Data Abstraction in Multiple Scopes

K. Rustan M. Leino

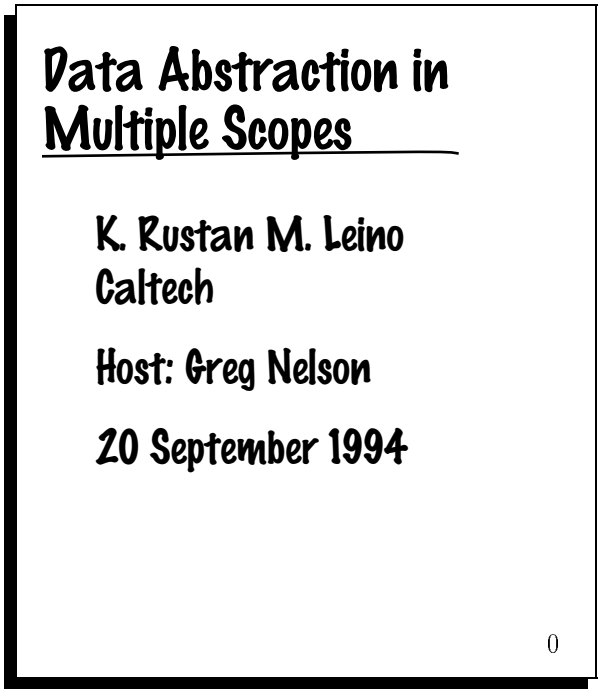
10 October 1994

This notes provides a transcription of my end-of-summer intern talk at Digital's Systems Research Center, 1994. The transcription provides some words that I said, some words I think I said, and some words I wish I had said.

On suggestion from Allan Heydon, I have included PostScript renderings of my slides in this note. This instills a nice sing-along flavor.

Omitted from this note is the nice introduction by Greg Nelson.

• ○ •



Ladies and gentlemen, I'm Rustan Leino. I'm going to tell you about data abstraction in multiple scopes. This is work I did as a summer intern here at SRC, where I worked together with Greg Nelson and others on the Extended Static Checking project.

Extended Static Checking

Idea: Automatically verify, at compile-time, that no execution of a program causes a checked run-time error.

How: Allow programmers to annotate their programs with specifications.

Challenges:

- Automatic theorem proving
- Verification condition generation
- Specification problems

1

The idea of extended static checking is to provide a limited form of automatic program verification. The goal is to prove the absence of checked run-time errors at compile-time, that is, statically. The vision here is that compilers will, in the future, include an extended static checker, just like compilers include type checkers today.

Verifying a program automatically is, in general, not possible, because the halting problem can be reduced to that task. To tackle this, programmers provide specifications of their programs. Having a tool that encourages programmers to write down specifications is a win by itself. Being able to prove a program correct with respect to these specifications is another leap for program technologies.

The area of extended static checking presents us with several challenges. One of these is automatic theorem proving. Here, issues include, Can we verify the programs at all?, Can we verify efficiently?, and possibly also, Can we verify them without intervention from the programmer?

Another challenge is generating the conditions to be verified. This involves translating specifications and programs into formulas.

The challenge that I focused on this summer is that of writing specifications. This is an area that I, before this summer, thought was solved. That turns out not to be the case.

Specification problems

- Data hiding in interfaces/modules requires data abstraction
- Data abstraction
  - has been around [Hoare72]
  - has been used in the SRC Extended Static Checker (ESC)
  - previously not sound in ESC !
- Modular verification

  Verify that an implementation meets its specification, using only the information from the implementation's scope

2

Let me describe to you the setting in which we find specification problems.

The programming language we use, Modula-3, features *modules* and *interfaces*. A procedure is declared in an interface, and its implementation is given in a module. This hides the private data of the module from the clients of the interface. The implementation of a procedure declared in an interface may have an effect on the private data seen in the module. We use *data abstraction* to combat this problem: the interface describes an abstract view of the behavior of the procedure; the module provides

the implementation and prescribes the relation between the abstract view and the concrete one.

Data abstraction, or more precisely, data refinement, has been around since [Hoare 1972], and much work has appeared in this area since. Data abstraction has also been used in the SRC Extended Static Checker. However, before this summer, this checker was not sound in its treatment of data abstraction.

Extended static checking interests us in *modular verification*. That means that one should be able to verify an implementation given only its module and its imported interfaces. Having the entire program in view at one time simplifies verification, but is unreasonable to require, because, for example, then a library could not be verified until it were linked with a complete program.

Problem

Arises in presence of

- data abstraction
- friends interfaces
- modular verification

My contributions

- Invented solution to this problem
- Constructed formal proof of soundness
- Identified further problems

3

The problem arises in the presence of three things: data abstraction, friends interfaces —which I will mention later—, and modular verification.

In the area of extended static checking, I made three main contributions this summer. Firstly, I invented a solution to the above problem, a solution I will describe in this talk. Secondly, for this solution, I constructed a formal proof of soundness, which I will not go through in this talk. I will, however, explain more precisely what the theorem being proven is. Finally, I identified some further problems. While finding solutions is difficult, finding solutions to problems that you don't know how to state is even more difficult. That's why having identified these further problems is an indication of progress.

In the rest of the talk, I will present a real problem that was found when trying to verify the readers and writers library here at SRC. Once I've done that, we can discuss the solution and the others things alluded to.

Procedure specifications

```
proc p() =
    modifies frame
    requires pre
    ensures post
```

Why a modifies clause?

4

Since we are dealing with specifications, I need to show you their format. Using a Larch-like notation for procedure specifications [Larch], a procedure `p` is specified in three parts: a *precondition* given by a `REQUIRES` clause, a *postcondition* given by an `ENSURES` clause, and a *frame* given by a `MODIFIES` clause. The frame specifies what variables the procedure is allowed to modify in order to establish the postcondition from the precondition. Without `MODIFIES` clauses, a procedure would be able to modify anything, just as long as the postcondition is established. That would not give us specifications that are strong enough.

Next, I will present the basic problem to you. In order to do so, I need to show some code — four slides thereof to be exact. Please bear with me as I go through these.

```
unit Wr;
type T;
spec var target: T → SEQ[CHAR];

proc PutChar ( wr: T;  ch: CHAR ) =
   modifies target[wr]
   ensures target[wr] = target_0[wr] & ch
```

5

Here's the first slide of code. It shows the interface of a *writer* class. A writer is an output stream. Examples of writers are *file writer*s, which write their output stream to a file in a file system, and *text writers*, which write their output to a text string in memory.

I said this was an interface, but the code reads `UNIT`. This is because what I'm doing need not distinguish between interfaces and modules, as in Modula-3. Instead, I will speak of either as a *unit*.

Unit `Wr` introduces a new type, `T`. It also declares a *specification variable*, `target`. Unlike a *program variable*, a specification variable does not occupy any space at run-time. Rather, its only purpose is to abstractly describe the behavior of some procedures, eliminating the need to mention the details of the particular implementation.

Specification variable `target` is of type `T` → `SEQ[CHAR]`. That is, `target` is a *map* from objects of type `T` to sequences of characters. Those familiar with Modula-3 may think of `SEQ[CHAR]` as Modula-3's `TEXT` type, but the exact nature of `SEQ[CHAR]` is not central to the discussion.

It may seem funny that the type of `target` is a map. How does a programmer create a function that is assigned to this variable? Instead of thinking of `target` as a map, you may think of it as a *ghost field* of the object type `T`. When `target` is applied to a particular object of type `T`, the result is a sequence of characters. I will use the notation `target[t]` to denote this field for an object `t`. A more common programming notation is `t.target`. Since the former notation is closer to the way these fields are modeled in the theory, I will stick to it.

Unit `Wr` also shows the specification of a procedure, `PutChar`, which takes as parameters a writer and a character. The procedure modifies the target of the writer to ensure that the target, upon termination of the procedure, equals the initial target extended with the given character.

```
unit WrFriends;
import Wr;
var buff: Wr.T → SEQ[CHAR];

(* Wr.target[wr] =
     flushed characters & buff[wr] *)
```

6

Here's the second slide of code. It shows a unit `WrFriends`, which is an example of a *friends interface*, to which I alluded earlier. It gets that name from the fact that the unit is intended for import by other units with a close tie to the implementation of writers — a tie only "good friends" are thought to have. To those familiar with the SRC Modula-3 library, `WrFriends` is the `WrClass` interface found there.

Unit `WrFriends` *imports* unit `Wr` to make the declarations in `Wr` visible in `WrFriends`. The import relation is transitive — that is, by importing `Wr`, `WrFriends` also imports all units imported by `Wr` (if there were any).

`WrFriends` declares a variable `buff`. Similar to `target` from the previous slide, `buff` can be thought of as a *field* of `Wr.T` objects. Note that we prefix type `T` from unit `Wr` with "`Wr.`" since it is imported from another unit. Note also that `buff` is not a specification variable; hence, `buff` is a field that will be present at run-time.

The unit ends with a comment describing to programmers the intended usage of field `buff`. The idea is the following. The target of each writer has a *flushed* portion and a *buffered* portion. Different writers store their flushed portion in different ways. For example, a file writer keeps the flushed portion on disk, whereas a text writer keeps it in a string in memory.

So that all different kinds of writers can make use of the same buffering mechanism, `buff` is introduced. Hence, all writers have this `buff` field. Procedure `PutChar`, then, simply adds the given character to the end of `buff` for the given writer, and calls the particular writer subclass to perform a flush when `buff` becomes too large.

In a sense, the solution to be presented is a formalization of this comment.

```
unit TextWr;
import Wr;
type T <: Wr.T;
proc Init(wr: T) =
  modifies Wr.target[wr]
  ensures Wr.target[wr] = "";
proc Target(wr: T): SEQ[CHAR] =
  ensures RES = Wr.target[wr]
```

7

This code slide shows a particular class of writers: text writers. The unit declares a type `T` as a

subtype of `Wr.T`. That is, objects of type `TextWr.T` make up some of the objects of type `Wr.T`.

Unit `TextWr` also declares two procedures, `Init` and `Target`. `Init` clears the target of a text writer. Its specification states that the target of the given text writer is modified to ensure that it equals the empty string upon termination.

Procedure `Target` returns the target for a given text writer. It does not modify anything in the process. `RES` refers to the result value returned by the procedure.

```
unit TextWrImpl;
import Wr, WrFriends, TextWr;
var flushed: TextWr.T → SEQ [CHAR];
rep Wr.target [wr: TextWr.T] is
  Wr.target [wr] =
  flushed [wr] & WrFriends.buff [wr];

impl Init ( wr: TextWr.T ) =
  flushed [wr] := "";
  WrFriends.buff [wr] := "";
impl Target ( wr: TextWr.T ) =
  return flushed [wr] & WrFriends.buff [wr]
```

8

The fourth and final slide of code shows the implementation of text writers. In a language like Modula-3, this unit would be a module. However, since we do not distinguish between modules and interfaces, we simply call this unit `TextWrImpl`.

This unit imports all of the previous units that we've seen. It declares a variable `flushed`, which will contain the flushed portion of text writers. Hence, the *representation* of `target` for text writers can be given, as stated by the `REP` clause. More precisely, `target` of a text writer is the concatenation of `flushed` and `buff` for that writer.

The notation used in the `REP` clause suggests that any relation can be used to describe the specification variable `target`. Although we don't think using such an abstraction relation would cause unmanageable problems, all abstraction relations that we have ever used in practice have been functional. The SRC Extended Static Checker assumes these to be functional, but our formal proof [KRML 42] does not.

The `TextWrImpl` unit also gives the implementation of procedures `Init` and `Target`. `Init`, which is supposed to set `target` to the empty string, sets both `flushed` and `buff` to the empty string for that writer, the concatenation of which is the empty string.

Procedure `Target`, which is supposed to return `target` of the given text writer, simply returns the concatenation of `flushed` and `buff` for that writer.

Question

Why can Init, which is specified to only modify Wr.target, modify flushed and WrFriends.buff?

9

Basic Problem

```
unit FaultyClient;
import Wr, WrFriends, TextWr;
    :
  TextWr.Init (wr); {Wr.target[wr] = "" }
  WrFriends.buff[wr] := ...;
  if TextWr.Target (wr) ≠ "" then Wrong end
```

What should prevent this program from validating?

10

Now that I've shown you the code for this programming example, I'm able to raise a question. Why is it that procedure `Init`, which was specified to only modify `Wr.target`, is allowed to modify variables `flushed` and `WrFriends.buff`?

[Time for contemplation and interaction with audience.]

That's it. We think it's okay for `Init` to modify these variables because they are part of `target`'s representation. Having decided that, we are ready to show the basic problem.

Consider the following client unit, which imports `Wr`, `WrFriends`, and `TextWr` — that is, all the units seen so far except the text writer implementation.

For some text writer `wr`, it calls `TextWr.Init`. After that call, we can conclude that the target of this writer is the empty string, as is shown as an annotation on the slide. Although the representation of `target` is not visible in this scope, you and I know that this means `flushed` and `buff` each equals the empty string.

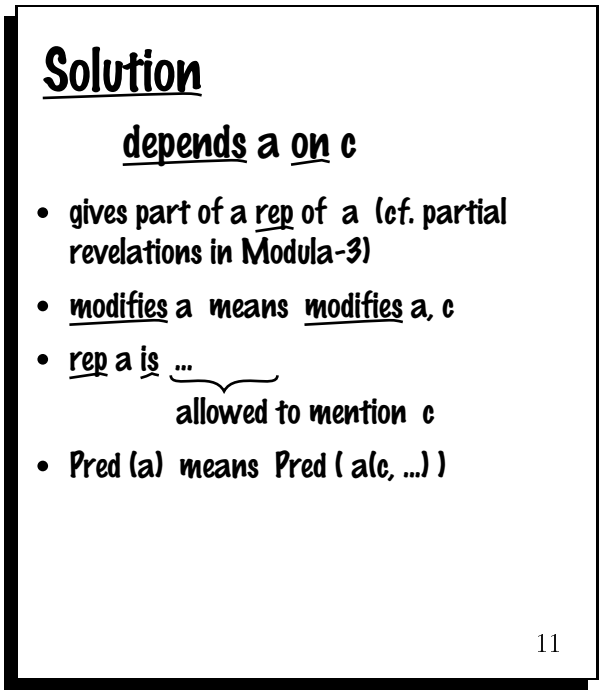
The next statement mucks with this writer's `buff` field. Hence, this statement actually affects `target`. But this fact goes unnoticed to the verification process, to which `target`'s `REP` clause is not visible. Therefore, the verification process treats the update of `buff` as having no effect on `target`.

The last line of code retrieves the value of `target[wr]`, via a call to `Target`, and compares this value to the empty string. If the update of

`buff` has no effect on `target`, then the two will be equal, and the branch that "goes wrong" is not taken.

However, you and I, knowing the details of the representation of `target` know that this code *will* go wrong at run-time. The question thus is, What should prevent this program from validating?

[Time for some deep pondering. Silence prevails.]



Now that we understand what the problem is, let's move on to its solution.

The solution is to introduce a new specification construct called `DEPENDS`. This will allow dependencies between variables — that one variable is part of the representation of another — to be revealed. The clause

```
DEPENDS a ON c
```

reveals that specification variable `a` may be represented in terms of variable `c`.

More precisely, the `DEPENDS` construct allows a programmer to give *part* of the representation of a specification variable. `DEPENDS` doesn't state what the representation *is*, but reveals a variable on which it depends.

Modula-3 programmers familiar with partially opaque types may spot a similarity between partial and full type revelations, and `DEPENDS` and `REP` clauses.

When a `MODIFIES` clause lists a specification variable, that is treated as a shorthand for also listing the variables on which the specification variable depends. In other words, the actual frame is the reflexive transitive closure of the frame clause given by the programmer. We often call this closure the *downward* closure, to indicate that the closure goes towards the more concrete representation.

We require that all variables on which a specification variable's representation depends be given in `DEPENDS` clauses. Thus, in order to mention a variable `c` in the `REP` of a specification variable `a`,

```
DEPENDS a ON c
```

must be visible in the scope in which the `REP` appears.

Finally, a predicate `Pred` that mentions a specification variable `a` is interpreted as the same predicate with `a` replaced by a function. The function, which in the slide I also call `a`, is a function of its dependencies. Here, there is an important detail, coined *residues*, that plays a rôle, but I won't go into that in this talk.

Let's take a look at how `DEPENDS` solves the problem I introduced earlier. To unit `WrFriends`, we add

```
DEPENDS Wr.target[t: Wr.T] ON buff[t] .
```

This states that, for every writer `t`, `target[t]` depends on `buff[t]`.

Similarly, in unit `TextWrImpl`, we add the line

```
DEPENDS Wr.target[t: TextWr.T] ON
    flushed[t] .
```

It discloses that, for every *text writer* `t`, `target[t]` depends on `flushed[t]`.

Given these dependencies, the `REP` clause in `TextWrImpl` is allowed to be stated. Furthermore, the faulty client's update of `buff` is now reflected as a change of `target` — a change whose precise character is here unknown. Hence, the faulty client will no longer verify.

Question

Under what restrictions can depends be used?

Visibility Requirement

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

12

Let us consider some restrictions that apply in the use of `DEPENDS` clauses. Certainly, there must be *some* restriction, because otherwise all `DEPENDS` clauses could be written in some distant unit that almost never is imported, and then these clauses would do no good given our goal of modular verification.

To withstand this problem, we will require that the dependency between two variables be visible wherever both of those two variables are. We call this rule the *visibility requirement*.

```
import Wr, WrFriends;
spec var a;
depends a on WrFriends.buff;
   ⋮
{a = A} Wr.PutChar (wr, ch) {a = A ?}
```

Authenticity Requirement

If a depends on c, then a must be visible anywhere c is.

13

Let us consider another program unit. This one declares a specification variable called `a`, and reveals that `a` depends on `buff`. For simplicity, I have left subscripts off.

Now, consider the code that follows. Initially, it is known that `a` takes some particular value, call it `A`. Then, `Wr.PutChar` is called. The `MODIFIES` clause of `PutChar` only lists `target`. Since `WrFriends` is imported into the current scope, the dependency of `target` on `buff` is known. Hence, the downward closure of `target` is calculated to be `target` and `buff`.

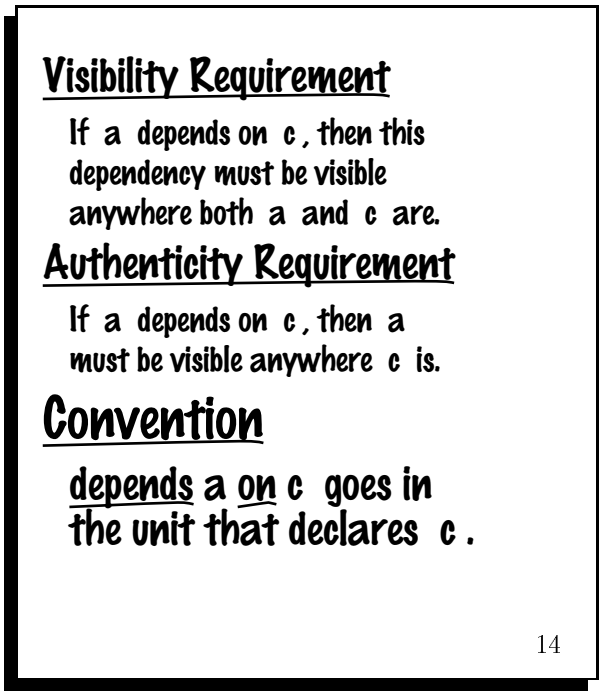
From this, the verification process concludes that `PutChar` has some effect on `target` and `buff`, but on no other variable. Consequently, `a` appears unmodified by the call to `PutChar`.

You and I know that `PutChar` actually has some effect on `buff`, and therefore that `a`, which depends on `buff`, is potentially modified. Hence, we would not want this piece of code to validate.

So, what went wrong? The verification of this

unit assumes that `PutChar` has no effect on `a`. Thus, the implementation of `PutChar` must be ensured to have no effect on `a`. But to guarantee this, `a` must be in scope during the verification of `PutChar`. That leads us to our next requirement: If a specification variable `a` depends on a variable `c`, then `a` must be visible everywhere `c` is.

We call this rule the *authenticity requirement*, because it prevents "unauthentic" abstract representations of concrete variables. Stated in terms of the example, it prevents the unauthentic abstract representation `a` of the concrete variable `buff`.



Some of you may be wondering, How can the visibility and authenticity requirements be enforced? The answer is simple: By following a simple convention, both of the requirements follow.

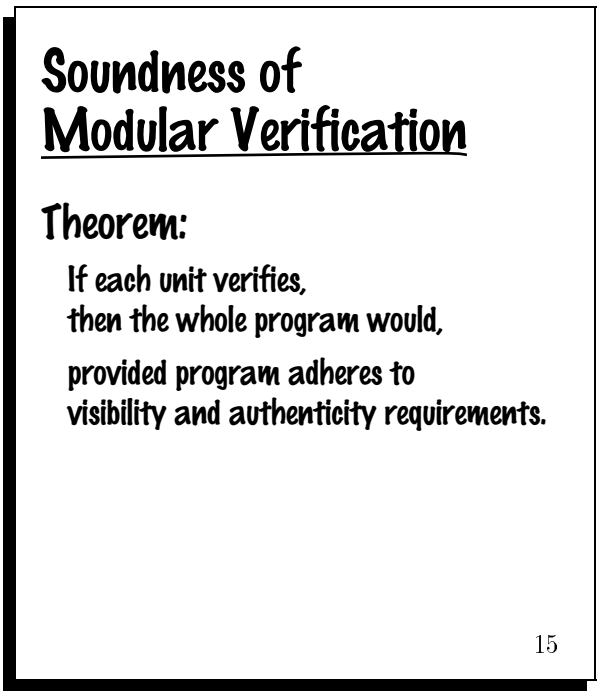
The convention states that a `DEPENDS` clause

```
DEPENDS a ON c
```

should be placed in the unit that declares `c`.

Note first that this clause can only be written down in scopes in which both `a` and `c` are visible; otherwise, the compiler will complain with an `undefined identifier` error message.

The authenticity requirement dictates that the unit that declares `a` be imported from the unit that declares `c` — otherwise it would be possible for a unit to import `c`'s unit to make `c` visible, but to leave `a` not visible. The visibility requirement is then upheld by placing the `DEPENDS` clause in the unit that declares `c`.



I have now shown a couple of restrictions on the use of `DEPENDS`. Naturally, we wonder whether these two requirements are enough. To answer that question, we're interested in proving the *soundness* of modular verification.

Loosely speaking, this means that if each unit verifies, then the whole program would verify, provided the program satisfies the two stated require-

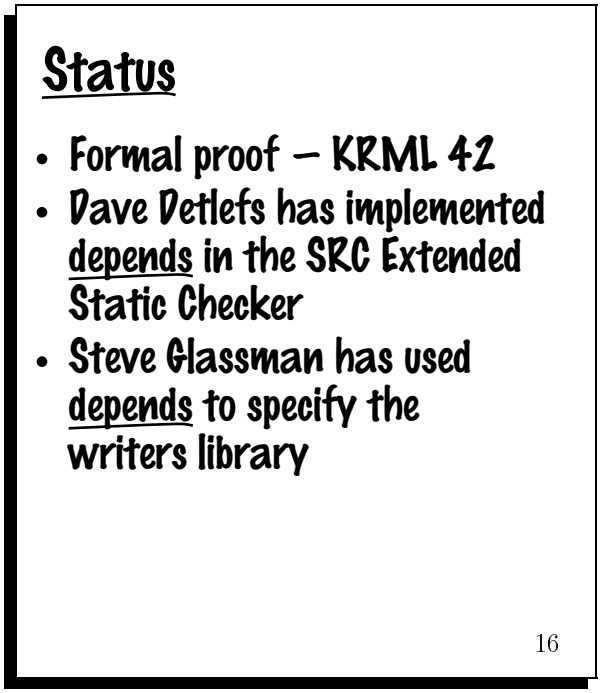
ments. Such a theorem allows the verification of a procedure implementation with respect to its specification to be performed in the scope of the unit in which the implementation occurs, *i.e.*, using only the information from that unit and its imports.

I have a formal proof of this theorem, so it is indeed a theorem.

A couple of remarks are in order. Firstly, although I won't show the formal proof here, an interesting property of the proof is that the visibility and authenticity requirements appear in distinct places. For that reason alone, it is worthwhile stating the requirements separately, even though when teaching the rules to a programmer, just stating the convention is probably good enough.

Secondly, since soundness holds, you may wonder about completeness. Completeness would mean that if the program could be verified given *all* its units at once, then each unit would verify by itself, too. We have no hope of achieving this. For example, in the faulty client example I showed, if the line that mucks with `buff` actually sets `buff` to the empty string, then `target` remains unchanged. But the only way to determine that is to have the `REP` in scope, and requiring that violates the essence of data hiding.

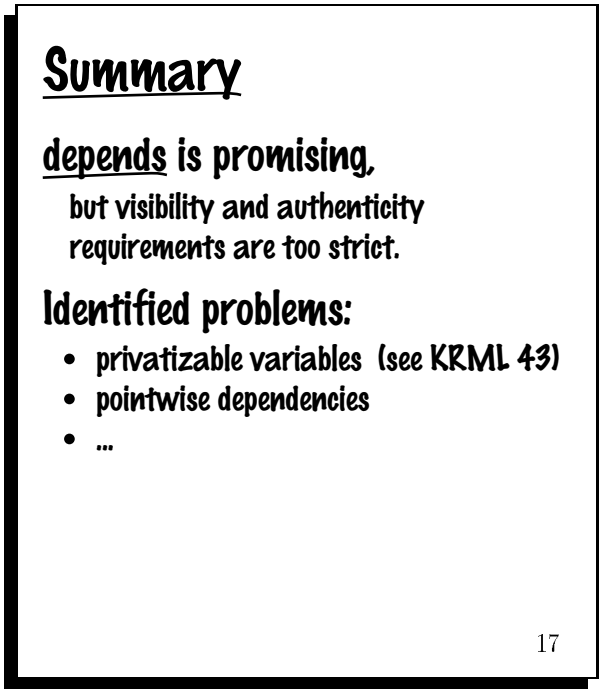
So, it is not completeness in which we're interested. Instead, we're interested in *adequacy*. That is, we want to be able to specify and verify things we care about. More about that later.



The current status of `DEPENDS` is that its soundness in modular verification has a formal proof, recorded in [KRML 42].

Dave Detlefs has incorporated `DEPENDS` into the SRC Extended Static Checker.

Steve Glassman has used `DEPENDS` in trying to specify the writers library.



In summary, `DEPENDS` is promising, but it's not the end of the story. There are more problems to solve, because the visibility and authenticity requirements are too strong for the solution to be adequate.

The solution does appear adequate as long as there is only one level of *specification* variables, *i.e.*, so long as no specification variable depends on another specification variable. However, when one module is implemented in terms of another, this is usually not good enough.

For example, consider a lookup table interface, which, to hide the details of its implementation, declares a specification variable `contents`. Lookup tables may then later be implemented using linked lists, trees, hash tables, or anything else.

If the implementation of some class of writers uses such lookup tables, then `Wr.target` for those writers will depend on `contents`. From the presented requirements, the dependency of `target` on `contents` needs to be declared in the lookup table interface. But that seems totally unreasonable, since the lookup table implementor cannot anticipate all lookup table clients. We categorize this kind of problem under what I have called *privatizable variables*, as seen in [KRML 43].

In order to understand privatizable variables, we think we first need to understand *pointwise dependencies*. These are dependencies of, say, `target[wr]` — that is, `target` for a particular writer `wr` — on `buff[wr]` — that is, `buff` for that same writer. The formal proof to which I alluded does not deal with pointwise dependencies. Rather, it only treats dependencies from all of `target` to all of `buff`. We believe it is but a clerical task to extend the proof to also handle pointwise dependencies, but I have not yet done this.

I will continue this work back at Caltech. Although there are more things we need to understand, the invention of `DEPENDS` and the identification of some outstanding problems provide hope of achieving modular verification of object-oriented programs, and, for that matter, hope of achieving extended static checking.

References

[Hoare 1972] C.A.R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1(4):271–81, 1972.

[Larch] J.V. Guttag and J.J. Horning and J.M. Wing. The Larch family of specification languages. *IEEE Software*, 2(5):24-36, 1985.

[KRML 42] K.R.M. Leino and G. Nelson. A formal proof of KRML 40. *KRML 42*, September 1994.

[KRML 43] K.R.M. Leino. Specifications and private data: A call for privatizable variables. *KRML 43*, September 1994.