Pointwise dependencies in data abstraction

K. Rustan M. Leino and Greg Nelson
20 April 1995
Digital's Systems Research Center
130 Lytton Ave., Palo Alto, CA 94301, U.S.A.
{rustan,gnelson}@pa.dec.com

We extend the work in [0] to *pointwise dependencies*. We assume some familiarity with the issues in Part III of [0], but outline the relevant ingredients as a reminder.

0 A pointless world

In this section, we provide a quick refresher of the setting.

A program consists of a set of declarations, some of which declare *variables*. A scope is a subset of a program, in which there is a declaration for every identifier that is mentioned. If a declaration is in scope, we say it is *visible*.

Variables come in two flavors: *abstract* and *concrete* (also known as *specification* and *program* variables). An abstract variable a is declared by

$$\mathbf{spec\ var}\ a \quad ,$$

and a program variable c is declared by

$$\mathbf{var}\ c \quad .$$

Both flavors of variables can appear in specifications, but only program variables are allowed in executable code (like assignment statements).

Being abstract, a specification variable is given a *representation* in terms of (more) concrete variables. For example,

$$\mathbf{rep}\ a\ \mathbf{is}\ a = c^2 \tag{0}$$

states that abstract variable a is an abstract representation of c^2 . The right-hand side (after the $\mathbf{is}$) is a predicate that describes how to eliminate a . We restrict our attention to predicates that determine a uniquely from the other variables.

In the presence of many scopes, the predicate in a $\mathbf{rep}$ clause must mention only *dependencies* of the abstract variable. A dependency of a on c is declared by

$$\mathbf{depends}\ a\ \mathbf{on}\ c \quad .$$

The set of dependencies —the *downward closure*— of a variable is given by the reflexive and transitive relation $Resolve$.

An abstract variable can be mentioned in predicates. Such predicates undergo *functionalization* before they are interpreted. A variable a is functionalized into a function $f{\star}a$ whose parameters are the list of symbols in $Resolve(a)$. (The symbol $\star$ in $f{\star}a$ is just part of the name of the function.) For example, given declarations

$$\mathbf{spec\ var}\ a \ ;\quad \mathbf{var}\ c \ ;\quad \mathbf{depends}\ a\ \mathbf{on}\ c \quad ,$$

any occurrence of a in a predicate is functionalized, written $F(a)$, into $f{\star}a(a, c)$.

Without a with a associated $\mathbf{rep}$ clause in scope, a function $f{\star}a$ is treated as an uninterpreted function. Having a $\mathbf{rep}$ clause in scope provides an interpretation of the function. In particular, the declaration

$$\mathbf{rep}\ a\ \mathbf{is}\ P$$

gives rise to the axiom

$$\langle \forall\, r \triangleright F(P) \rangle \qquad ,$$

where r denotes the list of variables in $Resolve(a)$. For example, declaration (0) gives rise to the axiom

$$\langle \forall\, a, c \triangleright f{\star}a(a, c) = c^2 \rangle \qquad .$$

Note that, despite its signature, $f{\star}a$ is not a function of a. This is no accident; the a in the signature is called a *residue*, and in the absence of a visible $\mathbf{rep}$ clause, residues are essential for sound modular verification (details are found in [0]).

A procedure specification is given in the Larch-style

$$\mathbf{modifies}\ w\ \mathbf{requires}\ Pre\ \mathbf{ensures}\ Post \qquad ,$$

where w, called the *frame*, is a list of variables, and Pre and $Post$, the *pre*- and *postconditions*, respectively, are predicates. The specification states that, started in a state satisfying Pre, the procedure establishes $Post$, having modified the values of only those variables listed in w. When interpreting such a specification, w is resolved to its downward closure, and then Pre and $Post$ are functionalized.

In order to guarantee soundness of modular verification, there are two restrictions on the placement of $\mathbf{depends}$ clauses.

Visibility requirement. If a depends on c, then this dependency must be visible anywhere both a and c are.

Authenticity requirement. If a depends on c, then a is visible anywhere c is.

Together, the requirements essentially boil down to that $\mathbf{depends}\ a\ \mathbf{on}\ c$ is declared "near" c.

Given the above, Chapter 12 of [0] proves the soundness of modular verification.

1 Pointwise dependencies

We are now interested in one more attribute of variables: we allow them to be declared to be *maps*. To show that a variable is a map, one declares it with an appended " $: \mathbf{map}$ ". (Note. Other than knowing that some variables are maps, we are not concerned, at this time, with the type of a variable.)

Two operations are defined on maps, $select$ and $store$. A map can be thought of as an array (or as a function). For a map a,

$$select(a, t)$$

selects element t of a. As a matter of convenience, we abbreviate this expression by $a[t]$, in which " t " is called the *subscript*. The value of

$$store(a, t, x)$$

is the same as the value of a, except that selecting element t of the former yields x. Thus, we have the familiar

$$select(store(a,t,x),t) \;=\; x \qquad .$$

We require that variables declared as maps are used only via $select$ and $store$. Furthermore, we assume that the only use of $store$ is in commands of the form

$$a := store(a,r,x) \qquad ,$$

which is commonly abbreviated

$$a[r] := x$$

(thus alleviating the programmer from ever having to write "$store$" directly). Since only program variables are allowed in commands, $store$ will never be applied to maps that are specification variables.

Maps invite programmers to write a $\mathbf{rep}$ clause describing the representation for a map at each point (in fact, this is the only way we allow the representation of a map to be given). For example, given

$$\mathbf{spec\ var}\ a : \mathbf{map} \;\; ; \;\; \mathbf{var}\ c : \mathbf{map} \;\; ; \;\; \mathbf{var}\ d \qquad ,$$

we might write

$$\mathbf{rep}\ a[t]\ \mathbf{is}\ a[t] = c[t]^2 + d \qquad .$$

Here, "$\mathbf{rep}\ a[t]$" specifies that a's representation is given pointwise, and that t is a dummy variable that can be used in the representation predicate. As before, the predicate must mention only dependencies of the abstract variable, but we now need to say what a dependency of a map is.

All dependencies of a map are given from a particular point, as if there were a separate downward closure for each point of the map. A common form of such a dependency is

$$\mathbf{depends}\ a[t]\ \mathbf{on}\ c[t] \qquad .$$

As with pointwise $\mathbf{rep}$ clauses, the first occurrence of "$[t]$" declares a dummy that names the point and whose scope is what succeeds "$\mathbf{on}$". Sometimes, that dummy is not used, as is

$$\mathbf{depends}\ a[t]\ \mathbf{on}\ d \qquad .$$

These two $\mathbf{depends}$ clauses declare the downward closure of $a[t]$ to be $\{c[t], d\}$, written

$$Resolve(a,t) \;=\; \{c[t], d\} \qquad ,$$

or with the alternative syntax

$$Resolve(a[t]) \;=\; \{c[t], d\} \qquad .$$

Remember that t is but a dummy; so, for example, $Resolve(a[x])$ means

$$\{c[x], d\} \qquad .$$

Now we are able to state the following rule: The variables (and $select$'s on map variables) that are allowed in the predicate of $\mathbf{rep}\ a[t]\ \mathbf{is}\ \ldots$ are the elements of $Resolve(a[t])$.

2 Pointwise functionalization

We now describe how F works for maps. We continue, throughout the rest of this note, to use the example

$$\begin{array}{l}\mathbf{spec\ var}\ a : \mathbf{map}\ ;\quad \mathbf{var}\ c : \mathbf{map}\ ;\quad \mathbf{var}\ d\ ;\\ \mathbf{depends}\ a[t]\ \mathbf{on}\ c[t], d\ ;\\ \mathbf{rep}\ a[t]\ \mathbf{is}\ a[t] = c[t]^2 + d\qquad .\end{array}$$

F distributes over connectives. Thus,

$$F(x+y) \ = \ F(x) + F(y) \qquad .$$

More generally, F distributes over functions, as in

$$F(add(x,y)) \ = \ add(F(x), F(y)) \qquad .$$

We could apply the same idea to $select$ on specification variables. That is,

$$F(select(a,t)) \ = \ select(F(a), F(t)) \qquad ,$$

alternatively written as

$$F(a[t]) \ = \ F(a)[F(t)] \qquad .$$

However, this does not seem attractive, because then $F(a)$ needs to denote a map while the $\mathbf{rep}$ clause is written pointwise. Moreover, consider a command that modifies $c[j]$, where $j \neq t$. Does this have an effect on $F(a[t])$, *i.e.*, on

$$f{\star}a(a,c,d)[F(t)] \qquad ?$$

This is not clear, because it is not implied that the functionalized expression is a function of c only at element t.

Instead, we specialize F when its argument is a $select$ expression.

$$F(a[t]) \ = \ f{\star}a(a[F(t)], c[F(t)], d)$$

This closely resembles functionalization of non-map variables, where an abstract variable a is replaced by an application of function $f{\star}a$ whose arguments are $Resolve(a)$. Here, $a[t]$ is replaced by a function $f{\star}a$ whose arguments are given as $Resolve(a[F(t)])$.

Note that this describes how to functionalize $a[t]$ for *any* expression t, because t is functionalized in the result. Note, also, that it solves the problem with the above proposal, because here c occurs only in the form $c[F(t)]$, which makes explicit of which point of c the expression is a function.

The axioms generated by $\mathbf{rep}$ clauses of maps need attention. Applying the non-map axiom-generating process to maps yields

$$\langle\, \forall\, a[t], c[t], d \triangleright f{\star}a(a[t], c[t], d) \ = \ c[t]^2 + d \,\rangle$$

for our example. This is what we want, if only we interpret the list of dummies the "correctly". The dummy $a[t]$ is to be treated as *one* symbol, not two, and not as $select$ applied to two arguments. Hence, we prefer to write $a[t]$ as $a{\star}t$, producing the axiom

$$\langle\, \forall\, a{\star}t, c{\star}t, d \triangleright f{\star}a(a{\star}t, c{\star}t, d) \ = \ c{\star}t^2 + d \,\rangle \qquad . \tag{1}$$

(Recall that $\star$ is simply another character that can be used in identifiers. Thus, $c{\star}t^2$ means $(c{\star}t)^2$, not $c{\star}(t^2)$, which is but nonsense.)

3 Modifies clauses

We discuss how **modifies** clauses are interpreted. In the absence of specification variables, we think of

$$\mathbf{modifies}\ c[t]\ \mathbf{ensures}\ Q \quad ,$$

where c denotes a map (and t is some expression) and Q mentions only program variables (and thus $F(Q) = Q$), as syntactic sugar (see [0]) for

$$\mathbf{modifies}\ c\ \mathbf{ensures}\ Q \wedge \langle \forall j \mid j \neq t_0 \triangleright c_0[j] = c[j] \rangle \quad ,$$

where t_0 and c_0 denote the initial values of t and c, respectively. Let's call this sugar expansion A.

In Section 0, we defined

$$\mathbf{modifies}\ a\ \mathbf{ensures}\ Q$$

to mean

$$\mathbf{modifies}\ r\ \mathbf{ensures}\ Q \quad ,$$

where r is the list of elements in $Resolve(a)$. Let's call this expansion B. In the presence of specification variables, the pre- and postconditions are also later functionalized into

$$\mathbf{modifies}\ r\ \mathbf{ensures}\ F(Q) \quad .$$

Now we face a decision: How do we interpret the **modifies** clause in

$$\mathbf{modifies}\ a[t]\ \mathbf{ensures}\ Q \tag{2}$$

when a is a specification variable? Should we apply A first and then B (call it AB), or should we apply B first and then A (call it BA)?

Naturally, we first investigate whether or not there is a difference. We do BA first. Applying B to (2), we get

$$\mathbf{modifies}\ a[t], c[t], d\ \mathbf{ensures}\ Q \quad .$$

A subsequent application of A yields

$$\begin{array}{l}\mathbf{modifies}\ a, c, d \\ \mathbf{ensures}\ Q \wedge \langle \forall j \mid j \neq t_0 \triangleright a_0[j] = a[j] \rangle \wedge \langle \forall j \mid j \neq t_0 \triangleright c_0[j] = c[j] \rangle \quad .\end{array}$$

Lastly, functionalizing the postcondition gives us

$$\begin{array}{l}\mathbf{modifies}\ a, c, d \\ \mathbf{ensures}\ F(Q) \wedge \\ \qquad \langle \forall j \mid j \neq t_0 \triangleright f{\star}a(a_0[j], c_0[j], d_0) = f{\star}a(a[j], c[j], d) \rangle \wedge \\ \qquad \langle \forall j \mid j \neq t_0 \triangleright c_0[j] = c[j] \rangle \quad .\end{array}$$

Now for AB. Apply A to (2) and we get

$$\mathbf{modifies}\ a\ \mathbf{ensures}\ Q \wedge \langle \forall j \mid j \neq t_0 \triangleright a_0[j] = a[j] \rangle \quad .$$

How do we now apply B, since we have defined a way to resolve $select(a, t)$, but not just a? We approximate this by ignoring subscripts. Applying B then gets us

$$\begin{array}{l}\mathbf{modifies}\ a, c, d \\ \mathbf{ensures}\ Q \wedge \langle \forall j \mid j \neq t_0 \triangleright a_0[j] = a[j] \rangle \quad ,\end{array}$$

which, after functionalization, turns into

$$\mathbf{modifies}\ a, c, d$$
$$\mathbf{ensures}\ F(Q) \wedge \langle \forall j \mid j \neq t_0 \triangleright f{\star}a(a_0[j], c_0[j], d_0) \ = \ f{\star}a(a[j], c[j], d) \rangle \qquad .$$

The first thing to notice is that AB and BA do produce different results. The $\mathbf{modifies}$ clauses of both are the same, but the $\mathbf{ensures}$ clauses differ. In particular, the $\mathbf{ensures}$ clause that results from BA is stronger than the one resulting from AB. The extra conjunct is

$$\langle \forall j \mid j \neq t_0 \triangleright c_0[j] = c[j] \rangle \qquad .$$

Although this extra conjunct will usually hold —and, in fact, it may even be difficult to mechanically prove the $\mathbf{ensures}$ clause without first proving that conjunct—, we argue that we should not add this conjuct to the $\mathbf{ensures}$ clause.

Where this makes a difference is where c is visible but the $\mathbf{rep}$ clause for a is not —as would be the case, for example, if c were found in a friends interface (see [0]) and we're considering the scope of a friend—. Then, any calls to a procedure that modifies $a[j]$, for some j, can potentially alter the value of $c[k]$, for *any* k, not just $k = j$. However, any such modification must preserve the value of $a[k]$, that is, of $f{\star}a(a[k], c[k], d)$.

4 Modular verificiation

We have not yet redone the soundness proof of modular verification found in Chapter 12 of [0]. However, an quick inspection of the proof gives us hope that the proof will go through without substantial changes. A conclusive test is in the works...

For the proof, we need to adjust the visibility and authenticity requirements to accommodate for pointwise dependencies. We propose to subject pointwise $\mathbf{depends}$ clauses to follow the same rules as before, ignoring any subscripts. Thus, for example, the clause

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

should be placed near the declaration of c.

If our guess that the soundness proof will go through with only minor changes turns out to be correct, we will not be surprised, for another reason: What we have discussed in this note has been focused more on how things are proven in any one scope, as opposed to the differences in proofs due to limited scope. Let's be a little more specific.

The structure of the soundness proof, as pointed out in [0], is such that it "extends" a correctness proof conducted in a restricted scope to a correctness proof in the complete ("flat") scope. The soundness proof tells us nothing re the net worth of what is being proven. This note deals with what seems to be sensible interpretations of pointwise dependencies, predicates involving such, and $\mathbf{modifies}$ clauses listing such.

Stated slightly differently, we expect that whatever we would have chosen in the choices that we made in Sections 2 and 3, has no impact on the proof—we think these are choices that concern the net worth of what is proven and not the soundness of modular verification.

We expect addressing $\mathbf{depends}$ clauses of the form

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

to give us more trouble (see below).

5 Work to be done

We have yet to do the actual formal proof.

Our next step will be to accommodate object-orientedness. We can do this by letting the dummy t be part of a representation predicate. More specifically, we're interested in letting t be a pivot of a "case split". Additional syntax can be of help here. We can allow $\mathbf{rep}$ clauses and corresponding $\mathbf{depends}$ clauses that have the form

$$\mathbf{rep}\ a[t]\ \mathbf{where}\ Q(t)\ \mathbf{is}\ P \qquad ,$$

where Q is a predicate on t, and P is a predicate whose symbols are drawn from $Resolve(a[t])$ (this does not include t itself). For each symbol a, the different Q's must be mutually exclusive (except $\mathbf{depends}$ clauses, where implication everywhere is also okay). Furthermore, there needs to be a way to statically determine, as if from the syntax or static types of the language, which Q's hold for a given t, so as to get $Resolve$ to do the right thing. Having this extension lets us substitute $\mathbf{istype}(t, T)$ for $Q(t)$, where T is some subtype of the index type of a.

The big step after that is to explore the extra requirements needed to make

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

sound. From its syntax alone, we can tell what we need to alter the visibility and authenticity requirements, since those requirements currently mention only two variables, a and c, and this declaration mentions three, a, b, and c.

References

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