July 29, 1998

SRC Research Report

156

Wrestling with rep exposure

David L. Detlefs
K. Rustan M. Leino
Greg Nelson

digital

Systems Research Center
130 Lytton Avenue
Palo Alto, California 94301

http://www.research.digital.com/SRC/

Systems Research Center

The charter of SRC is to advance both the state of knowledge and the state of the art in computer systems. From our establishment in 1984, we have performed basic and applied research to support Digital's business objectives. Our current work includes exploring distributed personal computing on multiple platforms, networking, programming technology, system modelling and management techniques, and selected applications.

Our strategy is to test the technical and practical value of our ideas by building hardware and software prototypes and using them as daily tools. Interesting systems are too complex to be evaluated solely in the abstract; extended use allows 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 strategy, including personal computing, distributed systems, and the Internet.

We also perform complementary work of a more mathematical flavor. Some of it is in established fields of theoretical computer science, such as the analysis of algorithms, computational geometry, and logics of programming. Other work explores new ground motivated by problems that arise in our systems research.

We have a strong commitment 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. We seek users for our prototype systems among those with whom we have common interests, and we encourage collaboration with university researchers.

Wrestling with rep exposure

David L. Detlefs, K. Rustan M. Leino, and Greg Nelson

July 29, 1998

Author Affiliations David L. Detlefs is a staff engineer at Sun Microsystems Laboratories. He can be reached at `david.detlefs@sun.com`. This work was completed by him and the other two authors before he left SRC in 1996.

Abstract

A central methodological problem in programming with multiple levels of abstractions is the loosely defined problem of *rep exposure*. This paper traces the problem of rep exposure to the precisely defined notion of *abstract aliasing*. The paper also outlines a statically-enforceable discipline for avoiding abstract aliasing, but the outline is incomplete.

0 Introduction

The danger of *rep exposure* arises when a reference to a mutable component of an abstract data type is transferred into or out of the scope in which the representation of the data type is hidden. The danger is that operations on the mutable component could affect the value of the abstract data value (or vice-versa: operations on the abstract data value could affect the value of the mutable component). In the scope where the representation of the abstract data type is known, these interference effects are predictable and can be reasoned about using the method of data abstraction described by C.A.R. Hoare in 1972 [Hoa72]. But outside the scope, they are unpredictable and seem difficult to reason about.

For example, if a stack s were implemented in terms of a sequence $q[s]$, then a series of $push$ and pop operations on s would not behave as expected if there were interleaved updates to $q[s]$, and vice versa.

The danger of rep exposure could be avoided by prohibiting the transfer of mutable components across abstraction boundaries. But this is too strict: it would prohibit many useful programs. We mention three examples:

- The initialization method for an abstract type may well take mutable parameters that become part of the initialized abstract value.
- For efficiency reasons, it may be desirable to return a mutable component of an abstract data type to a client; perhaps with restrictions on the operations that the client can perform on the component.
- If the abstract data type is a "container class" (such as a set, sequence, or table), the elements inserted and removed from the container may well be mutable. But the container would be truly useless if its methods were prohibited from storing an in-parameter into the container or from returning an element from the container as a out-parameter.

An effective methodology for dealing with rep exposure would allow what is useful while preventing what is harmful. In this paper, we introduce a methodology that we think is a step toward this goal. Our methodology it not a full solution, but it applies to many example programs that we have studied, it is simple, and it can be enforced mechanically.

Our approach builds on our previous work in reasoning about modular verification in terms of *dependencies*, as introduced by Leino in his Ph.D. thesis [Lei95] and extended by Leino and Nelson [LN98a]. We will try to make this paper accessible to readers who don't know about dependencies, by defining the relevant terms as we need them.

1 Definitions

A *program* is a collection of *declarations*. Declarations introduce names for entities (such as types, abstract and concrete data fields, methods) and/or specify properties of named entities (such as subtype relationships, representations of abstract fields, method specifications and method implementations). The declarations of a program are partitioned into *units* (sometimes called interfaces and modules). The declarations visible (that is, in scope) in a unit are its own declarations and the declarations visible in units that it *imports*.

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

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

we write

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

Also, we write $f[o]$ where others write $o.f$. This semantics models an implementation in which objects are references to data records containing field values, and in which two objects are equal when they reference the same data record.

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

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

An abstract field occupies no memory at run-time; it is a fictitious field whose value (or *representation*) is later given in terms of other fields. This representation is declared by a syntax like

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

The variables appearing in the right-hand side of the **rep** construct for an abstract variable are called the *dependencies* of the abstract variable. The dependencies can themselves be either concrete or abstract.

We require that dependencies be declared explicitly. For example, the representation (0) would cause a static error unless $f[t]$ were declared as a dependency of $\mathit{valid}[t]$, which is done by a declaration of the form

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

We allow only two forms of dependencies in this paper: *static* dependencies of the form

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

and *dynamic* dependencies of the form

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

where b is a concrete field. If all the dependencies in a program are static, then the representation of each abstraction is confined to the fields of a single object, but if the program contains dynamic dependencies, then some abstraction's representation includes fields of multiple objects connected by references (like the field b in (2)). We call the field b a *pivot field*.

We impose the rule that the static dependency (1) be visible wherever c is, and the dynamic dependency (2) be visible wherever b is. These rules seem necessary for modular soundness, as explained in our companion paper [LN98a]. Because of the rule that a dynamic dependency must be visible anywhere its pivot field is, it follows that in any scope where a field is visible, it is known whether the field is a pivot field or not.

Dependencies affect the verification process. For example, in a scope where **depends** $a[t]$ **on** $c[t]$ is visible, a procedure call that is known to change $a[t]$ is assumed by the verifier to possibly change $c[t]$. But unless the actual **rep** clause for $a[t]$ is also visible, nothing can be assumed about the nature of the change to $c[t]$. Thus dependencies are abstractions of **rep** clauses: they specify what is part of the representation of what, but they hide the explicit nature of the representation. This makes them valuable in dealing with rep exposure.

The distinction between static and dynamic dependencies allows us to give a more precise account of rep exposure: only the mutable components corresponding to dynamic dependencies are dangerous (because with multiple objects, unexpected interference may occur in scopes where the pivot field and dependency are not visible); those associated with static dependencies are not (because with a single object, there is no pivot field to be out of scope).

To prevent harmful rep exposure, we claim that it suffices to prevent *abstract aliasing*, which roughly means to prevent values of a pivot field from escaping the scope where the field is declared. The precise definition of abstract aliasing is somewhat odd, since it involves both the static program text and dynamic possibility. We say that $a[E]$ and $c[F]$ are *directly abstractly aliased* at some dynamic execution point if $F = b[E] \ \wedge \ F \neq \textbf{nil}$ holds for some dependency **depends** $a[t]$ **on** $c[b[t]]$ that is not in scope at the corresponding program point (which implies that b is not in scope either at that program point). Notice that the condition $F = b[E]$ makes sense even outside the scope of b, since b's value exists even at an execution point where b is not visible at the corresponding program point. Two expressions of the form $a[E]$ and $c[F]$ are *abstractly aliased* at some dynamic execution point if they are related by the transitive closure of the direct abstract aliased relation for that execution point, and all free variables of $a[E]$ and

$c[F]$ are in scope at the corresponding program point. In a program without information hiding, where all variables are in scope everywhere, abstract aliasing never occurs.

2 Example

As an example that will be useful here and later in the paper, we will consider a design of a lexer abstraction built on top of a reader abstraction.

A *reader* models an input stream. Here is a part of the unit declaring the reader abstraction:

$$
\begin{array}{l}
\textbf{unit } \mathit{Rd} \\
\quad \textbf{type } \mathit{Rd.T} \\
\quad \textbf{spec var } \mathit{rvalid}\colon \mathit{Rd.T} \to \textbf{bool} \\
\qquad \vdots \\
\quad \textbf{proc } \mathit{GetChar}(\mathit{rd}\colon \mathit{Rd.T})\colon \textbf{char} \\
\qquad \textbf{requires } \mathit{rvalid}[\mathit{rd}] \\
\qquad \vdots \\
\quad \textbf{proc } \mathit{Close}(\mathit{rd}\colon \mathit{Rd.T}) \\
\qquad \textbf{modifies } \mathit{rvalid}[\mathit{rd}] \qquad .
\end{array}
$$

This declares a type $Rd.T$ and a boolean-valued abstract field $rvalid$ that records whether objects of that type are valid. Typically, there will be many operations that require and preserve validity, of which we show one example, $GetChar$. We have also shown one procedure, $Close$, that destroys validity. The interface does not declare the fields that are relevant only to the implementation of readers. These fields would be declared in another unit, an implementation unit. The implementation unit would also provide the representation of the abstract field $rvalid$.

A *lexer* is an abstraction that returns lexical tokens. The interface to lexers is very similar to that of readers:

$$
\begin{array}{l}
\textbf{unit } \mathit{Lexer} \textbf{ import } \mathit{Token} \\
\quad \textbf{type } \mathit{Lexer.T} \\
\quad \textbf{spec var } \mathit{lvalid}\colon \mathit{Lexer.T} \to \textbf{bool} \\
\qquad \vdots \\
\quad \textbf{proc } \mathit{GetToken}(\mathit{lx}\colon \mathit{Lexer.T})\colon \mathit{Token.T} \\
\qquad \textbf{requires } \mathit{lvalid}[\mathit{lx}] \\
\qquad \vdots
\end{array}
$$

The implementation of lexers contains a variety of fields, of which we show one, rdr, which is a reference to the reader that supplies the stream to be converted into tokens:

$$
\begin{array}{l}
\textbf{unit}\ \mathit{LexerImpl}\ \textbf{import}\ \mathit{Lexer}, \mathit{Rd}, \mathit{Token} \\
\quad \textbf{var}\ \mathit{rdr}\colon \mathit{Lexer}.T \rightarrow \mathit{Rd}.T \\
\quad \textbf{depends}\ \mathit{lvalid}[\mathit{lx}\colon \mathit{Lexer}.T]\ \textbf{on}\ \mathit{rvalid}[\mathit{rdr}[\mathit{lx}]] \\
\quad \textbf{rep}\ \mathit{lvalid}[\mathit{lx}\colon \mathit{Lexer}.T]\ \equiv\ \mathit{rvalid}[\mathit{rdr}[\mathit{lx}]]\ \wedge\ \ldots \\
\qquad \vdots
\end{array}
$$

The idea is that the reader $rdr[lx]$ supplies the character stream that is tokenized by the lexer lx. We omit the other data fields of lexers, and the part of the representation of $lvalid$ that concerns these fields.

In fact, the field rdr is a pivot field, since we need the conjunct $rvalid[rdr[lx]]$ in the representation of $lvalid[lx]$, and therefore there is a dynamic dependency of $lvalid$ on $rvalid$.

Envision a situation where the lexer interface provides a procedure, P, that returns the associated reader. A client could then use procedure P to retrieve the reader of a lexer, close the reader, and then operate on the lexer:

$$
\begin{array}{l}
\text{given } \mathit{lvalid}[\mathit{lx}], \\
\quad \mathit{rd} := P(\mathit{lx})\ ; \\
\quad \mathit{Close}(\mathit{rd})\ ; \\
\qquad \vdots \\
\quad \mathit{tok} := \mathit{GetToken}(\mathit{lx}) \qquad .
\end{array}
$$

This would go wrong at run-time (because the lexer is invalidated by the closing of the reader), but a modular verifier would miss the error. Note that in this program fragment, abstract aliasing occurs as defined in Section 1: after the line $rd := P(lx)$, the condition $rd = rdr[lx] \wedge rd \neq \textbf{nil}$ holds, and (we assume) this is a scope where the rdr field is not visible. (If the rdr field were visible, then the dependency would be too, and a modular verifier would not miss the error.)

Informally, we say that P causes *simple upward leaking*: it creates the possibility of abstract aliasing by "leaking" the value of a pivot field by directly returning it to a caller outside the scope of the field.

3 Practical basis

We have not yet implemented our methodology. We had planned to do so in the context of the Modula-3 Extended Static Checker (ESC) [DLNS98, LN98b, Det96, ESC], but this plan was not completed. In the meantime, the design of our methodology was shaped by the following programs:

- The Modula-3 object-oriented buffered streams package (readers and writers; about a thousand lines of Modula-3; originally written in Modula-2+ by Kai Li and Butler Lampson) [BN91].
- The NI2 indexing library (about 20,000 lines of C code written by Mike Burrows; the heart of the AltaVista World-Wide Web Index).

Both of these libraries use object-oriented programming techniques to implement abstract data types. Both of them hide the representation of their abstract types, and both of them occasionally transfer mutable components of the hidden representation into and out of the scope in which the representations are hidden, thus risking rep exposure.

We have annotated the I/O streams library and checked it with ESC. Since the methodology of this paper was not implemented, we never mechanically verified the absence of abstract aliasing, but we did mechanically check the package for many other errors, including race conditions, deadlocks, array index errors, and **nil** -dereference errors. Thus we have identified all the pivot fields and are aware of all places in which mutable components of abstract types cross abstraction boundaries. We believe that our methodology for avoiding abstract aliasing is flexible enough to handle this library.

Mike Burrows has checked the NI2 indexing library with LCLint [Eva96], which warns about transfers of mutable components across abstraction boundaries. He found that these warnings were not indicative of real errors, and therefore used the "`-repexpose`" flag to LCLint, which globally suppresses all warnings of this type. We ran LCLint without this flag and examined the warnings that it reported. We believe this showed us the parts of the program relevant to abstract aliasing, while saving us the need to read all 20,000 lines. It is possible, but not certain, that our methodology for avoiding abstract aliasing is flexible enough to handle this library.

The experience of Mike Burrows with LCLint and NI2 suggests that abstract aliasing may be more of a theoretical than a practical problem. It is clear that rep exposure makes simple program proof systems unsound. But it is not clear whether inadvertent rep exposure is a common source of errors.

4 Methodology

This section describes our methodology informally.

To prevent simple upward leaking, we impose the somewhat drastic restriction that no return value (or, more generally, out-parameter) of any procedure is allowed to be the value of any pivot field.

Slightly more subtly, a procedure could leak the value of a pivot field by assigning it to a global variable or to some field of some other object. We defend against this with another drastic restriction: at any instant, the values of pivot fields are not allowed to overlap with the values of global variables, nor with the values of non-pivot fields. We call this restriction *apartheid*. For brevity in the future, a *non-pivot location* means a global variable or non-pivot field. Thus, apartheid forbids any overlap between the values of pivot fields and the values of non-pivot locations.

Let us say that a procedure *captures* an in-parameter if it assigns the parameter to some global variable or field (pivot or not). In order to enforce apartheid, each procedure specification will have to disclose whether it captures any of its parameters. For a formal parameter that is captured into a pivot field, the procedure specification must require the value of the parameter to be distinct from the values of all non-pivot locations. Otherwise, the checker would complain that the procedure body may violate apartheid. Similarly, for a formal parameter that is captured into a non-pivot location, the specification must require the value of the parameter to be distinct from the values of all pivot fields. Our methodology includes a **captures** notation to make it convenient to write these kinds of specifications.

So far our methodology allows a computation to transfer a value between a pivot field and a non-pivot location, so long as no value is simultaneously shared between a pivot field and a non-pivot location. We have noticed that this freedom is unused in the example programs that we have encountered, and it appears to us that giving up this freedom simplifies our methodology. Therefore, we make another drastic rule, which we call *monomorphism*. The rule is that once an object becomes the value of a pivot field, it is no longer allowed ever to become the value of a non-pivot location, and vice versa. This rule is reminiscent of Leino and Stata's technique for keeping track of which objects have reference count zero without keeping track of the exact reference count of objects [LS97].

In light of the monomorphism rule, we make the following definitions. At a particular state in a computation, we say that a value is a *pivot* if it is or has been the value of some pivot field, that it is *plenary* if it is or has been the value of some non-pivot location, and that it is *virgin* otherwise.

Our methodology also includes what we call the *disjoint ranges requirement*. It states that pivot fields declared in distinct scopes have disjoint ranges. That is, if b and d are pivot fields whose declarations occur in different scopes, then $b[s] = d[t] \;\wedge\; b[s] \neq \textbf{nil}$ is forbidden, for any s and t. The justification for this requirement is rather technical and is explained in our companion paper on dependencies [LN98a].

5 Enforcing the methodology

In this section, we describe our methodology more formally, and also describe how to enforce it mechanically. Given as input a program annotated with specifications, we show how to transform it into another annotated program that will verify exactly when the input program would verify and the input program obeys our methodology.

For our purposes, it is not necessary to describe the specification language in any detail. We assume the reader is familiar with pre- and postconditions and modifies clauses, which we introduce with the Larch keywords **requires**, **ensures**, and **modifies** (see, for example, the CLU book [LG86]).

The methodology is enforced in three steps: we introduce some special fields, we transform the input program, and we transform the input specifications.

It is a consequence of our methodology that an object starts off being virgin and can transition into being either plenary or a pivot, but not both. To keep track of these transitions, we introduce three special boolean-valued fields: virgin, pivot, and $\mathit{plenary}$. These fields are special in that they are used only in describing the semantics of programs—the input program cannot read or write them directly. Furthermore, the fields pivot and $\mathit{plenary}$ are not allowed to occur directly in specifications; they can be introduced into specifications only using the constructs described in this section.

We transform the input program to an equivalent program that keeps the special fields up-to-date. The following table shows how the transformation is done. In the table, T denotes any object type, E any object-valued expression, b any pivot field, f any non-pivot field, g any global variable, and o any object.

input program	transformed program
$o := \mathbf{new}(T)$	$o := \mathbf{new}(T)$; $\mathit{virgin}[o] := \mathit{true}$; $\mathit{pivot}[o] := \mathit{false}$; $\mathit{plenary}[o] := \mathit{false}$
$b[o] := E$	$b[o] := E$; $\mathit{virgin}[b[o]] := \mathit{false}$; $\mathit{pivot}[b[o]] := \mathit{true}$
$f[o] := E$	$f[o] := E$; $\mathit{virgin}[f[o]] := \mathit{false}$; $\mathit{plenary}[f[o]] := \mathit{true}$
$g := E$	$g := E$; $\mathit{virgin}[g] := \mathit{false}$; $\mathit{plenary}[g] := \mathit{true}$

As a consequence of these transformations, the following predicates are guaranteed to hold at all control points in the input program. In these formulas, o ranges over non-**nil** objects, and b, f, and g are used as in the table above.

$$\langle \forall o :: \mathit{virgin}[o] \neq (\mathit{pivot}[o] \vee \mathit{plenary}[o]) \rangle$$

$$\langle \forall o :: \mathit{pivot}[b[o]] \rangle \wedge \langle \forall o :: \mathit{plenary}[f[o]] \rangle \wedge \mathit{plenary}[g]$$

(Since $b[o]$ and g may be **nil**, we introduce the boundary assumptions $pivot[\textbf{nil}]$ and $plenary[\textbf{nil}]$.)

The third step in enforcing our methodology is to transform the specifications. This involves desugaring each **captures** clause and inserting extra conditions in pre- and postconditions.

There are two forms of **captures** clauses. The first form is

$$\textbf{captures}\ o \quad ,$$

where o is an in-parameter. It desugars into

$$\begin{array}{l} \textbf{requires}\ o = \textbf{nil} \lor \neg pivot[o] \\ \textbf{modifies}\ virgin[o], plenary[o] \end{array} \quad .$$

Thus, if a procedure specification includes **captures** o, the procedure implementation is assured that o is not a pivot and is allowed to capture o into a non-pivot location.

The second form is

$$\textbf{captures}\ o\ \textbf{into}\ b[t] \quad ,$$

where o is an in-parameter, b is a pivot field, and t is an expression. It desugars into

$$\begin{array}{l} \textbf{requires}\ o = \textbf{nil} \lor virgin[o] \\ \textbf{modifies}\ virgin[o], pivot[o], b[t] \\ \textbf{ensures}\ b'[t] = o \end{array} \quad ,$$

where b' denotes the value of b in the post-state. Thus, if a procedure specification includes **captures** o **into** $b[t]$, the procedure implementation is assured that o is virgin and is constrained to capture o into the specific pivot field $b[t]$.

The **captures into** annotation can be viewed as a more precise version of LCLint's annotation `exposed`. Thus, we write

$$\begin{array}{l} \textbf{proc}\ m(v, x) \\ \quad \textbf{captures}\ x\ \textbf{into}\ b[v] \end{array}$$

where in LCLint one would write

$$\textbf{proc}\ m(v,\ \texttt{/*@exposed@*/}\ x) \quad .$$

Perhaps surprisingly, **captures into** is not a strengthening of **captures**: the latter is used when a parameter is captured into a non-pivot location, the former when a parameter is captured into a pivot field. The reason for the asymmetry is as follows. When a parameter is captured into a pivot field, the soundness of the methodology requires explicit mention of the pivot field in the specification (as we shall see below). But when a parameter is captured into a non-pivot location, explicit mention of the non-pivot location is not necessary, and typically not useful.

If there is no **captures** clause for some parameter, a caller can pass a pivot as the corresponding actual parameter: since the parameter is not captured, there is no danger that the procedure call would violate apartheid.

All that remains is to strengthen the pre- and postconditions of the input program.

We strengthen each procedure's postcondition by a conjunct

$$r = \textbf{nil} \lor \neg pivot'[r] \tag{3}$$

for each of its out-parameters r (including the return value). This enforces the rule against returning pivots.

Finally, we add invariants as conjuncts to all pre- and postconditions. To enforce apartheid, we add

$$\langle \forall\, o \;::\; \neg(pivot[o] \land plenary[o]) \;\rangle \qquad .$$

To enforce the disjoint ranges requirement, we add

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

for each pair of pivot fields b and d that are visible and whose declarations are in distinct scopes. In these formulas, o, s, and t range over non-**nil** objects.

Note that the apartheid invariant mentions only the fields $pivot$ and $plenary$, both of which change monotonically in any execution. Therefore it is impossible for a procedure to temporarily violate and then restore apartheid. This observation permits the invariant to be checked incrementally, instead of at procedure boundaries: one can check $\neg pivot[x]$ at every update of $plenary[x]$ and check $\neg plenary[x]$ at every update of $pivot[x]$.

The disjoint ranges requirement does not necessarily change monotonically, so it is enforced only on procedure boundaries.

6 What we have achieved

The contribution of our methodology is that it allows passing a pivot value across an abstraction boundary in cases where this is useful. In this section, we illustrate this contribution by continuing the lexer/reader example.

Recall that each lexer lx contains a reader $rdr[lx]$ that supplies the characters that lx tokenizes. We argue that this reader should be a parameter to the lexer initialization method, so that, for example, a lexer from a file could be constructed as follows:

$$\begin{array}{l} rd := File.Open(\texttt{"/etc/passwd"})\;; \\ lx := Lexer.Init(\mathbf{new}(Lexer.T),\, rd) \qquad . \end{array} \tag{4}$$

The alternative would be to make the lexer implementation itself allocate the reader. But then it could support only a fixed number of reader subtypes. By making the reader a parameter to the lexer initialization method, any reader subtype can be used, even subtypes that were not envisioned at the time the lexer implementation was coded. This is one of the key advantages of object-oriented programming.

Since the lexer initialization method captures its reader parameter, our methodology forces this fact to be disclosed in its specification:

$$\begin{array}{l} \mathbf{proc}\ Lexer.Init(lx\colon Lexer.T\ ;\ rd\colon Rd.T)\colon Lexer.T \\ \quad \mathbf{requires}\ lx \neq \mathbf{nil} \wedge virgin[lx] \wedge rvalid[rd] \\ \quad \mathbf{captures}\ rd\ \mathbf{into}\ rdr[lx] \\ \quad \mathbf{modifies}\ lvalid[lx] \\ \quad \mathbf{ensures}\ lvalid'[lx] \wedge \mathbf{result} = lx \qquad . \end{array}$$

(The precondition $virgin[lx]$ is not absolutely necessary, but it is convenient as will be explained in Section 8.) $Lexer.Init$ returns the lexer that it initializes, following a common convention for initialization methods.

Because $Lexer.Init$ captures its rd parameter into a pivot field, its specification must use the second form of **captures**, in which the pivot field rdr is mentioned explicitly. Consequently, the field must be visible in the interface, rather than hidden in the implementation. Therefore, we must move the declaration of the rdr field from the unit $LexerImpl$ to the unit $Lexer$. Because of our rules for the placement of dependencies, the declaration

$$\mathbf{depends}\ lvalid[lx\colon Lexer.T]\ \mathbf{on}\ rvalid[rdr[lx]]$$

must also move from $LexerImpl$ to $Lexer$.

At first these changes seem shocking and bad, but they are necessary and harmless. Necessary, because immediately after the call $Lexer.Init(lx, rd)$ both lx and rd are visible expressions and $rd = rdr[lx]$. Therefore, if the field rdr were not visible, abstract aliasing would occur by the definition in Section 1. Harmless, because the exposure of the rdr field in the interface does not entail any real loss of abstraction: the **rep** clause of $lvalid$ is still hidden in the implementation. In fact, the pivot field is effectively read-only to clients: since the dependency is visible, changes to $rdr[lx]$ are known to affect $lvalid[lx]$, but, since the **rep** is not visible, it is impossible to prove that a change to $rdr[lx]$ maintains $lvalid[lx]$. A more subtle example of this "read-only by specification" technique is described by Leino and Nelson [LN98a].

In summary, we can soundly allow pivots to cross abstraction boundaries by placing the dependency in the interface: The abstraction representation remains hidden in the implementation. If a perverse client tried to close the reader from under the lexer, the checker would detect that this action compromises the validity of the lexer, because the dependency is in scope. Indeed, we have structured the annotation language to railroad the programmer into this pattern: To verify the body of a procedure that captures a pivot, the programmer must specify it using the **captures into** notation, and thus must declare the pivot field in the interface, and therefore must declare the dependency there as well.

7 What we have not achieved

Our approach carefully controls the global variables and fields that can contain a pivot $b[o]$, but it doesn't say anything about how the *owner* of this pivot, o, might be reached. For example, our methodology does not prevent the code fragment (4) that initializes rd and lx to be followed by a procedure call like

$$P(lx, rd) \qquad ,$$

where P is a procedure that is implemented where the rdr field is not visible. In this scenario, abstract aliasing would occur in the implementation of P. Even more alarmingly, abstract aliasing could occur if the first parameter to P were any object from which lx is reachable, or if lx were reachable from a global variable visible to the implementation of P.

8 Discussion of variations

To smooth the exposition, we have presented our methodology in strict and simple terms. In this section, we sketch some possible variations.

First, the rule against returning pivots is more drastic than necessary. For example, it would be perfectly sound to forbid the return only of those pivots that were not among a procedure's in-parameters. The more liberal rule is sound because simple upward leaking occurs only when a procedure provides a caller with access to a pivot that was not accessible before the call. In fact, we expect a liberalization along these lines to be very convenient; for example, to avoid clashing with the convention that initialization methods return the object initialized. The reader may have wondered why the conjunct $virgin[lx]$ was present in the precondition of *Lexer.Init* : without the precondition, no implementation that uses the initialization convention will be able to establish the postcondition (3), $\neg pivot[\mathbf{result}]$. By liberalizing the rule about returning pivots, this restrictive precondition could be omitted.

Another way to liberalize the rule against returning pivots would be to have a dual to the **captures into** annotation, which would allow returning a pivot provided that the specification states explicitly what is being returned. For example, if x is an out-parameter or a global variable, the annotation

$$\textbf{returns}\ E\ \textbf{as}\ x$$

could cause the conjunct $\neg pivot'[x]$ to be omitted from the postcondition, and in its place add the conjunct

$$\textbf{ensures}\ x' = E \qquad .$$

This liberalization is sound for the same reason as the other liberalization is sound: returning a pivot does not cause abstract aliasing if the associated pivot field is in scope in the caller. We found several procedures in the NI2 indexing library for which **returns as**, or something like it, would be useful.

Secondly, it would be perfectly sound to weaken the precondition in the desugaring of **captures** o **into** $b[t]$ from **requires** $virgin[o]$ to **requires** $\neg plenary[o]$. However, we doubt that this would be convenient in practice, since, for one thing, the precondition of virginity is generally needed to prove that an initialization of a pivot field preserves the disjoint ranges requirement.

Thirdly, one can imagine cases where a procedure captures a parameter into a pivot field, but where it is inconvenient to name the owner at the point of declaration of the procedure. This suggests a notation like

$$\textbf{captures}\ o\ \textbf{into}\ b$$

whose desugaring has the postcondition

$$\langle \exists t \ :: \ b'[t] = o \rangle \qquad .$$

We have encountered one example program in which this notation might be useful.

9 Related work

The central problem addressed in this paper is to find an effective methodology for dealing with rep exposure that allows what is useful (passing a reader to an initializing method of a lexer) while preventing what is harmful (invisibly invalidating a lexer by closing the reader under it).

We learned the term rep exposure from the CLU community, but this community seems not to have developed any formal methodology for avoiding the problem [LG86].

Keeping track of references is a crucial difficulty in many kinds of static program analyses. There are several papers introducing annotation techniques related to ours. The Larch C Lint checking tool (LCLint) warns of places where a C program transfers mutable components across abstraction boundaries [Eva96]. The annotations for turning off inappropriate warnings from LCLint are similar, but less precise, than those used in our methodology. In the course of developing a tool to assist in the structural change of programs, Chan, Boyland, and Scherlis have encountered problems similar to the rep exposure problem, and have developed a number of annotations similar to ours [CBS98].

Our pivot and virgin attributes are similar to the *unique* and *free* modes of John Hogg's Islands paper [Hog91]. Almeida's Balloons paper also outlines a programming discipline for programming with object references [Alm97]. However, the Islands and Balloons papers are not directed toward the rep exposure problem and do not consider the connection between a reader and its enclosing lexer, and therefore do not provide a solution to the problem we are addressing in this paper.

The Flexible Alias Protection paper of Noble, Vitek, and Potter describes programming rules that prevent rep exposure [NVP98]. But the rules are too strict: they outlaw the lexer/reader program, because in this program a part of the representation of the lexer is referenced from outside the scope of the lexer's implementation.

Jones describes simple formal rules for avoiding *interference*, which is related to abstract aliasing, but his rules are too strict to be useful [Jon96].

The methodology described in this paper can be more precise because it is based on dependencies [LN98a]. Another of our contributions is that our paper is the first treatment of the rep exposure problem that contains no pictures.

10 Conclusions

It is difficult to design a programming discipline that prevents the unsound cases of rep exposure but allows useful and sound object-oriented programming styles. The theory of dependencies sheds new light on this problem. In particular:

- Rep exposure is usually defined as the transfer of mutable components of an abstract data type across the abstraction boundary for that type. But in fact, only those components that correspond to dynamic dependencies are problematical; components that correspond to static dependencies can be ignored. From this observation we have traced the source of unsoundness to the notion of abstract aliasing, precisely defined in terms of dependencies.

- If some abstraction $a[t]$ depends on the mutable value of some component $b[t]$ (that is, if **depends** $a[t]$ **on** $c[b[t]]$), a checker can detect the interference between updates to $a[t]$ and $c[b[t]]$ even if the details of the representation of a are not visible. All that is required is for the dependency to be in scope. This observation allows pivots to cross abstraction boundaries provided that the associated pivot field and dependency are in scope in the relevant interface.

Our analysis has led us to a conclusion that is shockingly different from the other approaches that we know. Instead of forbidding pivot values from ever crossing an abstraction boundary, we allow them to, provided that the pivot field is declared in the interface to the abstraction. Soundness is saved by declaring the dependency in the interface as well.

We have developed these observations into a set of rules for avoiding abstract aliasing and have described how to enforce them mechanically. The rules prevent many cases of abstract aliasing, but not all cases.

Acknowledgements

We are grateful to Martín Abadi, Monika Henzinger, Cynthia Hibbard, Sharon Perl, Jim Saxe, Raymie Stata, and Bill Weihl for comments on the presentation.

References

[Alm97] 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.

[BN91] Mark R. Brown and Greg Nelson. I/O streams: Abstract types, real programs. In Greg Nelson, editor, *Systems Programming with Modula-3*, Series in Innovative Technology, chapter 6, pages 130–169. Prentice-Hall, Englewood Cliffs, NJ, 1991.

[CBS98] Edwin C. Chan, John T. Boyland, and William L. Scherlis. Promises: Limited specifications for analysis and manipulation. In *Proceed-*

ings of the IEEE International Conference on Software Engineering (ICSE'98), pages 167–176. IEEE Computer Society, April 1998.

[Det96] David L. Detlefs. An overview of the Extended Static Checking system. In *Proceedings of The First Workshop on Formal Methods in Software Practice*, pages 1–9. ACM SIGSOFT, January 1996.

[DLNS98] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Research Report 159, Digital Equipment Corporation Systems Research Center, 1998. To appear.

[ESC] Extended Static Checking home page, Compaq Systems Research Center. On the Web at `http://www.research.digital.com/SRC/esc/Esc.html`.

[Eva96] David Evans. *LCLint User's Guide, Version 2.1a*, April 1996. Available from `http://www.sds.lcs.mit.edu/lclint/guide/index.html`.

[Hoa72] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1(4):271–281, 1972. Reprinted in C. A. R. Hoare and C. B. Jones, editors, *Essays in Computing Science*, Series in Computer Science, chapter 8. Prentice Hall International, 1989.

[Hog91] John Hogg. Islands: Aliasing protection in object-oriented languages. In Andreas Paepcke, editor, *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'91)*, pages 271–285. ACM Press, October 1991.

[Jon96] C. B. Jones. Accommodating interference in the formal design of concurrent object-based programs. *Formal Methods in System Design*, 8(2):105–122, March 1996.

[Lei95] K. Rustan M. Leino. *Toward Reliable Modular Programs*. PhD thesis, California Institute of Technology, 1995. Available as Technical Report Caltech-CS-TR-95-03.

[LG86] Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development*. MIT Electrical Engineering and Computer Science Series. MIT Press, 1986.

[LN98a] K. Rustan M. Leino and Greg Nelson. Abstraction and specification revisited. Internal manuscript KRML 71, Digital Equipment Corporation Systems Research Center, 1998. To appear as a SRC Research Report 160.

[LN98b] K. Rustan M. Leino and Greg Nelson. An Extended Static Checker for Modula-3. In Kai Koskimies, editor, *Compiler Construction: 7th International Conference, CC'98*, volume 1383 of *Lecture Notes in Computer Science*, pages 302–305. Springer, April 1998.

[LS97] K. Rustan M. Leino and Raymie Stata. Virginity: A contribution to the specification of object-oriented software. Technical Note 1997-001, Digital Equipment Corporation Systems Research Center, April 1997.

[NVP98] 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.