge \\
 & & & \qquad \grave{f}[t] = \acute{f}[t] \ \wedge\ \grave{\varepsilon}[t] = \acute{\varepsilon}[t] \ \Rightarrow \\
 & & & \qquad\quad \mathcal{F}.a(\grave{sres},\, \grave{res}.a,\, \grave{f},\, \grave{\varepsilon})[t] = \\
 & & & \qquad\quad \mathcal{F}.a(\acute{sres},\, \acute{res}.a,\, \acute{f},\, \acute{\varepsilon})[t] \,\rangle \\
13: & \multicolumn{3}{l}{X(\langle\, \forall\, t{:}\,T,\, sres,\, res.a, f \ ::\ t \neq \mathbf{nil} \ \Rightarrow\ \mathcal{F}.a(sres,\, res.a, f)[t] = e \,\rangle) \ =} \\
 & \multicolumn{3}{r}{\langle\, \forall\, t{:}\,T,\, sres,\, res.a, f,\, \varepsilon \ ::\ t \neq \mathbf{nil} \ \Rightarrow\ \mathcal{F}.a(sres,\, res.a, f,\, \varepsilon)[t] = e \,\rangle}
\end{array}
$$

Figure 17: The definition of the refunctionalization meta function X.

A25 Properties of X

The refunctionalization meta function X enjoys several distributive and commutative properties. We list these properties in this section for use in the main proof of Soundness Lemma C. After we give the main proof in the next section, we will spend a number of sections giving the proofs of the properties listed here.

The properties in this section are stated in the context of any D, ε, and E that satisfy $Extend(D, \varepsilon, E)$.

For any user expression e in D,

$$X(F_D(e)) \;=\; F_E(e) \tag{68}$$

For the background predicate, rep axioms, and pointwise axioms,

$$[X(BP_D) \;\Leftarrow\; BP_E] \tag{69}$$

$$X(Rep_D) \;=\; Rep_E \tag{70}$$

$$X(PW_D) \;=\; PW_E \tag{71}$$

For any modifies list w in D,

$$[X(mc_D(w)) \;\wedge\; modcon_E(w, \varepsilon, \varepsilon) \;\equiv\; mc_E(w)] \tag{72}$$

$$[X(mc_D(w)) \;\Rightarrow\; mcr_E(w)] \tag{73}$$

where mcr_E is the meta function defined by (63) on page 127. For any command C in D and any predicate Q in D,

$$[X(wlp_D(C, Q)) \;\Rightarrow\; wlp_E(C, X(Q))] \tag{74}$$

For any implementation C in D of a method m,

$$[VC_D(m, C)] \;\Rightarrow\; [X(VC_D(m, C))] \tag{75}$$

Properties (69) through (74) essentially say that meta function X refunctionalizes formulas in the way we intended, that is, that X transforms a VC generated in D into a formula quite similar to the VC that would have been generated in E (the only differences are that X doesn't conjoin the background-predicate axioms that are generated only in E and that X introduces $modcon_E(w, \varepsilon, sres)$ instead of the proper modification constraint $modcon_E(w, \varepsilon, \varepsilon)$).

Property (75), which states that X preserves the validity of VCs, plays a vital rôle in our proof of the Soundness Theorem. The essence of the Soundness Theorem is captured by Soundness Lemma C, and at the heart of its proof are two pieces: the Chain of Equalities Corollary and X property (75).

A26 The main proof of Soundness Lemma C

Using the yet-to-be-proved properties of X stated in the previous section, we give the proof of Soundness Lemma C.

Proof of Soundness Lemma C. Let D, ε, E, m, and C satisfy the antecedent of Soundness Lemma C. Let zz be the concrete fields and residue variables in D, and suppose the specification for m is

requires p **modifies** w **ensures** q

We calculate,

$$
\begin{array}{ll}
 & [VC_D(m, C)] \\
\Rightarrow & \quad \{ \text{ (75): } X \text{ preserves the validity of VCs } \} \\
 & [X(VC_D(m, C))] \\
= & \quad \{ \ VC_D \ \} \\
 & [X(BP_D \wedge Rep_D \wedge PW_D \wedge F_D(p) \Rightarrow \langle \forall \grave{zz} :: \grave{zz} = zz \Rightarrow \\
 & \qquad\qquad wlp_D(C, \langle \forall \acute{zz} :: \acute{zz} = zz \Rightarrow F_D(q) \wedge mc_D(w) \rangle) \rangle] \\
= & \quad \{ \text{ definition of } X \text{ on operators } \Rightarrow \text{ and } \wedge \ \} \\
 & [X(BP_D) \wedge X(Rep_D) \wedge X(PW_D) \wedge X(F_D(p)) \Rightarrow \\
 & \ X(\langle \forall \grave{zz} :: \grave{zz} = zz \Rightarrow \\
 & \qquad\qquad wlp_D(C, \langle \forall \acute{zz} :: \acute{zz} = zz \Rightarrow F_D(q) \wedge mc_D(w) \rangle) \rangle)] \\
\Rightarrow & \quad \{ \text{ (69): } X \text{ and } BP \ \} \\
 & [BP_E \wedge X(Rep_D) \wedge X(PW_D) \wedge X(F_D(p)) \Rightarrow \\
 & \ X(\langle \forall \grave{zz} :: \grave{zz} = zz \Rightarrow \\
 & \qquad\qquad wlp_D(C, \langle \forall \acute{zz} :: \acute{zz} = zz \Rightarrow F_D(q) \wedge mc_D(w) \rangle) \rangle)] \\
= & \quad \{ \text{ (70) and (71) and (68): } X, Rep, PW, \text{ and } F \ \} \\
 & [BP_E \wedge Rep_E \wedge PW_E \wedge F_E(p) \Rightarrow \\
 & \ X(\langle \forall \grave{zz} :: \grave{zz} = zz \Rightarrow \\
 & \qquad\qquad wlp_D(C, \langle \forall \acute{zz} :: \acute{zz} = zz \Rightarrow F_D(q) \wedge mc_D(w) \rangle) \rangle)] \\
= & \quad \{ \ X \ \} \\
 & [BP_E \wedge Rep_E \wedge PW_E \wedge F_E(p) \Rightarrow \langle \forall \grave{zz}, \grave{\varepsilon} :: \grave{zz} = zz \wedge \grave{\varepsilon} = \varepsilon \Rightarrow \\
 & \qquad X(wlp_D(C, \langle \forall \acute{zz} :: \acute{zz} = zz \Rightarrow F_D(q) \wedge mc_D(w) \rangle)) \rangle] \\
\Rightarrow & \quad \{ \text{ (74): } X \text{ and } wlp \ \} \\
 & [BP_E \wedge Rep_E \wedge PW_E \wedge F_E(p) \Rightarrow \langle \forall \grave{zz}, \grave{\varepsilon} :: \grave{zz} = zz \wedge \grave{\varepsilon} = \varepsilon \Rightarrow \\
 & \qquad wlp_E(C, X(\langle \forall \acute{zz} :: \acute{zz} = zz \Rightarrow F_D(q) \wedge mc_D(w) \rangle)) \rangle] \\
= & \quad \{ \ X \ \}
\end{array}
$$

$$
\begin{array}{ll}
 & [BP_E \ \wedge \ Rep_E \ \wedge \ PW_E \ \wedge \ F_E(p) \ \Rightarrow \ \langle \forall \grave{zz}, \grave{\varepsilon} \ :: \ \grave{zz} = zz \ \wedge \ \grave{\varepsilon} = \varepsilon \ \Rightarrow \\
 & \quad wlp_E(C, \langle \forall \acute{zz}, \acute{\varepsilon} \ :: \ \acute{zz} = zz \ \wedge \ \acute{\varepsilon} = \varepsilon \ \Rightarrow \ X(F_D(q) \ \wedge \ mc_D(w)) \ \rangle) \ \rangle] \\
= & \quad \{ \ \text{definition of } X \text{ on } \wedge \text{, and (68)} \ \} \\
 & [BP_E \ \wedge \ Rep_E \ \wedge \ PW_E \ \wedge \ F_E(p) \ \Rightarrow \ \langle \forall \grave{zz}, \grave{\varepsilon} \ :: \ \grave{zz} = zz \ \wedge \ \grave{\varepsilon} = \varepsilon \ \Rightarrow \\
 & \quad wlp_E(C, \langle \forall \acute{zz}, \acute{\varepsilon} \ :: \ \acute{zz} = zz \ \wedge \ \acute{\varepsilon} = \varepsilon \ \Rightarrow \ F_E(q) \ \wedge \ X(mc_D(w)) \ \rangle) \ \rangle] \\
= & \quad \{ \ \text{(38): monotonicity of } wlp_E \text{, since by (73),} \\
 & \qquad [F_E(q) \ \wedge \ X(mc_D(w)) \ \Rightarrow \ mcr_E(w)] \ \} \\
 & [BP_E \ \wedge \ Rep_E \ \wedge \ PW_E \ \wedge \ F_E(p) \ \Rightarrow \ \langle \forall \grave{zz}, \grave{\varepsilon} \ :: \ \grave{zz} = zz \ \wedge \ \grave{\varepsilon} = \varepsilon \ \Rightarrow \\
 & \quad wlp_E(C, \langle \forall \acute{zz}, \acute{\varepsilon} \ :: \ \acute{zz} = zz \ \wedge \ \acute{\varepsilon} = \varepsilon \ \Rightarrow \ F_E(q) \ \wedge \ X(mc_D(w)) \ \rangle) \ \rangle] \ \wedge \\
 & [BP_E \ \wedge \ Rep_E \ \wedge \ PW_E \ \wedge \ F_E(p) \ \Rightarrow \ \langle \forall \grave{zz}, \grave{\varepsilon} \ :: \ \grave{zz} = zz \ \wedge \ \grave{\varepsilon} = \varepsilon \ \Rightarrow \\
 & \quad wlp_E(C, \langle \forall \acute{zz}, \acute{\varepsilon} \ :: \ \acute{zz} = zz \ \wedge \ \acute{\varepsilon} = \varepsilon \ \Rightarrow \ mcr_E(w) \ \rangle) \ \rangle] \\
\Rightarrow & \quad \{ \ \text{(36): } wlp_D \text{ implies } ulp_D \ \} \\
 & [BP_E \ \wedge \ Rep_E \ \wedge \ PW_E \ \wedge \ F_E(p) \ \Rightarrow \ \langle \forall \grave{zz}, \grave{\varepsilon} \ :: \ \grave{zz} = zz \ \wedge \ \grave{\varepsilon} = \varepsilon \ \Rightarrow \\
 & \quad wlp_E(C, \langle \forall \acute{zz}, \acute{\varepsilon} \ :: \ \acute{zz} = zz \ \wedge \ \acute{\varepsilon} = \varepsilon \ \Rightarrow \ F_E(q) \ \wedge \ X(mc_D(w)) \ \rangle) \ \rangle] \ \wedge \\
 & [BP_E \ \wedge \ Rep_E \ \wedge \ PW_E \ \wedge \ F_E(p) \ \Rightarrow \ \langle \forall \grave{zz}, \grave{\varepsilon} \ :: \ \grave{zz} = zz \ \wedge \ \grave{\varepsilon} = \varepsilon \ \Rightarrow \\
 & \quad ulp_E(C, \langle \forall \acute{zz}, \acute{\varepsilon} \ :: \ \acute{zz} = zz \ \wedge \ \acute{\varepsilon} = \varepsilon \ \Rightarrow \ mcr_E(w) \ \rangle) \ \rangle] \\
\Rightarrow & \quad \{ \ \text{Chain of Equalities Corollary (page 127) with } zz := (zz, \varepsilon) \ \} \\
 & [BP_E \ \wedge \ Rep_E \ \wedge \ PW_E \ \wedge \ F_E(p) \ \Rightarrow \ \langle \forall \grave{zz}, \grave{\varepsilon} \ :: \ \grave{zz} = zz \ \wedge \ \grave{\varepsilon} = \varepsilon \ \Rightarrow \\
 & \quad wlp_E(C, \langle \forall \acute{zz}, \acute{\varepsilon} \ :: \ \acute{zz} = zz \ \wedge \ \acute{\varepsilon} = \varepsilon \ \Rightarrow \ F_E(q) \ \wedge \ X(mc_D(w)) \ \rangle) \ \rangle] \ \wedge \\
 & [BP_E \ \wedge \ Rep_E \ \wedge \ PW_E \ \wedge \ F_E(p) \ \Rightarrow \ \langle \forall \grave{zz}, \grave{\varepsilon} \ :: \ \grave{zz} = zz \ \wedge \ \grave{\varepsilon} = \varepsilon \ \Rightarrow \\
 & \quad ulp_E(C, \langle \forall \acute{zz}, \acute{\varepsilon} \ :: \ \acute{zz} = zz \ \wedge \ \acute{\varepsilon} = \varepsilon \ \Rightarrow \ modcon_E(w, \varepsilon, \varepsilon) \ \rangle) \ \rangle] \\
= & \quad \{ \ \text{(39): conjunctivity, } wlp_E \text{, and } ulp_E \ \} \\
 & [BP_E \ \wedge \ Rep_E \ \wedge \ PW_E \ \wedge \ F_E(p) \ \Rightarrow \ \langle \forall \grave{zz}, \grave{\varepsilon} \ :: \ \grave{zz} = zz \ \wedge \ \grave{\varepsilon} = \varepsilon \ \Rightarrow \\
 & \quad wlp_E(C, \langle \forall \acute{zz}, \acute{\varepsilon} \ :: \ \acute{zz} = zz \ \wedge \ \acute{\varepsilon} = \varepsilon \ \Rightarrow \\
 & \qquad\qquad F_E(q) \ \wedge \ X(mc_D(w)) \ \wedge \ modcon_E(w, \varepsilon, \varepsilon) \ \rangle) \ \rangle] \\
= & \quad \{ \ \text{(72): } X \text{ and modification constraints} \ \} \\
 & [BP_E \ \wedge \ Rep_E \ \wedge \ PW_E \ \wedge \ F_E(p) \ \Rightarrow \ \langle \forall \grave{zz}, \grave{\varepsilon} \ :: \ \grave{zz} = zz \ \wedge \ \grave{\varepsilon} = \varepsilon \ \Rightarrow \\
 & \quad wlp_E(C, \langle \forall \acute{zz}, \acute{\varepsilon} \ :: \ \acute{zz} = zz \ \wedge \ \acute{\varepsilon} = \varepsilon \ \Rightarrow \ F_E(q) \ \wedge \ mc_E(w) \ \rangle) \ \rangle] \\
= & \quad \{ \ VC_E \text{, since } zz, \varepsilon \text{ are the concrete fields and residue variables} \\
 & \qquad \text{in } E \ \} \\
 & [VC_E(m, C)]
\end{array}
$$

■

Now, all that remains is proving the eight properties of X.

A27 X and user expressions

In this section, we prove X property (68): for any D, ε, and E such that $Extend(D, \varepsilon, E)$ and any user expression e,

$$X(F_D(e)) = F_E(e)$$

Proof of (68). We actually prove this property not just for user expressions, but for any expression e in the domain of F_D except residue variables. The proof is by induction over the shape of e.

CASE s:

$$\begin{array}{ll} & X(F_D(s)) \\ = & \qquad \{ \ F_D \text{ and } X \ \} \\ & s \\ = & \qquad \{ \ F_E \ \} \\ & F_E(s) \end{array}$$

CASE $\tilde{c}$:

$$\begin{array}{ll} & X(F_D(\tilde{c})) \\ = & \qquad \{ \ F_D \text{ and } X \ \} \\ & \tilde{c} \\ = & \qquad \{ \ F_E \ \} \\ & F_E(\tilde{c}) \end{array}$$

CASE $\tilde{a}$:

$$\begin{array}{ll} & X(F_D(\tilde{a})) \\ = & \qquad \{ \ F_D \ \} \\ & X(\mathcal{F}.a(s\tilde{res}, re\tilde{s}.a, F_D(\tilde{f}))) \\ = & \qquad \{ \ X \ \} \\ & \mathcal{F}.a(s\tilde{res}, re\tilde{s}.a, X(F_D(\tilde{f})), \tilde{\varepsilon}) \\ = & \qquad \{ \ \text{induction hypothesis} \ \} \\ & \mathcal{F}.a(s\tilde{res}, re\tilde{s}.a, F_E(\tilde{f}), \tilde{\varepsilon}) \\ = & \qquad \{ \ F_E \text{ on concrete field } \tilde{\varepsilon} \ \} \\ & \mathcal{F}.a(s\tilde{res}, re\tilde{s}.a, F_E(\tilde{f}), F_E(\tilde{\varepsilon})) \\ = & \qquad \{ \ F_E \ \} \\ & F_E(\tilde{a}) \end{array}$$

CASE $e0\,\mathbf{op}\,e1$:

$$
\begin{array}{ll}
& X(F_D(e0\,\mathbf{op}\,e1)) \\
= & \quad \{ \quad F_D \quad \} \\
& X(F_D(e0)\,\mathbf{op}\,F_D(e1)) \\
= & \quad \{ \quad X \quad \} \\
& X(F_D(e0))\,\mathbf{op}\,X(F_D(e1)) \\
= & \quad \{ \quad \text{induction hypothesis, twice} \quad \} \\
& F_E(e0)\,\mathbf{op}\,F_E(e1) \\
= & \quad \{ \quad F_E \quad \} \\
& F_E(e0\,\mathbf{op}\,e1)
\end{array}
$$

CASE $\langle \forall s :: e \rangle$:

$$
\begin{array}{ll}
& X(F_D(\langle \forall s :: e \rangle)) \\
= & \quad \{ \quad F_D \quad \} \\
& X(\langle \forall s :: F_D(e) \rangle) \\
= & \quad \{ \quad X \quad \} \\
& \langle \forall s :: X(F_D(e)) \rangle \\
= & \quad \{ \quad \text{induction hypothesis} \quad \} \\
& \langle \forall s :: F_E(e) \rangle \\
= & \quad \{ \quad F_E \quad \} \\
& F_E(\langle \forall s :: e \rangle)
\end{array}
$$

That concludes the proof of property (68). ■

A28 X and the background predicate

In this section, we prove X property (69): for any D, ε, and E such that $Extend(D, \varepsilon, E)$,

$$[X(BP_D) \ \Leftarrow\ BP_E]$$

Proof of (69). Since the background predicate BP_D is generated as a function of the type declarations in D, independent of what other declarations D contains, it follows that BP_D does not mention abstraction functions, residue variables, residue modification constraints, pointwise axioms, or rep axioms. Thus, we have

$$X(BP_D) \ =\ BP_D$$

We calculate,

$$
\begin{array}{ll}
 & X(BP_D) \\
= & \quad \{ \text{ observation above } \} \\
 & BP_D \\
\Leftarrow & \quad \{ \ D \subseteq E \text{, which follows from } \mathit{Extend}(D, \varepsilon, E) \text{, and (26) } \} \\
 & BP_E
\end{array}
$$

■

A29 X and rep and pointwise axioms

In this section, we prove X properties (70) and (71): for any D, ε, and E such that $\mathit{Extend}(D, \varepsilon, E)$,

$$X(Rep_D) \ = \ Rep_E \qquad \text{and}$$

$$X(PW_D) \ = \ PW_E$$

Proof of (70) and (71). Let D, ε, and E satisfy $\mathit{Extend}(D, \varepsilon, E)$. By the definition of X, the refunctionalization of any pointwise axiom or rep axiom in D equals the corresponding axiom in E. Since D and E coincide in their abstract field and rep declarations, (70) and (71) follow. ■

A30 X and modification constraints

In this section, we prove properties (72) and (73), which relate X and modification constraints. We start by proving a three-part lemma that will be useful in the proofs of these properties.

Lemma. For any D, ε, and E such that $\mathit{Extend}(D, \varepsilon, E)$, any modifies list w in D, any field f in D, and any individual residue variable $res.a$ in D,

$$X(modcon_D(w, f, f)) \ = \ modcon_E(w, f, f) \tag{76}$$

$$X(modcon_D(w, res.a, res.a)) = modcon_E(w, res.a, res.a) \tag{77}$$

$$X(modcon_D(w, sres, sres)) \ = \ modcon_E(w, sres, sres) \ \wedge \ modcon_E(w, \varepsilon, sres) \tag{78}$$

Proof. For any field f in D, we calculate,

$$
\begin{array}{ll}
 & X(modcon_D(w, f, f)) \\
= & \quad \{ \ \text{(24): definition of } modcon_D \ \} \\
 & X(w{:}\langle \forall s :: F_D(\grave{f}[s] = \acute{f}[s] \ \vee \ modpoint_D(f, w, s)) \rangle) \\
= & \quad \{ \ X \text{ on unary operator ``} w{:}\text{'' and on scalar quantification } \} \\
 & w{:}\langle \forall s :: X(F_D(\grave{f}[s] = \acute{f}[s] \ \vee \ modpoint_D(f, w, s))) \rangle \\
= & \quad \{ \ \text{(68): } X \circ F_D = F_E \ \} \\
 & w{:}\langle \forall s :: F_E(\grave{f}[s] = \acute{f}[s] \ \vee \ modpoint_D(f, w, s)) \rangle \\
= & \quad \{ \ \text{lemma (33) with } x := f \ \} \\
 & w{:}\langle \forall s :: F_E(\grave{f}[s] = \acute{f}[s] \ \vee \ modpoint_E(f, w, s)) \rangle \\
= & \quad \{ \ \text{(24): definition of } modcon_E \ \} \\
 & modcon_E(w, f, f)
\end{array}
$$

For any individual residue variable $res.a$, we calculate,

$$
\begin{array}{ll}
 & X(modcon_D(w, res.a, res.a)) \\
= & \quad \{ \ \text{(24): definition of } modcon_D \ \} \\
 & X(w{:}\langle \forall s :: F_D(\grave{res}.a[s] = \acute{res}.a[s] \ \vee \ modpoint_D(res.a, w, s)) \rangle) \\
= & \quad \{ \ F_D \ \} \\
 & X(w{:}\langle \forall s :: \grave{res}.a[s] = \acute{res}.a[s] \ \vee \ F_D(modpoint_D(res.a, w, s)) \rangle) \\
= & \quad \{ \ X \text{ on individual-residue modification constraint } \} \\
 & w{:}\langle \forall s :: \grave{res}.a[s] = \acute{res}.a[s] \ \vee \ X(F_D(modpoint_D(res.a, w, s))) \rangle \\
= & \quad \{ \ \text{(68): } X \circ F_D = F_E \ \} \\
 & w{:}\langle \forall s :: \grave{res}.a[s] = \acute{res}.a[s] \ \vee \ F_E(modpoint_D(res.a, w, s)) \rangle \\
= & \quad \{ \ \text{lemma (33) with } x := res.a \ \} \\
 & w{:}\langle \forall s :: \grave{res}.a[s] = \acute{res}.a[s] \ \vee \ F_E(modpoint_E(res.a, w, s)) \rangle \\
= & \quad \{ \ F_E \ \} \\
 & w{:}\langle \forall s :: F_E(\grave{res}.a[s] = \acute{res}.a[s] \ \vee \ modpoint_E(res.a, w, s)) \rangle \\
= & \quad \{ \ \text{(24): definition of } modcon_E \ \} \\
 & modcon_E(w, res.a, res.a)
\end{array}
$$

For the shared residue variable, we calculate,

$$
\begin{array}{ll}
 & X(modcon_D(w, sres, sres)) \\
= & \quad \{ \ \text{(24): definition of } modcon_D \ \} \\
 & X(w{:}\langle \forall s :: F_D(\grave{sres}[s] = \acute{sres}[s] \ \vee \ modpoint_D(sres, w, s)) \rangle) \\
= & \quad \{ \ F_D \ \} \\
 & X(w{:}\langle \forall s :: \grave{sres}[s] = \acute{sres}[s] \ \vee \ F_D(modpoint_D(sres, w, s)) \rangle) \\
= & \quad \{ \ X \text{ on shared-residue modification constraint } \}
\end{array}
$$

$$w{:}\langle\, \forall s \ :: \ \grave{sres}[s] = \acute{sres}[s] \ \vee \ X(F_D(modpoint_D(sres, w, s))) \,\rangle \ \wedge$$
$$modcon_E(w, \varepsilon, sres)$$
$$= \qquad \{ \ (68){:}\ X \circ F_D = F_E \ \}$$
$$w{:}\langle\, \forall s \ :: \ \grave{sres}[s] = \acute{sres}[s] \ \vee \ F_E(modpoint_D(sres, w, s)) \,\rangle \ \wedge$$
$$modcon_E(w, \varepsilon, sres)$$
$$= \qquad \{ \ \text{lemma (33) with } x := sres \ \}$$
$$w{:}\langle\, \forall s \ :: \ \grave{sres}[s] = \acute{sres}[s] \ \vee \ F_E(modpoint_E(sres, w, s)) \,\rangle \ \wedge$$
$$modcon_E(w, \varepsilon, sres)$$
$$= \qquad \{ \ F_E \ \}$$
$$w{:}\langle\, \forall s \ :: \ F_E(\grave{sres}[s] = \acute{sres}[s] \ \vee \ modpoint_E(sres, w, s)) \,\rangle \ \wedge$$
$$modcon_E(w, \varepsilon, sres)$$
$$= \qquad \{ \ (24){:}\ \text{definition of } modcon_E \ \}$$
$$modcon_E(w, sres, sres) \ \wedge \ modcon_E(w, \varepsilon, sres)$$

That proves the three parts of the lemma. ■

Now for the proof of X property (72): for any D, ε, and E such that $Extend(D, \varepsilon, E)$ and any modifies list w in D,

$$[X(mc_D(w)) \ \wedge \ modcon_E(w, \varepsilon, \varepsilon) \ \equiv \ mc_E(w)]$$

Proof of (72). For any D, ε, and E such that $Extend(D, \varepsilon, E)$ and any modifies list w in D, we calculate

$$X(mc_D(w)) \ \wedge \ modcon_E(w, \varepsilon, \varepsilon)$$
$$= \qquad \{ \ mc_D \ \}$$
$$X(\langle\, \textstyle\bigwedge x \mid x \in D \ :: \ modcon_D(w, x, x) \,\rangle) \ \wedge \ modcon_E(w, \varepsilon, \varepsilon)$$
$$= \qquad \{ \ \text{definition of } X \text{ on } \wedge \ \}$$
$$\langle\, \textstyle\bigwedge x \mid x \in D \ :: \ X(modcon_D(w, x, x)) \,\rangle \ \wedge \ modcon_E(w, \varepsilon, \varepsilon)$$

= { split range: let f range over fields, let a range over abstract fields, and let (as usual) $res.a$ denote the residue variable associated with a }

$$\langle\, \textstyle\bigwedge f \mid f \in D \ :: \ X(modcon_D(w, f, f)) \,\rangle \ \wedge$$
$$\langle\, \textstyle\bigwedge a \mid a \in D \ :: \ X(modcon_D(w, res.a, res.a)) \,\rangle \ \wedge$$
$$X(modcon_D(w, sres, sres)) \ \wedge \ modcon_E(w, \varepsilon, \varepsilon)$$
$$= \qquad \{ \ \text{lemma (76)–(77)–(78)} \ \}$$
$$\langle\, \textstyle\bigwedge f \mid f \in D \ :: \ modcon_E(w, f, f) \,\rangle \ \wedge$$
$$\langle\, \textstyle\bigwedge a \mid a \in D \ :: \ modcon_E(w, res.a, res.a) \,\rangle \ \wedge$$
$$modcon_E(w, sres, sres) \ \wedge \ modcon_E(w, \varepsilon, sres) \ \wedge \ modcon_E(w, \varepsilon, \varepsilon)$$
$$= \qquad \{ \ \text{lemma (34): } [modcon_E(w, \varepsilon, \varepsilon) \ \Rightarrow \ modcon_E(w, \varepsilon, sres)] \ \}$$

$$\begin{array}{ll}
& \langle \bigwedge f \mid f \in D :: \ modcon_E(w, f, f) \ \rangle \ \wedge \\
& \langle \bigwedge a \mid a \in D :: \ modcon_E(w, res.a, res.a) \ \rangle \ \wedge \\
& modcon_E(w, sres, sres) \ \wedge \ modcon_E(w, \varepsilon, \varepsilon) \\
= & \qquad \{ \ \text{combine ranges} \ \} \\
& \langle \bigwedge x \mid x \in E :: \ modcon_E(w, x, x) \ \rangle \\
= & \qquad \{ \ mc_E \ \} \\
& mc_E(w)
\end{array}$$

■

And now the proof of X property (73): for any D, ε, and E such that $Extend(D, \varepsilon, E)$ and any modifies list w in D,

$$[X(mc_D(w)) \ \Rightarrow \ mcr_E(w)]$$

Proof of (73). Let D, ε, and E satisfy $Extend(D, \varepsilon, E)$, let w be any modifies list D, and let a by any abstract field in E such that $a \ \mathbf{on}_E \ \varepsilon$. Since D and E coincide in their declarations of abstract fields, a is also in D. Let $res.a$ denote the individual residue variable of a. Then, $modcon_E(w, res.a, res.a)$ denotes an arbitrary conjunct of $mcr_E(w)$. We calculate,

$$\begin{array}{ll}
& modcon_E(w, res.a, res.a) \\
= & \qquad \{ \ \text{lemma (77)} \ \} \\
& X(modcon_D(w, res.a, res.a)) \\
\Leftarrow & \qquad \{ \ \text{strengthening, since } res.a \in D \ \} \\
& \langle \bigwedge x \mid x \in D :: \ X(modcon_D(w, x, x)) \ \rangle \\
= & \qquad \{ \ \text{definition of } X \text{ on } \wedge \ \} \\
& X(\langle \bigwedge x \mid x \in D :: \ modcon_D(w, x, x) \ \rangle) \\
= & \qquad \{ \ mc_D \ \} \\
& X(mc_D(w))
\end{array}$$

■

A31 X and wlp

In this section, we prove X property (74): for any D, ε, and E such that $Extend(D, \varepsilon, E)$, any command C in D, and any predicate Q in D,

$$[X(wlp_D(C, Q)) \ \Rightarrow \ wlp_E(C, X(Q))]$$

Proof of (74). Let D, ε, and E satisfy $Extend(D, \varepsilon, E)$, let C be a command in D, and let Q be a predicate in D. We proceed by induction over the shape of C.

CASE $s := e$:

$$
\begin{array}{ll}
 & X(wlp_D(s := e, Q)) \\
= & \quad \{ \ \ wlp_D \ \ \} \\
 & X(\langle \forall s' :: s' = F_D(e) \ \Rightarrow \ \langle \forall s :: s = s' \ \Rightarrow \ Q \rangle\rangle) \\
= & \quad \{ \ \ X \ \ \} \\
 & \langle \forall s' :: s' = X(F_D(e)) \ \Rightarrow \ \langle \forall s :: s = s' \ \Rightarrow \ X(Q) \rangle\rangle \\
= & \quad \{ \ \ (68)\text{: } X \circ F_D = F_E \ \ \} \\
 & \langle \forall s' :: s' = F_E(e) \ \Rightarrow \ \langle \forall s :: s = s' \ \Rightarrow \ X(Q) \rangle\rangle \\
= & \quad \{ \ \ wlp_E \ \ \} \\
 & wlp_E(s := e, X(Q))
\end{array}
$$

CASE $c[e0] := e1$:

$$
\begin{array}{ll}
 & X(wlp_D(c[e0] := e1, Q)) \\
= & \quad \{ \ \ wlp_D \ \ \} \\
 & X(\langle \forall c' :: c' = store(c, F_D(e0), F_D(e1)) \ \Rightarrow \ \langle \forall c :: c = c' \ \Rightarrow \ Q \rangle\rangle) \\
= & \quad \{ \ \ X \ \ \} \\
 & \langle \forall c' :: c' = store(c, X(F_D(e0)), X(F_D(e1))) \ \Rightarrow \ \langle \forall c :: c = c' \ \Rightarrow \ X(Q) \rangle\rangle \\
= & \quad \{ \ \ (68)\text{: } X \circ F_D = F_E \text{, twice} \ \ \} \\
 & \langle \forall c' :: c' = store(c, F_E(e0), F_E(e1)) \ \Rightarrow \ \langle \forall c :: c = c' \ \Rightarrow \ X(Q) \rangle\rangle \\
= & \quad \{ \ \ wlp_E \ \ \} \\
 & wlp_E(c[e0] := e1, X(Q))
\end{array}
$$

CASE **assert** e:

$$
\begin{array}{ll}
 & X(wlp_D(\textbf{assert}\ e, Q)) \\
= & \quad \{ \ \ wlp_D \ \ \} \\
 & X(F_D(e) \ \wedge \ Q) \\
= & \quad \{ \ \ X \text{ on } \wedge \text{, and (68): } X \circ F_D = F_E \ \ \} \\
 & F_E(e) \ \wedge \ X(Q) \\
= & \quad \{ \ \ wlp_E \ \ \} \\
 & wlp_E(\textbf{assert}\ e, X(Q))
\end{array}
$$

CASE **assume** e:

$\quad X(wlp_D(\mathbf{assume}\ e, Q))$
$=\qquad \{\ wlp_D\ \}$
$\quad X(F_D(e)\ \Rightarrow\ Q)$
$=\qquad \{\ X\ \text{on}\ \Rightarrow\ \text{, and (68): } X \circ F_D = F_E\ \}$
$\quad F_E(e)\ \Rightarrow\ X(Q)$
$=\qquad \{\ wlp_E\ \}$
$\quad wlp_E(\mathbf{assume}\ e, X(Q))$

CASE $C0\ ;\ C1$:

$\quad X(wlp_D(C0\ ;\ C1, Q))$
$=\qquad \{\ wlp_D\ \}$
$\quad X(wlp_D(C0, wlp_D(C1, Q)))$
$\Rightarrow\qquad \{\ \text{induction hypothesis}\ \}$
$\quad wlp_E(C0, X(wlp_D(C1, Q)))$
$\Rightarrow\qquad \{\ \text{induction hypothesis, since (38): } wlp_E \text{ is monotonic}\ \}$
$\quad wlp_E(C0, wlp_E(C1, X(Q)))$
$=\qquad \{\ wlp_E\ \}$
$\quad wlp_E(C0\ ;\ C1, X(Q))$

CASE $C0\ \square\ C1$:

$\quad X(wlp_D(C0\ \square\ C1, Q))$
$=\qquad \{\ wlp_D\ \}$
$\quad X(wlp_D(C0, Q)\ \wedge\ wlp_D(C1, Q))$
$=\qquad \{\ X\ \text{on}\ \wedge\ \}$
$\quad X(wlp_D(C0, Q))\ \wedge\ X(wlp_D(C1, Q))$
$\Rightarrow\qquad \{\ \text{induction hypothesis, twice}\ \}$
$\quad wlp_E(C0, X(Q))\ \wedge\ wlp_E(C1, X(Q))$
$=\qquad \{\ wlp_E\ \}$
$\quad wlp_E(C0\ \square\ C1, X(Q))$

CASE $\mathbf{var}\ s\ \mathbf{in}\ C\ \mathbf{end}$:

$\quad X(wlp_D(\mathbf{var}\ s\ \mathbf{in}\ C\ \mathbf{end}, Q))$
$=\qquad \{\ wlp_D\ \}$
$\quad X(\langle \forall s\ ::\ wlp_D(C, Q)\ \rangle)$
$=\qquad \{\ X\ \}$
$\quad \langle \forall s\ ::\ X(wlp_D(C, Q))\ \rangle$
$\Rightarrow\qquad \{\ \text{induction hypothesis}\ \}$

$$
\begin{array}{ll}
 & \langle \forall s \ :: \ wlp_E(C, X(Q)) \ \rangle \\
= & \qquad \{ \quad wlp_E \quad \} \\
 & wlp_E(\textbf{var } s \textbf{ in } C \textbf{ end}, X(Q))
\end{array}
$$

CASE **call** $m(e)$: Suppose the specification of m, after renaming its parameter to a fresh dummy t, is

method $m(t\colon T)$ **requires** p **modifies** w **ensures** q

and let zz denote the list of concrete fields and residue variables in D. Then,

$$
\begin{array}{ll}
 & X(wlp_D(\textbf{call } m(e), Q)) \\
= & \qquad \{ \quad wlp_D \quad \} \\
 & X(\langle \forall t \ :: \ t = F_D(e) \ \Rightarrow \ F_D(p) \ \wedge \ \langle \forall \ddot{zz} \ :: \ \ddot{zz} = \grave{zz} \ \Rightarrow \\
 & \qquad \langle \forall \grave{zz} \ :: \ \grave{zz} = zz \ \Rightarrow \ \langle \forall \acute{zz} \ :: \ F_D(q) \ \wedge \ mc_D(w) \ \Rightarrow \\
 & \qquad\quad \langle \forall \grave{zz} \ :: \ \grave{zz} = \ddot{zz} \ \Rightarrow \ \langle \forall zz \ :: \ zz = \acute{zz} \ \Rightarrow \ Q \ \rangle\rangle\rangle\rangle\rangle\rangle) \\
= & \qquad \{ \quad X \quad \} \\
 & \langle \forall t \ :: \ t = X(F_D(e)) \ \Rightarrow \ X(F_D(p)) \ \wedge \ \langle \forall \ddot{zz}, \ddot{\varepsilon} \ :: \ \ddot{zz} = \grave{zz} \ \wedge \ \ddot{\varepsilon} = \grave{\varepsilon} \ \Rightarrow \\
 & \quad \langle \forall \grave{zz}, \grave{\varepsilon} \ :: \ \grave{zz} = zz \ \wedge \ \grave{\varepsilon} = \varepsilon \ \Rightarrow \ \langle \forall \acute{zz}, \acute{\varepsilon} \ :: \ X(F_D(q)) \ \wedge \ X(mc_D(w)) \ \Rightarrow \\
 & \qquad \langle \forall \grave{zz}, \grave{\varepsilon} \ :: \ \grave{zz} = \ddot{zz} \ \wedge \ \grave{\varepsilon} = \ddot{\varepsilon} \ \Rightarrow \ \langle \forall zz, \varepsilon \ :: \ zz = \acute{zz} \ \wedge \ \varepsilon = \acute{\varepsilon} \ \Rightarrow \\
 & \qquad\quad X(Q) \ \rangle\rangle\rangle\rangle\rangle\rangle \\
= & \qquad \{ \quad (68){:}\ X \circ F_D = F_E \text{, three times} \quad \} \\
 & \langle \forall t \ :: \ t = F_E(e) \ \Rightarrow \ F_E(p) \ \wedge \ \langle \forall \ddot{zz}, \ddot{\varepsilon} \ :: \ \ddot{zz} = \grave{zz} \ \wedge \ \ddot{\varepsilon} = \grave{\varepsilon} \ \Rightarrow \\
 & \quad \langle \forall \grave{zz}, \grave{\varepsilon} \ :: \ \grave{zz} = zz \ \wedge \ \grave{\varepsilon} = \varepsilon \ \Rightarrow \ \langle \forall \acute{zz}, \acute{\varepsilon} \ :: \ F_E(q) \ \wedge \ X(mc_D(w)) \ \Rightarrow \\
 & \qquad \langle \forall \grave{zz}, \grave{\varepsilon} \ :: \ \grave{zz} = \ddot{zz} \ \wedge \ \grave{\varepsilon} = \ddot{\varepsilon} \ \Rightarrow \ \langle \forall zz, \varepsilon \ :: \ zz = \acute{zz} \ \wedge \ \varepsilon = \acute{\varepsilon} \ \Rightarrow \\
 & \qquad\quad X(Q) \ \rangle\rangle\rangle\rangle\rangle\rangle \\
\Rightarrow & \qquad \{ \quad \text{by (72), } [X(mc_D(w)) \ \Leftarrow \ mc_E(w)] \quad \} \\
 & \langle \forall t \ :: \ t = F_E(e) \ \Rightarrow \ F_E(p) \ \wedge \ \langle \forall \ddot{zz}, \ddot{\varepsilon} \ :: \ \ddot{zz} = \grave{zz} \ \wedge \ \ddot{\varepsilon} = \grave{\varepsilon} \ \Rightarrow \\
 & \quad \langle \forall \grave{zz}, \grave{\varepsilon} \ :: \ \grave{zz} = zz \ \wedge \ \grave{\varepsilon} = \varepsilon \ \Rightarrow \ \langle \forall \acute{zz}, \acute{\varepsilon} \ :: \ F_E(q) \ \wedge \ mc_E(w) \ \Rightarrow \\
 & \qquad \langle \forall \grave{zz}, \grave{\varepsilon} \ :: \ \grave{zz} = \ddot{zz} \ \wedge \ \grave{\varepsilon} = \ddot{\varepsilon} \ \Rightarrow \ \langle \forall zz, \varepsilon \ :: \ zz = \acute{zz} \ \wedge \ \varepsilon = \acute{\varepsilon} \ \Rightarrow \\
 & \qquad\quad X(Q) \ \rangle\rangle\rangle\rangle\rangle\rangle \\
= & \qquad \{ \quad wlp_E \text{, since } zz, \varepsilon \text{ are the concrete fields and residue variables} \\
 & \qquad\quad \text{in } E \quad \} \\
 & wlp_E(\textbf{call } m(e), X(Q))
\end{array}
$$

■

A32 X and VC

We now have only one property left to prove to complete the proof of Soundness Lemma C and thus the Soundness Theorem, namely X property (75): for any

D, ε, and E such that $Extend(D, \varepsilon, E)$ and any implementation C in D of a method m,

$$[VC_D(m, C)] \;\Rightarrow\; [X(VC_D(m, C))]$$

Proof of (75). The proof consists of a number of steps that transform the syntactic expression Q into the syntactic expression $X(Q)$, preserving the validity of the original Q. This technique is the same as we used in Section A16, where we proved Soundness Lemma D from Soundness Lemma C, but the steps here will be different.

For use in the proof, we define three functions, " $(\,,\,)$ ", car, and cdr, which satisfy the following properties:

$$\langle \forall a, b, c, d :: (a, b) = (c, d) \;\equiv\; a = c \;\wedge\; b = d \,\rangle \tag{79}$$

$$\langle \forall p, a, b :: p = (a, b) \;\equiv\; a = car(p) \;\wedge\; b = cdr(p) \,\rangle \tag{80}$$

$$\langle \forall a, b, t :: (a, b)[t] \;\equiv\; (a[t], b[t]) \,\rangle \tag{81}$$

(Readers who doubt that such properties can be postulated in this soundness proof are encouraged to consult Part IV of this appendix.)

From the definition of X (Figure 17, page 134), we can see the syntactic differences between $VC_D(m, C)$ and $X(VC_D(m, C))$. These differences are shown side by side in Figure 18. In that figure, a is an abstract variable in D that depends on ε in E.

The idea is to transform the formula $VC_D(m, C)$ into $X(VC_D(m, C))$, preserving validity. We do so by a series of transformations to *the working formula*, which starts off as $VC_D(m, C)$ and ends up as $X(VC_D(m, C))$. Our series of transformations are guided by the differences shown in Figure 18.

So, initially, our working formula contains subexpressions of the forms suggested by Figure 19. In that figure, a takes the rôle of a typical abstract variable that depends on ε in E, and b takes the rôle of a typical abstract variable that does not depend on ε.

Step 0: From Figure 19 to Figure 20. We start by reconciling an easy difference: D1. For each formula of the form (shown in the left column in Figure 18 under the label) D1 in the working formula, we insert a surrounding quantification

$VC_D(m, C)$	$X(VC_D(m, C))$
D0. Functionalized forms:	
$\mathcal{F}.a(s\tilde{r}es, \ldots)$	$\mathcal{F}.a(s\tilde{r}es, \ldots, \tilde{\varepsilon})$
D1. Quantifications over shared residues:	
$\langle \forall\, s\tilde{r}es :: \ \ldots \rangle$	$\langle \forall\, \tilde{\varepsilon} :: \langle \forall\, s\tilde{r}es :: \ \ldots \rangle\rangle$
D2. Quantifications over shared residues with equality antecedents:	
$\langle \forall\, s\tilde{r}es :: \ s\tilde{r}es = s\bar{r}es \ \Rightarrow \ \ldots \rangle$	$\langle \forall\, \tilde{\varepsilon} :: \ \tilde{\varepsilon} = \bar{\varepsilon} \ \Rightarrow$ $\langle \forall\, s\tilde{r}es :: \ s\tilde{r}es = s\bar{r}es \ \Rightarrow \ \ldots \rangle\rangle$
D3. Shared-residue modification constraints:	
$w{:}\langle \forall\, s :: \ s\grave{r}es[s] = s\acute{r}es[s]$ $\vee \ \ldots \rangle$	$w{:}\langle \forall\, s :: \ s\grave{r}es[s] = s\acute{r}es[s]$ $\vee \ \ldots \rangle \ \wedge \ modcon_E(w, \varepsilon, sres)$
D4. Pointwise axioms:	
$\langle \forall\, t, s\grave{r}es, s\acute{r}es, \ldots ::$ $s\grave{r}es[t] = s\acute{r}es[t] \ \wedge \ \ldots \ \Rightarrow$ $\mathcal{F}.a(s\grave{r}es, \ldots)[t] =$ $\mathcal{F}.a(s\acute{r}es, \ldots)[t] \rangle$	$\langle \forall\, t, s\grave{r}es, s\acute{r}es, \ldots, \grave{\varepsilon}, \acute{\varepsilon} ::$ $s\grave{r}es[t] = s\acute{r}es[t] \ \wedge \ \ldots \ \wedge \ \grave{\varepsilon}[t] = \acute{\varepsilon}[t] \ \Rightarrow$ $\mathcal{F}.a(s\grave{r}es, \ldots, \grave{\varepsilon})[t] =$ $\mathcal{F}.a(s\acute{r}es, \ldots, \acute{\varepsilon})[t] \rangle$
D5. Rep axioms:	
$\langle \forall\, t{:}\, T, sres, \ldots :: \ t \neq \mathbf{nil} \ \Rightarrow$ $\mathcal{F}.a(sres, \ldots)[t] = e \rangle$	$\langle \forall\, t{:}\, T, sres, \ldots, \varepsilon :: \ t \neq \mathbf{nil} \ \Rightarrow$ $\mathcal{F}.a(sres, \ldots, \varepsilon)[t] = e \rangle$

Figure 18: A diagram showing the syntactic differences between $VC_D(m, C)$ and $X(VC_D(m, C))$.

$$
\begin{array}{ll}
\text{D0:} & \mathcal{F}.a(s\tilde{r}es, \ldots) \\
 & \mathcal{F}.b(s\tilde{r}es, \ldots) \\
\text{D1:} & \langle \forall\, s\tilde{r}es :: \ \ldots \rangle \\
\text{D2:} & \langle \forall\, s\tilde{r}es :: \ s\tilde{r}es = s\bar{r}es \ \Rightarrow \ \ldots \rangle \\
\text{D3:} & w{:}\langle \forall\, s :: \ s\grave{r}es[s] = s\acute{r}es[s] \ \vee \ \ldots \rangle \\
\text{D4:} & \langle \forall\, t, s\grave{r}es, s\acute{r}es, \ldots :: \ s\grave{r}es[t] = s\acute{r}es[t] \ \wedge \ \ldots \ \Rightarrow \\
 & \qquad \mathcal{F}.a(s\grave{r}es, \ldots)[t] = \mathcal{F}.a(s\acute{r}es, \ldots)[t] \rangle \\
 & \langle \forall\, t, s\grave{r}es, s\acute{r}es, \ldots :: \ s\grave{r}es[t] = s\acute{r}es[t] \ \wedge \ \ldots \ \Rightarrow \\
 & \qquad \mathcal{F}.b(s\grave{r}es, \ldots)[t] = \mathcal{F}.b(s\acute{r}es, \ldots)[t] \rangle \\
\text{D5:} & \langle \forall\, t{:}\, T, sres, \ldots :: \ t \neq \mathbf{nil} \ \Rightarrow \ \mathcal{F}.a(sres, \ldots)[t] = e \rangle \\
 & \langle \forall\, t{:}\, T, sres, \ldots :: \ t \neq \mathbf{nil} \ \Rightarrow \ \mathcal{F}.b(sres, \ldots)[t] = e \rangle
\end{array}
$$

Figure 19: A sketch of the initial working formula.

$$
\begin{array}{ll}
\text{D0:} & \mathcal{F}.a(\tilde{sres}, \ldots) \\
 & \mathcal{F}.b(\tilde{sres}, \ldots) \\
\text{D1:} & \langle \forall \tilde{\varepsilon} :: \langle \forall \tilde{sres} :: \ \ldots \rangle\rangle \\
\text{D2:} & \langle \forall \tilde{sres} :: \tilde{sres} = \bar{sres} \ \Rightarrow \ \ldots \rangle \\
\text{D3:} & w{:}\langle \forall s :: \grave{sres}[s] = \acute{sres}[s] \ \vee \ \ldots \rangle \\
\text{D4:} & \langle \forall t, \grave{sres}, \acute{sres}, \ldots :: \grave{sres}[t] = \acute{sres}[t] \ \wedge \ \ldots \ \Rightarrow \\
 & \qquad \mathcal{F}.a(\grave{sres}, \ldots)[t] = \mathcal{F}.a(\acute{sres}, \ldots)[t] \rangle \\
 & \langle \forall t, \grave{sres}, \acute{sres}, \ldots :: \grave{sres}[t] = \acute{sres}[t] \ \wedge \ \ldots \ \Rightarrow \\
 & \qquad \mathcal{F}.b(\grave{sres}, \ldots)[t] = \mathcal{F}.b(\acute{sres}, \ldots)[t] \rangle \\
\text{D5:} & \langle \forall t{:}T, sres, \ldots :: t \neq \mathbf{nil} \ \Rightarrow \ \mathcal{F}.a(sres, \ldots)[t] = e \rangle \\
 & \langle \forall t{:}T, sres, \ldots :: t \neq \mathbf{nil} \ \Rightarrow \ \mathcal{F}.b(sres, \ldots)[t] = e \rangle
\end{array}
$$

Figure 20: A sketch of the working formula after transforming it to reconcile the D1 differences.

over ε. The fact that this transformation preserves the validity of the working formula is justified by the following fact: if R is a predicate with no free occurrences of $\tilde{\varepsilon}$, then

$$[R \equiv \langle \forall \tilde{\varepsilon} :: R \rangle]$$

In Figure 19, the working formula has no free occurrences of $\tilde{\varepsilon}$ and inserting quantifications over $\tilde{\varepsilon}$ as suggested does not introduce any free occurrences of $\tilde{\varepsilon}$, so this transformation can be done one place at a time, in any order. After the transformation, the working formula has the shape suggested by Figure 20.

Step 1: From Figure 20 to Figure 21. Differences of the form described by D2 between the formulas $VC_D(m, C)$ and $X(VC_D(m, C))$ are also easy to reconcile in the working formula. First, we insert quantifications over $\tilde{\varepsilon}$ as needed, like we did in the previous transformation. Then, we insert the antecedent "$\tilde{\varepsilon} = \bar{\varepsilon} \Rightarrow$" in these places. Since the subexpressions of the form D2 appear only in positive positions (by characteristic V1, page 87), inserting these antecedents will only weaken the working formula, hence preserving its validity. After this transformation, the working formula has the shape suggested by Figure 21.

Step 2: From Figure 21 to Figure 22. To transform the working formula further, we need to tackle the problem that the arity of abstraction function $\mathcal{F}.a$ is

$$
\begin{array}{ll}
\text{D0:} & \mathcal{F}.a(\widetilde{sres}, \ldots) \\
& \mathcal{F}.b(\widetilde{sres}, \ldots) \\
\text{D1:} & \langle \forall \tilde{\varepsilon} :: \langle \forall \widetilde{sres} :: \ \ldots \rangle\rangle \\
\text{D2:} & \langle \forall \tilde{\varepsilon} :: \tilde{\varepsilon} = \bar{\varepsilon} \ \Rightarrow \ \langle \forall \widetilde{sres} :: \widetilde{sres} = \bar{sres} \ \Rightarrow \ \ldots \rangle\rangle \\
\text{D3:} & w{:}\langle \forall s :: \grave{sres}[s] = \acute{sres}[s] \ \vee \ \ldots \rangle \\
\text{D4:} & \langle \forall t, \grave{sres}, \acute{sres}, \ldots :: \grave{sres}[t] = \acute{sres}[t] \ \wedge \ \ldots \ \Rightarrow \\
& \qquad \mathcal{F}.a(\grave{sres}, \ldots)[t] = \mathcal{F}.a(\acute{sres}, \ldots)[t] \rangle \\
& \langle \forall t, \grave{sres}, \acute{sres}, \ldots :: \grave{sres}[t] = \acute{sres}[t] \ \wedge \ \ldots \ \Rightarrow \\
& \qquad \mathcal{F}.b(\grave{sres}, \ldots)[t] = \mathcal{F}.b(\acute{sres}, \ldots)[t] \rangle \\
\text{D5:} & \langle \forall t{:}\, T, sres, \ldots :: t \neq \textbf{nil} \ \Rightarrow \ \mathcal{F}.a(sres, \ldots)[t] = e \rangle \\
& \langle \forall t{:}\, T, sres, \ldots :: t \neq \textbf{nil} \ \Rightarrow \ \mathcal{F}.b(sres, \ldots)[t] = e \rangle
\end{array}
$$

Figure 21: A sketch of the working formula after transforming it also to reconcile the D2 differences.

different in D than in E. In particular, the arity of $\mathcal{F}.a$ in the current working formula is one less than the arity of $\mathcal{F}.a$ in $X(VC_D(m, C))$. To reconcile this difference, we will substitute a new function for $\mathcal{F}.a$, similar to what we did in Section A16, but here we must first worry about getting the appropriate extra argument, namely various bindings of $\tilde{\varepsilon}$, in place. To that end, we start by performing the transformation justified by the following calculation:

$$
\begin{array}{ll}
& \langle \forall \widetilde{sres} :: R \rangle \\
\Rightarrow & \qquad \{ \text{ instantiate } \widetilde{sres} \ \} \\
& \langle \forall \widetilde{sres} :: R(\widetilde{sres} := (\widetilde{sres}, \tilde{\varepsilon})) \rangle
\end{array}
$$

We perform this transformation at each place in the working formula where there's a quantification over $\widetilde{sres}$, except in pointwise axioms and rep axioms. The transformation preserves the validity of the working formula, since the affected quantifications occur in positive positions (by characteristic V1). The transformation yields a working formula whose shape is suggested by Figure 22. Note that this transformation does not change occurrences of $sres$ bound to the quantifications in pointwise axioms and rep axioms. Note also that the transformation does alter occurrences of $sres$ that appear in equality antecedents and in residue modification constraints, as shown in the figure.

Step 3: From Figure 22 to Figure 23. Let's clean up the equality antecedents right away. We apply the following calculation to all $\widetilde{sres}$ quantifications with

$$
\begin{array}{ll}
\text{D0:} & \mathcal{F}.a((s\tilde{r}es, \tilde{\varepsilon}), \ldots) \\
& \mathcal{F}.b((s\tilde{r}es, \tilde{\varepsilon}), \ldots) \\
\text{D1:} & \langle \forall \tilde{\varepsilon} :: \langle \forall s\tilde{r}es :: \ \ldots \rangle\rangle \\
\text{D2:} & \langle \forall \tilde{\varepsilon} :: \tilde{\varepsilon} = \bar{\varepsilon} \ \Rightarrow \ \langle \forall s\tilde{r}es :: (s\tilde{r}es, \tilde{\varepsilon}) = (s\bar{r}es, \bar{\varepsilon}) \ \Rightarrow \ \ldots \rangle\rangle \\
\text{D3:} & w{:}\langle \forall s :: (s\grave{r}es, \grave{\varepsilon})[s] = (s\acute{r}es, \acute{\varepsilon})[s] \ \vee \ \ldots \rangle \\
\text{D4:} & \langle \forall t, s\grave{r}es, s\acute{r}es, \ldots :: s\grave{r}es[t] = s\acute{r}es[t] \ \wedge \ \ldots \ \Rightarrow \\
& \qquad \mathcal{F}.a(s\grave{r}es, \ldots)[t] = \mathcal{F}.a(s\acute{r}es, \ldots)[t] \rangle \\
& \langle \forall t, s\grave{r}es, s\acute{r}es, \ldots :: s\grave{r}es[t] = s\acute{r}es[t] \ \wedge \ \ldots \ \Rightarrow \\
& \qquad \mathcal{F}.b(s\grave{r}es, \ldots)[t] = \mathcal{F}.b(s\acute{r}es, \ldots)[t] \rangle \\
\text{D5:} & \langle \forall t{:}\, T, sres, \ldots :: t \neq \mathbf{nil} \ \Rightarrow \ \mathcal{F}.a(sres, \ldots)[t] = e \rangle \\
& \langle \forall t{:}\, T, sres, \ldots :: t \neq \mathbf{nil} \ \Rightarrow \ \mathcal{F}.b(sres, \ldots)[t] = e \rangle
\end{array}
$$

Figure 22: A sketch of the working formula after substituting the pair $(s\tilde{r}es, \tilde{\varepsilon})$ for occurrences of $s\tilde{r}es$ in the formula sketched in Figure 21. This transformation affects D0, D2, and D3.

equality antecedents in the working formula:

$$
\begin{array}{ll}
& \langle \forall \tilde{\varepsilon} :: \tilde{\varepsilon} = \bar{\varepsilon} \ \Rightarrow \ \langle \forall s\tilde{r}es :: (s\tilde{r}es, \tilde{\varepsilon}) = (s\bar{r}es, \bar{\varepsilon}) \ \Rightarrow \ Q \rangle\rangle \\
= & \qquad \{ \ \text{(79): pairing and equality} \ \} \\
& \langle \forall \tilde{\varepsilon} :: \tilde{\varepsilon} = \bar{\varepsilon} \ \Rightarrow \ \langle \forall s\tilde{r}es :: s\tilde{r}es = s\bar{r}es \ \wedge \ \tilde{\varepsilon} = \bar{\varepsilon} \ \Rightarrow \ Q \rangle\rangle \\
= & \qquad \{ \ \text{predicate calculus} \ \} \\
& \langle \forall \tilde{\varepsilon} :: \tilde{\varepsilon} = \bar{\varepsilon} \ \Rightarrow \ \langle \forall s\tilde{r}es :: s\tilde{r}es = s\bar{r}es \ \Rightarrow \ Q \rangle\rangle
\end{array}
$$

Now we are ready to replace abstraction functions with new ones whose arity is one larger. Let Q denote the current working formula. For each abstract variable a that depends on ε in E (and as usual, we show the case where a has exactly one dependency, f, in D), we perform the following steps:

$$
\begin{array}{ll}
& [Q] \\
\Rightarrow & \qquad \{ \ \text{instantiate the universally quantified } \mathcal{F}.a \ \} \\
& [Q(\mathcal{F}.a := \langle \lambda\, p, res.a, f :: \mathcal{F}.a(car(p), res.a, f, cdr(p)) \rangle)]
\end{array}
$$

And for each abstract variable b that does not depend on ε in E (and as usual, we show the case where b has exactly one dependency, g, in D), we perform the following steps:

$$
\begin{array}{ll}
& [Q] \\
\Rightarrow & \qquad \{ \ \text{instantiate the universally quantified } \mathcal{F}.b \ \} \\
& [Q(\mathcal{F}.b := \langle \lambda\, p, res.b, g :: \mathcal{F}.b(car(p), res.b, g) \rangle)]
\end{array}
$$

$$
\begin{array}{ll}
\text{D0:} & \mathcal{F}.a(s\tilde{r}es, \ldots, \tilde{\varepsilon}) \\
 & \mathcal{F}.b(s\tilde{r}es, \ldots) \\
\text{D1:} & \langle \forall\, \tilde{\varepsilon} \;::\; \langle \forall\, s\tilde{r}es \;::\; \ldots \rangle\rangle \\
\text{D2:} & \langle \forall\, \tilde{\varepsilon} \;::\; \tilde{\varepsilon} = \bar{\varepsilon} \;\Rightarrow\; \langle \forall\, s\tilde{r}es \;::\; s\tilde{r}es = s\bar{r}es \;\Rightarrow\; \ldots \rangle\rangle \\
\text{D3:} & w{:}\langle \forall\, s \;::\; (s\grave{r}es, \grave{\varepsilon})[s] = (s\acute{r}es, \acute{\varepsilon})[s] \;\vee\; \ldots \rangle \\
\text{D4:} & \langle \forall\, t, s\grave{r}es, s\acute{r}es, \ldots \;::\; s\grave{r}es[t] = s\acute{r}es[t] \;\wedge\; \ldots \;\Rightarrow \\
 & \qquad \mathcal{F}.a(car(s\grave{r}es), \ldots, cdr(s\grave{r}es))[t] = \mathcal{F}.a(car(s\acute{r}es), \ldots, cdr(s\acute{r}es))[t] \;\rangle \\
 & \langle \forall\, t, s\grave{r}es, s\acute{r}es, \ldots \;::\; s\grave{r}es[t] = s\acute{r}es[t] \;\wedge\; \ldots \;\Rightarrow \\
 & \qquad \mathcal{F}.b(car(s\grave{r}es), \ldots)[t] = \mathcal{F}.b(car(s\acute{r}es), \ldots)[t] \;\rangle \\
\text{D5:} & \langle \forall\, t{:}\, T, sres, \ldots \;::\; t \neq \mathbf{nil} \;\Rightarrow\; \mathcal{F}.a(car(sres), \ldots, cdr(sres))[t] = e \;\rangle \\
 & \langle \forall\, t{:}\, T, sres, \ldots \;::\; t \neq \mathbf{nil} \;\Rightarrow\; \mathcal{F}.b(car(sres), \ldots)[t] = e \;\rangle
\end{array}
$$

Figure 23: A sketch of the working formula after cleaning up equality antecedents and replacing abstraction functions with ones of larger arity, which affects D0, D2, D4, and D5.

Note that since b does not depend on ε, the new $\mathcal{F}.b$ has the same arity as the old $\mathcal{F}.b$, but it throws away half of the first argument given to the old $\mathcal{F}.b$. After applying β-conversion to all of these λ-expressions and applying the car and cdr of pairs according to (80), the working formula will have the form sketched in Figure 23. Note that the substitutions we made affect all occurrences of abstraction functions, even those in pointwise axioms and rep axioms.

Step 4: From Figure 23 to Figure 24. At this point, the first four lines of the sketch of the working formula in Figure 23 look like we want them to, but the other lines of the sketch do not. We choose to address rep axioms next.

For any abstract variable a that depends on ε in E, we calculate,

$$
\begin{array}{ll}
 & \langle \forall\, t{:}\, T, sres, res.a, f \;::\; t \neq \mathbf{nil} \;\Rightarrow \\
 & \qquad \mathcal{F}.a(car(sres), res.a, f, cdr(sres))[t] = e \;\rangle \\
= & \qquad \{ \;\text{rename dummy } sres \text{ to a new name } p\text{; note that } e \text{ does not} \\
 & \qquad\quad \text{contain any occurrences of } sres\text{, since } e \text{ is a user expression} \;\} \\
 & \langle \forall\, t{:}\, T, p, res.a, f \;::\; t \neq \mathbf{nil} \;\Rightarrow\; \mathcal{F}.a(car(p), res.a, f, cdr(p))[t] = e \;\rangle \\
= & \qquad \{ \;\text{one-point rule to introduce } sres \text{ and } \varepsilon \text{ (neither of} \\
 & \qquad\quad \text{which occurs free in the body of the rep axiom) for the} \\
 & \qquad\quad \text{expressions } car(p) \text{ and } cdr(p) \;\}
\end{array}
$$

$$\langle \forall t{:}\, T, p, res.a, f, sres, \varepsilon \;::\; t \neq \textbf{nil} \;\wedge\; sres = car(p) \;\wedge\; \varepsilon = cdr(p) \;\Rightarrow\; \mathcal{F}.a(sres, res.a, f, \varepsilon)[t] = e \,\rangle$$

$= \quad \{$ (80): pairing, car, and cdr $\}$

$$\langle \forall t{:}\, T, p, res.a, f, sres, \varepsilon \;::\; t \neq \textbf{nil} \;\wedge\; p = (sres, \varepsilon) \;\Rightarrow\; \mathcal{F}.a(sres, res.a, f, \varepsilon)[t] = e \,\rangle$$

$= \quad \{$ one-point rule to eliminate p $\}$

$$\langle \forall t{:}\, T, sres, res.a, f, \varepsilon \;::\; t \neq \textbf{nil} \;\Rightarrow\; \mathcal{F}.a(sres, res.a, f, \varepsilon)[t] = e \,\rangle$$

This last line has the form of the corresponding rep axiom for a in $X(VC_D(m, C))$. For any abstract variable b that does not depend on ε in E, we calculate,

$$\langle \forall t{:}\, T, sres, res.b, g \;::\; t \neq \textbf{nil} \;\Rightarrow\; \mathcal{F}.b(car(sres), res.b, g)[t] = e \,\rangle$$

$= \quad \{$ rename dummy $sres$ to a new name p; note that e does not contain any occurrences of $sres$, since e is a user expression $\}$

$$\langle \forall t{:}\, T, p, res.b, g \;::\; t \neq \textbf{nil} \;\Rightarrow\; \mathcal{F}.b(car(p), res.b, g)[t] = e \,\rangle$$

$= \quad \{$ one-point rule to introduce $sres$ and ε (neither of which occurs free in the body of the rep axiom) for the expressions $car(p)$ and $cdr(p)$ $\}$

$$\langle \forall t{:}\, T, p, res.b, g, sres, \varepsilon \;::\; t \neq \textbf{nil} \;\wedge\; sres = car(p) \;\wedge\; \varepsilon = cdr(p) \;\Rightarrow\; \mathcal{F}.b(sres, res.b, g)[t] = e \,\rangle$$

$= \quad \{$ (80): pairing, car, and cdr $\}$

$$\langle \forall t{:}\, T, p, res.b, g, sres, \varepsilon \;::\; t \neq \textbf{nil} \;\wedge\; p = (sres, \varepsilon) \;\Rightarrow\; \mathcal{F}.b(sres, res.b, g)[t] = e \,\rangle$$

$= \quad \{$ one-point rule to eliminate p $\}$

$$\langle \forall t{:}\, T, sres, res.b, g, \varepsilon \;::\; t \neq \textbf{nil} \;\Rightarrow\; \mathcal{F}.b(sres, res.b, g)[t] = e \,\rangle$$

$= \quad \{$ ε does not occur free in the body of the quantification $\}$

$$\langle \forall t{:}\, T, sres, res.b, g \;::\; t \neq \textbf{nil} \;\Rightarrow\; \mathcal{F}.b(sres, res.b, g)[t] = e \,\rangle$$

This last line has the form of the corresponding rep axiom for b in $X(VC_D(m, C))$. Applying these transformations to every rep axiom in the working formula, we end up with a working formula whose shape is suggested by Figure 24.

Step 5: From Figure 24 to Figure 25. For any abstract variable a that depends on ε in E, we calculate,

$$\langle \forall t, \grave{sres}, \acute{sres}, \grave{res}.a, \acute{res}.a, \grave{f}, \acute{f} \;::\; \grave{sres}[t] = \acute{sres}[t] \;\wedge\; \grave{res}.a[t] = \acute{res}.a[t] \;\wedge\; \grave{f}[t] = \acute{f}[t] \;\Rightarrow\; \mathcal{F}.a(car(\grave{sres}), \grave{res}.a, \grave{f}, cdr(\grave{sres}))[t] = \mathcal{F}.a(car(\acute{sres}), \acute{res}.a, \acute{f}, cdr(\acute{sres}))[t] \,\rangle$$

$= \quad \{$ rename dummies $\grave{sres}$, $\acute{sres}$ to new names $\grave{p}$, $\acute{p}$ $\}$

$$
\begin{array}{ll}
\text{D0:} & \mathcal{F}.a(\tilde{sres}, \ldots, \tilde{\varepsilon}) \\
 & \mathcal{F}.b(\tilde{sres}, \ldots) \\
\text{D1:} & \langle \forall\, \tilde{\varepsilon} \;::\; \langle \forall\, \tilde{sres} \;::\; \ldots \rangle\rangle \\
\text{D2:} & \langle \forall\, \tilde{\varepsilon} \;::\; \tilde{\varepsilon} = \bar{\varepsilon} \;\Rightarrow\; \langle \forall\, \tilde{sres} \;::\; \tilde{sres} = \bar{sres} \;\Rightarrow\; \ldots \rangle\rangle \\
\text{D3:} & w{:}\langle \forall\, s \;::\; (\grave{sres}, \grave{\varepsilon})[s] = (\acute{sres}, \acute{\varepsilon})[s] \;\vee\; \ldots \rangle \\
\text{D4:} & \langle \forall\, t, \grave{sres}, \acute{sres}, \ldots \;::\; \grave{sres}[t] = \acute{sres}[t] \;\wedge\; \ldots \;\Rightarrow \\
 & \qquad \mathcal{F}.a(car(\grave{sres}), \ldots, cdr(\grave{sres}))[t] = \mathcal{F}.a(car(\acute{sres}), \ldots, cdr(\acute{sres}))[t] \rangle \\
 & \langle \forall\, t, \grave{sres}, \acute{sres}, \ldots \;::\; \grave{sres}[t] = \acute{sres}[t] \;\wedge\; \ldots \;\Rightarrow \\
 & \qquad \mathcal{F}.b(car(\grave{sres}), \ldots)[t] = \mathcal{F}.b(car(\acute{sres}), \ldots)[t] \rangle \\
\text{D5:} & \langle \forall\, t{:}\, T, sres, \ldots, \varepsilon \;::\; t \neq \mathbf{nil} \;\Rightarrow\; \mathcal{F}.a(sres, \ldots, \varepsilon)[t] = e \rangle \\
 & \langle \forall\, t{:}\, T, sres, \ldots \;::\; t \neq \mathbf{nil} \;\Rightarrow\; \mathcal{F}.b(sres, \ldots)[t] = e \rangle \ldots
\end{array}
$$

Figure 24: A sketch of the working formula after massaging the rep axioms in Figure 23, which affects D5.

$$
\begin{array}{ll}
 & \langle \forall\, t, \grave{p}, \acute{p}, \grave{res}.a, \acute{res}.a, \grave{f}, \acute{f} \;:: \\
 & \qquad \grave{p}[t] = \acute{p}[t] \;\wedge\; \grave{res}.a[t] = \acute{res}.a[t] \;\wedge\; \grave{f}[t] = \acute{f}[t] \;\Rightarrow \\
 & \qquad\quad \mathcal{F}.a(car(\grave{p}), \grave{res}.a, \grave{f}, cdr(\grave{p}))[t] = \mathcal{F}.a(car(\acute{p}), \acute{res}.a, \acute{f}, cdr(\acute{p}))[t] \rangle \\
= & \qquad \{ \;\; \text{one-point rule to introduce } \grave{sres},\ \acute{sres},\ \grave{\varepsilon}, \text{ and } \acute{\varepsilon} \text{ as names for the} \\
 & \qquad\quad car \text{ and } cdr \text{ expressions} \;\; \} \\
 & \langle \forall\, t, \grave{p}, \acute{p}, \grave{res}.a, \acute{res}.a, \grave{f}, \acute{f}, \grave{sres}, \acute{sres}, \grave{\varepsilon}, \acute{\varepsilon} \;:: \\
 & \qquad \grave{sres} = car(\grave{p}) \;\wedge\; \grave{\varepsilon} = cdr(\grave{p}) \;\wedge\; \acute{sres} = car(\acute{p}) \;\wedge\; \acute{\varepsilon} = cdr(\acute{p}) \;\wedge \\
 & \qquad \grave{p}[t] = \acute{p}[t] \;\wedge\; \grave{res}.a[t] = \acute{res}.a[t] \;\wedge\; \grave{f}[t] = \acute{f}[t] \;\Rightarrow \\
 & \qquad\quad \mathcal{F}.a(\grave{sres}, \grave{res}.a, \grave{f}, \grave{\varepsilon})[t] = \mathcal{F}.a(\acute{sres}, \acute{res}.a, \acute{f}, \acute{\varepsilon})[t] \rangle \\
= & \qquad \{ \;\; \text{(80): pairing, } car\text{, and } cdr \;\; \} \\
 & \langle \forall\, t, \grave{p}, \acute{p}, \grave{res}.a, \acute{res}.a, \grave{f}, \acute{f}, \grave{sres}, \acute{sres}, \grave{\varepsilon}, \acute{\varepsilon} \;:: \\
 & \qquad \grave{p} = (\grave{sres}, \grave{\varepsilon}) \;\wedge\; \acute{p} = (\acute{sres}, \acute{\varepsilon}) \;\wedge \\
 & \qquad \grave{p}[t] = \acute{p}[t] \;\wedge\; \grave{res}.a[t] = \acute{res}.a[t] \;\wedge\; \grave{f}[t] = \acute{f}[t] \;\Rightarrow \\
 & \qquad\quad \mathcal{F}.a(\grave{sres}, \grave{res}.a, \grave{f}, \grave{\varepsilon})[t] = \mathcal{F}.a(\acute{sres}, \acute{res}.a, \acute{f}, \acute{\varepsilon})[t] \rangle \\
= & \qquad \{ \;\; \text{one-point rule to eliminate } \grave{p} \text{ and } \acute{p} \;\; \} \\
 & \langle \forall\, t, \grave{res}.a, \acute{res}.a, \grave{f}, \acute{f}, \grave{sres}, \acute{sres}, \grave{\varepsilon}, \acute{\varepsilon} \;:: \\
 & \qquad (\grave{sres}, \grave{\varepsilon})[t] = (\acute{sres}, \acute{\varepsilon})[t] \;\wedge\; \grave{res}.a[t] = \acute{res}.a[t] \;\wedge\; \grave{f}[t] = \acute{f}[t] \;\Rightarrow \\
 & \qquad\quad \mathcal{F}.a(\grave{sres}, \grave{res}.a, \grave{f}, \grave{\varepsilon})[t] = \mathcal{F}.a(\acute{sres}, \acute{res}.a, \acute{f}, \acute{\varepsilon})[t] \rangle \\
= & \qquad \{ \;\; \text{(81): pairing and select, and (79): pairing and equality} \;\; \}
\end{array}
$$

$$
\begin{array}{l}
\langle \forall\, t, \grave{sres}, \acute{sres}, \grave{res.a}, \acute{res.a}, \grave{f}, \acute{f}, \grave{\varepsilon}, \acute{\varepsilon} \;::\\
\quad \grave{sres}[t] = \acute{sres}[t] \;\wedge\; \grave{res.a}[t] = \acute{res.a}[t] \;\wedge\; \grave{f}[t] = \acute{f}[t] \;\wedge\; \grave{\varepsilon}[t] = \acute{\varepsilon}[t] \;\Rightarrow\\
\qquad \mathcal{F}.a(\grave{sres}, \grave{res.a}, \grave{f}, \grave{\varepsilon})[t] = \mathcal{F}.a(\acute{sres}, \acute{res.a}, \acute{f}, \acute{\varepsilon})[t] \;\rangle
\end{array}
$$

This last line has the form of the pointwise axiom for $\mathcal{F}.a$ in $X(VC_D(m, C))$. For any abstract variable b that does not depend on ε in E, we calculate,

$$
\begin{array}{ll}
 & \langle \forall\, t, \grave{sres}, \acute{sres}, \grave{res.b}, \acute{res.b}, \grave{g}, \acute{g} \;::\\
 & \quad \grave{sres}[t] = \acute{sres}[t] \;\wedge\; \grave{res.b}[t] = \acute{res.b}[t] \;\wedge\; \grave{g}[t] = \acute{g}[t] \;\Rightarrow\\
 & \qquad \mathcal{F}.b(car(\grave{sres}), \grave{res.b}, \grave{g})[t] = \mathcal{F}.b(car(\acute{sres}), \acute{res.b}, \acute{g})[t] \;\rangle\\
= & \quad \{ \text{ rename dummies } \grave{sres}\text{, } \acute{sres} \text{ to new names } \grave{p}\text{, } \acute{p} \;\}\\
 & \langle \forall\, t, \grave{p}, \acute{p}, \grave{res.b}, \acute{res.b}, \grave{g}, \acute{g} \;::\\
 & \quad \grave{p}[t] = \acute{p}[t] \;\wedge\; \grave{res.b}[t] = \acute{res.b}[t] \;\wedge\; \grave{g}[t] = \acute{g}[t] \;\Rightarrow\\
 & \qquad \mathcal{F}.b(car(\grave{p}), \grave{res.b}, \grave{g})[t] = \mathcal{F}.b(car(\acute{p}), \acute{res.b}, \acute{g})[t] \;\rangle\\
= & \quad \{ \text{ one-point rule to introduce } \grave{sres}\text{, } \acute{sres}\text{, } \grave{\varepsilon}\text{, and } \acute{\varepsilon} \text{ as names for}\\
 & \qquad car \text{ and } cdr \text{ expressions } \}\\
 & \langle \forall\, t, \grave{p}, \acute{p}, \grave{res.b}, \acute{res.b}, \grave{g}, \acute{g}, \grave{sres}, \acute{sres}, \grave{\varepsilon}, \acute{\varepsilon} \;::\\
 & \quad \grave{sres} = car(\grave{p}) \;\wedge\; \grave{\varepsilon} = cdr(\grave{p}) \;\wedge\; \acute{sres} = car(\acute{p}) \;\wedge\; \acute{\varepsilon} = cdr(\acute{p}) \;\wedge\\
 & \quad \grave{p}[t] = \acute{p}[t] \;\wedge\; \grave{res.b}[t] = \acute{res.b}[t] \;\wedge\; \grave{g}[t] = \acute{g}[t] \;\Rightarrow\\
 & \qquad \mathcal{F}.b(\grave{sres}, \grave{res.b}, \grave{g})[t] = \mathcal{F}.b(\acute{sres}, \acute{res.b}, \acute{g})[t] \;\rangle\\
= & \quad \{ \text{ (80): pairing, } car\text{, and } cdr \;\}\\
 & \langle \forall\, t, \grave{p}, \acute{p}, \grave{res.b}, \acute{res.b}, \grave{g}, \acute{g}, \grave{sres}, \acute{sres}, \grave{\varepsilon}, \acute{\varepsilon} \;::\\
 & \quad \grave{p} = (\grave{sres}, \grave{\varepsilon}) \;\wedge\; \acute{p} = (\acute{sres}, \acute{\varepsilon}) \;\wedge\\
 & \quad \grave{p}[t] = \acute{p}[t] \;\wedge\; \grave{res.b}[t] = \acute{res.b}[t] \;\wedge\; \grave{g}[t] = \acute{g}[t] \;\Rightarrow\\
 & \qquad \mathcal{F}.b(\grave{sres}, \grave{res.b}, \grave{g})[t] = \mathcal{F}.b(\acute{sres}, \acute{res.b}, \acute{g})[t] \;\rangle\\
= & \quad \{ \text{ one-point rule to eliminate } \grave{p} \text{ and } \acute{p} \;\}\\
 & \langle \forall\, t, \grave{res.b}, \acute{res.b}, \grave{g}, \acute{g}, \grave{sres}, \acute{sres}, \grave{\varepsilon}, \acute{\varepsilon} \;::\\
 & \quad (\grave{sres}, \grave{\varepsilon})[t] = (\acute{sres}, \acute{\varepsilon})[t] \;\wedge\; \grave{res.b}[t] = \acute{res.b}[t] \;\wedge\; \grave{g}[t] = \acute{g}[t] \;\Rightarrow\\
 & \qquad \mathcal{F}.b(\grave{sres}, \grave{res.b}, \grave{g})[t] = \mathcal{F}.b(\acute{sres}, \acute{res.b}, \acute{g})[t] \;\rangle\\
= & \quad \{ \text{ (81): pairing and select, and (79): pairing and equality } \}\\
 & \langle \forall\, t, \grave{sres}, \acute{sres}, \grave{res.b}, \acute{res.b}, \grave{g}, \acute{g}, \grave{\varepsilon}, \acute{\varepsilon} \;::\\
 & \quad \grave{sres}[t] = \acute{sres}[t] \;\wedge\; \grave{res.b}[t] = \acute{res.b}[t] \;\wedge\; \grave{g}[t] = \acute{g}[t] \;\wedge\; \grave{\varepsilon}[t] = \acute{\varepsilon}[t] \;\Rightarrow\\
 & \qquad \mathcal{F}.b(\grave{sres}, \grave{res.b}, \grave{g})[t] = \mathcal{F}.b(\acute{sres}, \acute{res.b}, \acute{g})[t] \;\rangle\\
\Leftarrow & \quad \{ \text{ weaken antecedent } \}
\end{array}
$$

$$
\begin{array}{ll}
\text{D0:} & \mathcal{F}.a(\tilde{sres}, \ldots, \tilde{\varepsilon}) \\
 & \mathcal{F}.b(\tilde{sres}, \ldots) \\
\text{D1:} & \langle \forall \tilde{\varepsilon} :: \langle \forall \tilde{sres} :: \ \ldots \rangle\rangle \\
\text{D2:} & \langle \forall \tilde{\varepsilon} :: \tilde{\varepsilon} = \bar{\varepsilon} \Rightarrow \langle \forall \tilde{sres} :: \tilde{sres} = \bar{sres} \Rightarrow \ \ldots \rangle\rangle \\
\text{D3:} & w{:}\langle \forall s :: (\grave{sres}, \grave{\varepsilon})[s] = (\acute{sres}, \acute{\varepsilon})[s] \ \vee \ \ldots \rangle \\
\text{D4:} & \langle \forall t, \grave{sres}, \acute{sres}, \ldots, \grave{\varepsilon}, \acute{\varepsilon} :: \grave{sres}[t] = \acute{sres}[t] \ \wedge \ \ldots \ \wedge \ \grave{\varepsilon}[t] = \acute{\varepsilon}[t] \Rightarrow \\
 & \qquad \mathcal{F}.a(\grave{sres}, \ldots, \grave{\varepsilon})[t] = \mathcal{F}.a(\acute{sres}, \ldots, \acute{\varepsilon})[t] \rangle \\
 & \langle \forall t, \grave{sres}, \acute{sres}, \ldots :: \grave{sres}[t] = \acute{sres}[t] \ \wedge \ \ldots \Rightarrow \\
 & \qquad \mathcal{F}.b(\grave{sres}, \ldots)[t] = \mathcal{F}.b(\acute{sres}, \ldots)[t] \rangle \\
\text{D5:} & \langle \forall t{:}\, T, sres, \ldots, \varepsilon :: t \neq \mathbf{nil} \Rightarrow \mathcal{F}.a(sres, \ldots, \varepsilon)[t] = e \rangle \\
 & \langle \forall t{:}\, T, sres, \ldots :: t \neq \mathbf{nil} \Rightarrow \mathcal{F}.b(sres, \ldots)[t] = e \rangle
\end{array}
$$

Figure 25: A sketch of the working formula after fixing up pointwise axioms from Figure 24, which affects D4.

$$
\begin{array}{ll}
 & \langle \forall t, \grave{sres}, \acute{sres}, \grave{res}.b, \acute{res}.b, \grave{g}, \acute{g}, \grave{\varepsilon}, \acute{\varepsilon} :: \\
 & \qquad \grave{sres}[t] = \acute{sres}[t] \ \wedge \ \grave{res}.b[t] = \acute{res}.b[t] \ \wedge \ \grave{g}[t] = \acute{g}[t] \Rightarrow \\
 & \qquad\quad \mathcal{F}.b(\grave{sres}, \grave{res}.b, \grave{g})[t] = \mathcal{F}.b(\acute{sres}, \acute{res}.b, \acute{g})[t] \rangle \\
= & \qquad \{ \ \grave{\varepsilon} \text{ and } \acute{\varepsilon} \text{ do not occur free in the body of the quantification } \} \\
 & \langle \forall t, \grave{sres}, \acute{sres}, \grave{res}.b, \acute{res}.b, \grave{g}, \acute{g} :: \\
 & \qquad \grave{sres}[t] = \acute{sres}[t] \ \wedge \ \grave{res}.b[t] = \acute{res}.b[t] \ \wedge \ \grave{g}[t] = \acute{g}[t] \Rightarrow \\
 & \qquad\quad \mathcal{F}.b(\grave{sres}, \grave{res}.b, \grave{g})[t] = \mathcal{F}.b(\acute{sres}, \acute{res}.b, \acute{g})[t] \rangle
\end{array}
$$

This last line has the form of the pointwise axiom for $\mathcal{F}.b$ in $X(VC_D(m, C))$. Applying these transformations for every abstract variable to the working formula, we end up with a working formula whose shape is suggested by Figure 25. The strengthening step we did in the calculation for $\mathcal{F}.b$ has the effect of weaking the working formula, since by characteristic V0, rep axioms appear only in negative positions.

Step 6: From Figure 25 to Figure 26. We now have only one difference left between the working formula and formula $X(VC_D(m, C))$: shared-residue modification constraints. Consider the elided disjunct of the modification constraint shown in Figure 25. According to V3 (page 87), each modification constraint for *sres* in $VC_D(m, C)$ has the form

$$w{:}\langle \forall s :: \grave{sres}[s] = \acute{sres}[s] \ \vee \ F_D(modpoint_D(sres, w, s)) \rangle$$

$$
\begin{array}{ll}
\text{D0:} & \mathcal{F}.a(\tilde{sres}, \ldots, \tilde{\varepsilon}) \\
& \mathcal{F}.b(\tilde{sres}, \ldots) \\
\text{D1:} & \langle \forall \tilde{\varepsilon} :: \langle \forall \tilde{sres} :: \ \ldots \rangle\rangle \\
\text{D2:} & \langle \forall \tilde{\varepsilon} :: \tilde{\varepsilon} = \bar{\varepsilon} \Rightarrow \langle \forall \tilde{sres} :: \tilde{sres} = \bar{sres} \Rightarrow \ \ldots \rangle\rangle \\
\text{D3:} & w{:}\langle \forall s :: \grave{sres}[s] = \acute{sres}[s] \ \vee \ \ldots \rangle \ \wedge \ modcon_E(w, \varepsilon, sres) \\
\text{D4:} & \langle \forall t, \grave{sres}, \acute{sres}, \ldots, \grave{\varepsilon}, \acute{\varepsilon} :: \grave{sres}[t] = \acute{sres}[t] \ \wedge \ \ldots \ \wedge \ \grave{\varepsilon}[t] = \acute{\varepsilon}[t] \Rightarrow \\
& \qquad \mathcal{F}.a(\grave{sres}, \ldots, \grave{\varepsilon})[t] = \mathcal{F}.a(\acute{sres}, \ldots, \acute{\varepsilon})[t] \rangle \\
& \langle \forall t, \grave{sres}, \acute{sres}, \ldots :: \grave{sres}[t] = \acute{sres}[t] \ \wedge \ \ldots \Rightarrow \\
& \qquad \mathcal{F}.b(\grave{sres}, \ldots)[t] = \mathcal{F}.b(\acute{sres}, \ldots)[t] \rangle \\
\text{D5:} & \langle \forall t{:}\,T, sres, \ldots, \varepsilon :: t \neq \mathbf{nil} \Rightarrow \mathcal{F}.a(sres, \ldots, \varepsilon)[t] = e \rangle \\
& \langle \forall t{:}\,T, sres, \ldots :: t \neq \mathbf{nil} \Rightarrow \mathcal{F}.b(sres, \ldots)[t] = e \rangle
\end{array}
$$

Figure 26: The final sketch of the working formula, which includes the transformations of shared-residue modification constraints, which affect D3.

for some w and s. Now that we've transformed the working formula into what looks like $X(VC_D(m, C))$ except for modification constraints, we have transformed the elided $F_D(modpoint_D(sres, w, s))$ into $X(F_D(modpoint_D(sres, w, s)))$ for which

$$
\begin{array}{ll}
& X(F_D(modpoint_D(sres, w, s))) \\
= & \qquad \{ \ (68)\text{: } X \circ F_D = F_E \ \} \\
& F_E(modpoint_D(sres, w, s)) \\
= & \qquad \{ \ \text{lemma (33)} \ \} \\
& F_E(modpoint_E(sres, w, s))
\end{array}
$$

In summary, the elided disjunct of the modification constraint shown in Figure 25 has the form $F_E(modpoint_E(sres, w, s))$. We calculate,

$$
\begin{array}{ll}
& w{:}\langle \forall s :: (\grave{sres}, \grave{\varepsilon})[s] = (\acute{sres}, \acute{\varepsilon})[s] \ \vee \ F_E(modpoint_E(sres, w, s)) \rangle \\
= & \qquad \{ \ (81)\text{: pairing and select, and (79): pairing and equality} \ \} \\
& w{:}\langle \forall s :: (\grave{sres}[s] = \acute{sres}[s] \ \wedge \ \grave{\varepsilon}[s] = \acute{\varepsilon}[s]) \ \vee \ F_E(modpoint_E(sres, w, s)) \rangle \\
= & \qquad \{ \ \text{distribute } \wedge, \ \vee, \text{ and } \forall \ \} \\
& w{:}\langle \forall s :: \grave{sres}[s] = \acute{sres}[s] \ \vee \ F_E(modpoint_E(sres, w, s)) \rangle \ \wedge \\
& w{:}\langle \forall s :: \grave{\varepsilon}[s] = \acute{\varepsilon}[s] \ \vee \ F_E(modpoint_E(sres, w, s)) \rangle \\
= & \qquad \{ \ (24)\text{: definition of } modcon_E \ \} \\
& w{:}\langle \forall s :: \grave{sres}[s] = \acute{sres}[s] \ \vee \ F_E(modpoint_E(sres, w, s)) \rangle \ \wedge \\
& modcon_E(w, \varepsilon, sres)
\end{array}
$$

The last formula of this calculation has the form of shared-residue modification constraints in $X(VC_D(m, C))$, see D3 of Figure 18. Applying this transformation to the working formula, we end up with a working formula whose shape is suggested by Figure 26, which agrees on all accounts with the formula $X(VC_D(m, C))$. In summary, we have performed validity preserving steps that transform formula $VC_D(m, C)$ into formula $X(VC_D(m, C))$.

Tah-dah, we have proved X property (75), and thus also the Soundness Theorem. ∎

Part IV
Epilogue

We conclude with a few miscellaneous observations about our theorem and proof.

Length. We can easily imagine readers who doubt the utility of an 80-page proof. One purpose of a proof is to persuade. With an eye to this purpose, we placed the informal arguments to Soundness Lemma C as early as possible. Another purpose of a proof is to reveal errors. Regarding this purpose, we can report that our proof has already demonstrated its value: an earlier attempted proof came to ground on Saxe's counterexample in Section 6.3. Of course, we would be delighted to learn of a shorter proof.

Limitations of the VC generator. Observant readers may have noticed that the verification condition generator that we have proved to be modularly sound does not include the refinements explained in Section 8. That is, it does not allow "freeconditions" or "free modification of unused state". We suspect that we could remedy these limitations of the proof without major changes, but we have not checked the details.

Names of adornments. For most of the proof, it seems to make things easier if the number of different adornments is kept small. For example, this means that definition (24) of *modcon* can hard-wire the pre- and post-adornments $\grave{x}$ and $\acute{x}$, rather than taking them as additional parameters. But to apply the Chain of Equalities Lemma (page 113), we need to rename variables so that each generation of variables has its own unique adornment. As the Chain Rewriting Lemma (page 124), its proof, and the proof of the Chain of Equalities Corollary (page 127) show, this takes some effort.

The inclusion of Soundness Lemma D. The specialization Soundness Lemma D of the Soundness Theorem applies when the difference between scopes D and E is one abstract field declaration and a number of dependency declarations on that field. By including this lemma, the definition of X (Figure 17, page 134) can assume ε to be concrete. This means that various applications of F_E to expressions involving ε can be omitted in the definition of X, since F_E is the identity on concrete fields. This appears to reduce clutter in the proofs of the X properties, especially in the proof of X property (75) in Section A32. It also means that the proof of Soundness Lemma D, which uses Soundness Lemma C whose proof uses X and its properties, focuses only on the differences of the field ε being abstract versus it being concrete.

Selection determines maps. In the proofs of the Chain of Equalities Lemma (page 113) and the Chain Rewriting Lemma (page 124), we use axiom (44) (specifically, on pages 115 and 123), which says that two maps are equal if all of their elements are. In other words, there is no other feature of maps that distinguishes them. While we don't see anything wrong with using this axiom about select in our soundness proof, it is interesting that in our experience with applying ESC to real systems programs, we have not found a need to provide this axiom to the theorem prover. Stated differently, although our soundness proof relies on this axiom, whether or not the verification conditions we produce hold does not seem to rely on this axiom.

Selection on map pairs. The proof of X property (75) in Section A32 introduces three functions (pairing, car, and cdr) and postulates three axioms about these functions. (To be suggestive, these functions have familiar names, but they should nevertheless be considered to be fresh function symbols that do not clash with other interpreted or uninterpreted function symbols in the user language.)

The first of these axioms, (79), seems in no way controversial.

The second axiom, (80), is not controversial in its "only if" direction. The "if" direction implies that car and cdr are total functions that return two pieces a and b whose pairing is again p. Thus, there are no indivisible "atoms" on which car and cdr are not defined: everything is a "pair".

The third axiom, (81) relates the pairing and the previously defined function select. This may appear to lie beyond more usual, “trusted” axioms.

The fact that these axioms, and select axiom (44) for that matter, are used in the proof means in general that they should always appear in the background predicate as things that one can use to prove or disprove a verification condition. However, if one can prove that these axioms are a *conservative extension* of the select and stores axioms (43), then they need not be included in the background predicate. Being a conservative extension means that their inclusion does not allow one to prove more things about relevant formulas (in particular, generated verification conditions) that do not contain the new function symbols than one could without the additional axioms.

We have not proved that these axioms are a conservative extension. In fact, we know that, in general, they are not. In particular, if one mixes maps and non-maps (for example, by using an integer as the first argument to select), then one can find counterexamples that show that these axioms are not a conservative extension. However, by considering a multi-sorted logic that distinguishes between maps and non-maps, then the counterexamples we know do not apply. In such a multi-sorted logic, one needs two sets of the three functions in Section A32. The signatures of these functions would be

$$\begin{array}{l}
\mathit{select}\colon \mathit{Map} \times \mathit{NonMap} \rightarrow \mathit{NonMap} \\
\mathit{store}\colon \mathit{Map} \times \mathit{NonMap} \times \mathit{NonMap} \rightarrow \mathit{Map} \\
\mathit{mappair}\colon \mathit{Map} \times \mathit{Map} \rightarrow \mathit{Map} \\
\mathit{mapcar}\colon \mathit{Map} \rightarrow \mathit{Map} \\
\mathit{mapcdr}\colon \mathit{Map} \rightarrow \mathit{Map} \\
\mathit{nonmappair}\colon \mathit{NonMap} \times \mathit{NonMap} \rightarrow \mathit{NonMap} \\
\mathit{nonmapcar}\colon \mathit{NonMap} \rightarrow \mathit{NonMap} \\
\mathit{nonmapcdr}\colon \mathit{NonMap} \rightarrow \mathit{NonMap}
\end{array}$$

We claim our VC generation to be consistent with these signatures. In the multi-sorted logic, axioms (79) and (80) would be duplicated, once for each set of pairing functions, and axiom (81) would be written

$$\begin{array}{l}
\langle\, \forall\, a\colon \mathit{Map},\, b\colon \mathit{Map},\, t\colon \mathit{NonMap} \;:: \\
\qquad\qquad\qquad \mathit{mappair}(a, b)[t] \;\equiv\; \mathit{nonmappair}(a[t], b[t]) \,\rangle
\end{array}$$

We do not know whether or not these functions and axioms are a conservative extension in such a multi-sorted logic.

Use of modularity requirements and residues in the proof. As motivated in Sections 6 and 7.2, the soundness of modular verification relies crucially on the modularity requirements and the use of residue variables in the VC generation. So where are these actually used in the proof?

The visibility and top-down requirements are used in the proof of lemma (29). This lemma is used to prove lemma (32), which in turn proves lemma (33), which is used in two places. First, lemma (33) proves the three-part lemma (76)–(77)–(78), which in turn is used to prove the X properties (72) and (73). Second, lemma (33) is used in the proof of X property (75).

The visibility requirement is also used in the first case in the proof of Soundness Lemma F and the top-down requirement is used in the second case of that proof.

Residues appear more ubiquitously in the proof, but the lemma that cannot be proved without them is Soundness Lemma C. More precisely, individual residue variables are vital to the Chain of Equalities Corollary (page 127) and shared residue variables are vital to X property (75).

References

[0] Martín Abadi and Luca Cardelli. *A theory of objects*. Springer-Verlag, 1996.

[1] Martín Abadi and K. Rustan M. Leino. A logic of object-oriented programs. In *Theory and Practice of Software Development: Proceedings / TAPSOFT '97, 7th International Joint Conference CAAP/FASE*, volume 1214 of *Lecture Notes in Computer Science*, pages 682–696. Springer, April 1997. Expanded version available as Research Report 161, Digital Equipment Corporation Systems Research Center, September 1988.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[3] Paulo Sérgio Almeida. Balloon types: Controlling sharing of state in data types. In Mehmet Akşit and Satoshi Matsuoka, editors, *ECOOP'97—Object-oriented Programming: 11th European Conference*, volume 1241 of *Lecture Notes in Computer Science*, pages 32–59. Springer, June 1997.

[4] American National Standards Institute, Inc. *The Programming Language Ada Reference Manual*, volume 155 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, Germany, 1983. ANSI/MIL-STD-1815A-1983.

[5] R. J. R. Back. *Correctness Preserving Program Refinements: Proof Theory and Applications*, volume 131 of *Mathematical Centre Tracts*. Mathematical Centre, Amsterdam, 1980.

[6] John Boyland. Alias burying: Unique variables without destructive reads. *Software—Practice & Experience*. To appear.

[7] David L. Detlefs, K. Rustan M. Leino, and Greg Nelson. Wrestling with rep exposure. Research Report 156, Digital Equipment Corporation Systems Research Center, July 1998.

[8] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Research Report 159, Compaq Systems Research Center, December 1998.

[9] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, NJ, 1976.

[10] Edsger W. Dijkstra and Carel S. Scholten. *Predicate Calculus and Program Semantics*. Texts and Monographs in Computer Science. Springer-Verlag, 1990.

[11] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley Publishing Company, 1990.

[12] George W. Ernst, Raymond J. Hookway, and William F. Ogden. Modular verification of data abstractions with shared realizations. *IEEE Transactions on Software Engineering*, 20(4):288–307, April 1994.

[13] Extended Static Checking for Java home page, Compaq Systems Research Center. On the web at `http://research.compaq.com/SRC/esc/`.

[14] P. H. B. Gardiner and Carroll Morgan. A single complete rule for data refinement. *Formal Aspects of Computing*, 5(4):367–382, 1993.

[15] James Gosling, Bill Joy, and Guy Steele. *The Java™ Language Specification*. Addison-Wesley, 1996.

[16] David Gries and Jan Prins. A new notion of encapsulation. In *Proceedings of the ACM SIGPLAN 85 Symposium on Language Issues in Programming Environments*, volume 20, number 7 in *SIGPLAN Notices*, pages 131–139. ACM, July 1985.

[17] David Gries and Dennis Volpano. The transform — a new language construct. *Structured Programming*, 11(1):1–10, 1990.

[18] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1(4):271–81, 1972.

[19] C. A. R. Hoare and N. Wirth. An axiomatic definition of the programming language PASCAL. *Acta Informatica*, 2(4):335–355, 1973.

[20] John Hogg. Islands: Aliasing protection in object-oriented languages. In Andreas Paepcke, editor, *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '91)*, volume 26, number 11 in *SIGPLAN Notices*, pages 271–285. ACM, November 1991.

[21] Daniel Jackson. Aspect: Detecting bugs with abstract dependences. *ACM Transactions on Software Engineering and Methodology*, 4(2):109–145, April 1995.

[22] Cliff B. Jones. *Systematic Software Development using VDM*. International Series in Computer Science. Prentice-Hall, Englewood Cliffs, NJ, 1986.

[23] H. B. M. Jonkers. Upgrading the pre- and postcondition technique. In S. Prehn and W. J. Toetenel, editors, *VDM '91 Formal Software Development Methods: 4th International Symposium of VDM Europe. Volume 1: Conference Contributions*, volume 551 of *Lecture Notes in Computer Science*, pages 428–456. Springer-Verlag, October 1991.

[24] Rajeev Joshi. Extended static checking of programs with cyclic dependencies. In James Mason, editor, *1997 SRC Summer Intern Projects*, Technical Note 1997-028. Digital Equipment Corporation Systems Research Center, 1997.

[25] Leslie Lamport and Fred B. Schneider. Constraints: A uniform approach to aliasing and typing. In *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 205–216, January 1985.

[26] Gary T. Leavens. An overview of Larch/C++: Behavioral specifications for C++ modules. In Haim Kilov and William Harvey, editors, *Specification of Behavioral Semantics in Object-Oriented Information Modeling*, chapter 8, pages 121–142. Kluwer Academic Publishers, 1996.

[27] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06f, Iowa State University, Department of Computer Science, July 1999. Available at `ftp://ftp.cs.iastate.edu/pub/techreports/TR98-06/`.

[28] K. Rustan M. Leino. *Toward Reliable Modular Programs*. PhD thesis, California Institute of Technology, 1995. Technical Report Caltech-CS-TR-95-03.

[29] K. Rustan M. Leino. Ecstatic: An object-oriented programming language with an axiomatic semantics. In *The Fourth International Workshop on Foundations of Object-Oriented Languages*, January 1997. Proceedings available from `http://www.cs.williams.edu/~kim/FOOL/`.

[30] K. Rustan M. Leino. Data groups: Specifying the modification of extended state. In *Proceedings of the 1998 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '98)*, volume 33, number 10 in *SIGPLAN Notices*, pages 144–153. ACM, October 1998.

[31] K. Rustan M. Leino. Recursive object types in a logic of object-oriented programs. *Nordic Journal of Computing*, 5(4):330–360, Winter 1998.

[32] K. Rustan M. Leino, Greg Nelson, and James B. Saxe. ESC/Java user's manual. Technical Note 2000-002, Compaq Systems Research Center, October 2000.

[33] K. Rustan M. Leino and Raymie Stata. Checking object invariants. Technical Note 1997-007, Digital Equipment Corporation Systems Research Center, January 1997.

[34] K. Rustan M. Leino and Raymie Stata. Virginity: A contribution to the specification of object-oriented software. *Information Processing Letters*, 70(2):99–105, April 1999.

[35] Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development*. MIT Electrical Engineering and Computer Science Series. MIT Press, 1986.

[36] John McCarthy and James Painter. Correctness of a compiler for arithmetic expressions. In J.-T. Schwartz, editor, *Proceedings of Symposia in Applied Mathematics*. American Mathematical Society, 1967.

[37] Robin Milner. An algebraic definition of simulation between programs. Technical Report Stanford Artificial Intelligence Project Memo AIM-142, Computer Science Department Report No. CS-205, Stanford University, February 1971.

[38] James G. Mitchell, William Maybury, and Richard Sweet. The Mesa language manual, version 5.0. Technical Report CSL-79-3, Xerox PARC, April 1979.

[39] Joseph M. Morris. Laws of data refinement. *Acta Informatica*, 26(4):287–308, February 1989.

[40] H. Mössenböck and N. Wirth. The programming language Oberon-2. *Structured Programming*, 12(4):179–195, 1991.

[41] Peter Müller and Arnd Poetzsch-Heffter. Modular specification and verification techniques for object-oriented software components. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, chapter 7, pages 137–159. Cambridge University Press, 2000.

[42] Greg Nelson. Verifying reachability invariants of linked structures. *Conference Record of the Tenth Annual ACM Symposium on Principles of Programming Languages*, pages 38–47, January 1983.

[43] Greg Nelson. A generalization of Dijkstra's calculus. *ACM Transactions on Programming Languages and Systems*, 11(4):517–561, 1989.

[44] Greg Nelson, editor. *Systems Programming with Modula-3*. Series in Innovative Technology. Prentice-Hall, Englewood Cliffs, NJ, 1991.

[45] James Noble, Jan Vitek, and John Potter. Flexible alias protection. In Eric Jul, editor, *ECOOP'98—Object-oriented Programming: 12th European Conference*, volume 1445 of *Lecture Notes in Computer Science*, pages 158–185. Springer, July 1998.

[46] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972. Reprinted as `http://www.acm.org/classics/may96/`.

[47] J. E. Stoy and C. Strachey. OS6—an experimental operating system for a small computer. Part II: Input/output and filing system. *The Computer Journal*, 15(3):195–203, 1972.

[48] Mark Utting. Reasoning about aliasing. In *Proceedings of the Fourth Australasian Refinement Workshop (ARW-95)*, pages 195–211. School of Computer Science and Engineering, The University of New South Wales, April 1995.

[49] N. Wirth. The programming language Oberon. *Software—Practice & Experience*, 18(7):671–690, July 1988.

[50] Niklaus Wirth. Modula: a language for modular multiprogramming. *Software—Practice & Experience*, 7(1):3–35, January–March 1977.

[51] Niklaus Wirth. *Programming in Modula-2*. Springer-Verlag, 1982.