Proving pointer programs in Hoare Logic

Richard Bornat

Department of Computer Science, Queen Mary and Westfield College, University of London, LONDON E1 4NS, UK
`richard@dcs.qmw.ac.uk http://www.dcs.qmw.ac.uk/~richard`

Abstract. It is possible, but difficult, to reason in Hoare logic about programs which address and modify data structures defined by pointers. The challenge is to approach the simplicity of Hoare logic's treatment of variable assignment, where substitution affects only relevant assertion formulæ. The axiom of assignment to object components treats each component name as a pointer-indexed array. This permits a formal treatment of inductively defined data structures in the heap but tends to produce instances of modified component mappings in arguments to inductively defined assertions. The major weapons against these troublesome mappings are assertions which describe spatial separation of data structures. Three example proofs are sketched.

1 Introduction

The power of the Floyd/Hoare treatment of imperative programs [8][11] lies in its use of variable substitution to capture the semantics of assignment: simply, R^x_E, the result of replacing every free occurrence of variable x in R by formula E, is the precondition which guarantees that assignment $x := E$ will terminate in a state satisfying R.[1] At a stroke difficult semantic questions that have to do with stores and states are converted into simpler syntactic questions about first-order logical formulæ.

We encounter several difficulties when we attempt to use a similar approach to deal with programs which manipulate and modify recursive data structures defined by pointers. The first difficulty, whose solution has been known for some time, is aliasing: distinct and very different pointer formulæ may refer to the same object. The second difficulty is the treatment of assertions which include inductive formulæ describing heap data structures. The final difficulty is the complexity of the proofs: not only do we have to reason formally about sets, sequences, graphs and trees, we have to make sure that the locality of assignment operations is reflected in the treatment of assertions about the heap.

For all of these reasons, Hoare logic isn't widely used to verify pointer programs. Yet most low-level and all object-oriented programs use heap pointers freely. If we wish to prove properties of the kind of programs that actually get written and used, we shall have to deal with pointer programs on a regular basis.

[1] I neglect definedness conditions throughout this paper.

In one situation program verification is a practical necessity. The idea of ‘proof-carrying code’ (see, for example, Necula and Lee [22] and Appel and Felty [1]) is that programs should be distributed along with a proof of their properties, the proof to be checked by each user before the program is used. Machine checkers are simple, reliable and fast, so if appropriate proofs can be provided we won’t have to spend ages reading the fine print before running our latest downloaded system extension.

Proof-carrying code is a long way off, but being able to deal effectively with pointer programs will take us a step along the way.

This paper is therefore about verifying the properties of programs. It is not about proof development, but it is not intended as a challenge to those who prefer program refinement to program verification. Once we can make reliable proofs about imperative pointer algorithms, surely the mechanisms developed to support proof can be used in aid of other activities.

1.1 Background

This paper was inspired by the work of Morris, who put forward in 1981 [21] axioms for assignment to object components, described a mechanism for dealing with inductively defined data structures and presented a semi-formal proof of the Schorr-Waite graph marking algorithm. Earlier still, Burstall presented in 1972 [6] a treatment of list-processing algorithms and pointed out the analogy between the treatment of array element and object components; his Distinct Non Repeating List Systems made possible succinct and convincing proofs of list and tree algorithms. Recently Reynolds [24] revisited Burstall’s work, refining the treatment of spatial separation between objects and data structures in the heap and extending its range of application, but working with forward rather than backward reasoning.

Attempts have been made to apply Hoare logic to pointer programs by incorporating a model of the store, or part of the store, into the assertion logic. Luckham and Suzuki [19], Leino [18] and Bijlsma [3], for example, identify a subsection of the heap with each pointer type. Kowaltowski [16] includes the entire heap.

Other work recognises the analogy between array element and object component assignment but doesn’t do so effectively: both Hoare and Wirth [13] and Gries and Levin [10], for example, give an axiom for object-component assignment which deals only with the simplest non-pointer cases and neglects entirely to deal with pointer aliasing.

Cousot [7] gives a brief survey of other work, most of which is semantic in character and does not address the practical concerns of program verification.

1.2 Notation

I write $\rightarrow, \wedge, \vee, \neg$ for logical implication, conjunction, disjunction and negation; $\hat{=}$ means equal by definition; @ is append (sequence concatenation); $\pitchfork$ is disjointness of sequences; $\in$ is sequence and/or set membership. E_F^x is the result of substituting formula F for every free occurrence of variable x in formula E. $A \oplus B \mapsto E$ is a mapping which is everywhere the same as A, except at B which it maps to E. I use $\Rightarrow_f$ to define *f*-linked sequence data structures from section 6 onwards.

2 The Problem of Aliasing

The Hoare logic treatment of assignment is sound when we are sure that distinct variable names refer to distinct storage locations; it can mislead if that assurance is lost. *Aliasing*, which occurs when distinct formulæ describe the same storage location, comes in at least the following range of flavours.

- *Parameter aliasing* is the best known and probably the most difficult to deal with. It arises during the execution of procedures and functions, when a call-by-reference parameter (a **var** parameter in Pascal [14], for example) stands for a storage location outside the procedure which is also nameable in another way from within the procedure.
- *Subscript aliasing* arises in languages which use arrays: $b[I]$ and $b[J]$ are the same storage location just when $I = J$ as integers.
- *Pointer aliasing*, analysed below, arises when an object (a node, a record) can be referred to indirectly via a pointer value.[1]
- *Overlap aliasing* occurs when storage locations can contain storage locations, as when an object is updatable by assignment, simultaneously updating all of its components, and those components are each separately updatable storage locations.
- *View aliasing* occurs when the same area of store can be addressed in different ways, as with Pascal variant records, C [15] union types, or C casting.

Aliasing is caused by identity or overlap of *lvalues* (addresses of objects in the store, see Strachey [26]), but both subscript and pointer aliasing can be dealt with by comparing *rvalues* (contents of storage locations). That makes them in principle tractable in the Floyd/Hoare tradition, which deals entirely with rvalues. Parameter, overlap and view aliasing are outside the scope of this paper.

2.1 Subscript Aliasing

Distinct formulæ indexing the same array will be equivalent as lvalues if the rvalues of their subscript expressions are equal. The Pascal program fragment

```
b[i]:=b[j]+1;
if b[i]=b[j] then writeln(output, "aliased!")
else writeln(output, "distinct")
```

[1] In languages such as C [15] which allow pointers to stack components, pointer aliasing is used to imitate parameter aliasing. The obvious implementation of call by reference depends on the use of pointers. Pointer aliasing and parameter aliasing are, therefore, closely related. But at source language level they are distinct phenomena.

will print `aliased!` when `i=j`, or `distinct` when `i<>j`.

Following McCarthy and Painter [20] and Hoare and Wirth [13], there's a well-known solution to subscript aliasing. Even though, in every practical implementation of a programming language, an array is a collection of separately addressable storage locations, it can be treated as if it was a single variable containing a mapping from indices to rvalues. The instruction $b[I]:=E$ is considered to assign a new mapping $b \oplus I \mapsto E$ to b, and therefore $R^{b}_{b \oplus I \mapsto E}$ is the precondition that the assignment $b[I]:=E$ will terminate in a state satisfying *R*. Array element access is resolved by comparing indices, so that $(b \oplus I \mapsto E)[J] = (\text{if } I = J \text{ then } E \text{ else } b[J] \text{ fi})$.

This interpretation of array element assignment is a complete solution to the problem of subscript aliasing, though it must be used carefully with concurrent assignments in case there are aliases in the set of assigned locations. It resolves the problem entirely by the *rvalue* comparison $I = J$ in the reduction rule, even though aliasing is between *lvalues* $b[I]$ and $b[J]$.

2.2 Pointer Aliasing

In practice a computer memory is a giant array, and memory addresses – the primitive mechanism underlying lvalues and pointers – are its indices. Whenever two distinct occurrences of pointer values are equal, we have pointer aliasing.

Consider, for example, the problem (part of the in-place list-reversal example below) of moving an element from the head of one *tl*-linked list *r* to the head of a similar list *p*, using assignment to heap object components. If we write it as a sequence of non-concurrent assignments[1] we must use an auxiliary variable *q* and a sequence of at least four instructions, one of which alters a pointer in the heap. This is one solution:

$q:=r;\ r:=r.tl;\ q.tl:=p;\ p:=q$

At every intermediate stage of execution of this program there is more than one way to refer to particular objects in the store. After the first assignment, *q* and *r* are the same pointer; after the second, *q.tl* and *r*; after the third, *q.tl* and *p*; after the last, *p* and *q*. If the *p* and *r* lists aren't disjoint collections of objects, still more aliasing may be produced.

It's tempting to treat the heap as a pointer-indexed array of component-indexed objects, perhaps subdividing it by object type into sub-arrays (see, for example, Luckham and Suzuki [19] and Leino [18]). But in practice this has proved awkward to deal with. First, it means there has to be a translation between object-component formulæ in the program on the one hand and array-indexing formulæ in assertions on the other. Second, it forces us towards 'global reasoning': every object component assignment seems to affect every assertion which has to do with the heap. By contrast the Floyd-Hoare treatment of assignment to a variable concentrates attention on assertions that involve that variable, leaving others untouched, making steps of 'local reasoning' whose restriction to particular formulæ matches the locality of assignment

[1] I do not consider concurrent assignment in this paper.

to a single variable. Third, our assertions about the content of the heap usually need to be expressed inductively, using auxiliary definitions which can easily hide aliases.

Even though these difficulties might in principle be overcome, it feels like the wrong thing to do: we don't program with a memory array in mind but rather in terms of distinct variables, arrays, objects and components, and our reasoning should as far as possible operate at the same level as our thinking.

3 The Treatment of Subscript Aliasing

In [21] Morris introduced an assignment rule for assignment to components of heap objects which generalised Burstall's treatment [6] of hd/tl structures. He gave a treatment of inductively defined data structures using the notion of paths between objects, and presented a proof of the Schorr-Waite graph-marking algorithm [25] which seems far closer to a proof of the original than other published treatments [9][28].

3.1 Morris's Treatment

Morris treats a language which is like Java [2] in that it has pointer-aliasing but no parameter-aliasing, and no whole-object assignment. Program (stack) variables like *x*, *y*, *p*, *q* can hold pointers to objects in the heap. Objects (in the heap) have components indexed by names like *e, f, g, h* and are referred to using dot-suffix notation, for example *p.e.e.f*.

Because there is no whole-object assignment, aliasing is only of components of objects. Because there is no arithmetic on component names we know immediately, when *e* and *f* are distinct component names and no matter what the values of *A* and *B*, that *A.e* can't be the same lvalue as *B.f*. Similarly, we know that *A.e* and *B.e* will be the same lvalue just when the rvalues *A* and *B* are equal.[1] Lvalue-aliasing of object components can therefore be detected by identity of component name and rvalue comparison of pointers.

These insights made it possible to define a rule for object-component assignment which avoids mentioning the memory array altogether:

$$\frac{Q \rightarrow R_E^{/A.f}}{\{Q\}A.f := E\{R\}}$$

Object component substitution $R_E^{/A.f}$ is just like variable substitution except when it is dealing with object component references. Morris's axioms appear to be:

$$\frac{}{(B.g)_E^{/A.f} \mathrel{\hat{=}} B.g}\;(f \text{ and } g \text{ distinct}) \qquad \frac{}{(B.f)_E^{/A.f} \mathrel{\hat{=}} \text{if } A = B \text{ then } E \text{ else } B.f \text{ fi}}$$

Formally, this treatment deals only with single occurrences of component names in object component formulae – *p.hd* and *p.tl* are dealt with correctly, for example, but

[1] I assume, for simplicity, that distinct types of object use distinct component names.

p.hd.tl isn't – so it is valid only with a restricted programming language and a restricted vocabulary of assertion logics. There are similar axioms for assignment to array elements, similarly flawed – $b[i]$ is dealt with, but not $b[b[i]]$.[1]

3.2 Calculating Object-Component Assignment Axioms

It is possible to calculate object-component substitution axioms which work without restriction in Morris's target language. Since objects can't overlap, we can treat the heap as a pointer-indexed collection of objects, each of which is a name-indexed collection of components. An object-component reference $A.f$ in a heap H corresponds to a double indexing, once of the heap and once of the object.

$$[\![A.f]\!]\mathsf{H} \mathrel{\hat{=}} \mathsf{H}\left[[\![A]\!]\mathsf{H}\right][f]$$

Assigning a value to *A.f* replaces the object $\mathsf{H}[A]$ with a new mapping and therefore the heap becomes a new mapping as well:

$$[\![F_E^{/A.f}]\!]\mathsf{H} \mathrel{\hat{=}} [\![F]\!]\mathsf{H}', \text{ where } \mathsf{H}' = \left(\mathsf{H} \oplus \left([\![A]\!]\mathsf{H}\right) \mapsto \left(\mathsf{H}\left[[\![A]\!]\mathsf{H}\right] \oplus f \mapsto [\![E]\!]\mathsf{H}\right)\right)$$

When *f* and *g* are distinct component names:

$$[\![(B.g)_E^{/A.f}]\!]\mathsf{H} = [\![B.g]\!]\mathsf{H}' = \mathsf{H}'\left[[\![B]\!]\mathsf{H}'\right][g] = \mathsf{H}'\left[[\![B_E^{/A.f}]\!]\mathsf{H}\right][g]$$

$$= \left(\mathsf{H} \oplus \left([\![A]\!]\mathsf{H}\right) \mapsto \left(\mathsf{H}\left[[\![A]\!]\mathsf{H}\right] \oplus f \mapsto [\![E]\!]\mathsf{H}\right)\right)\left[[\![B_E^{/A.f}]\!]\mathsf{H}\right][g]$$

$$= \left(\text{if } [\![A]\!]\mathsf{H} = [\![B_E^{/A.f}]\!]\mathsf{H} \text{ then } \left(\mathsf{H}\left[[\![A]\!]\mathsf{H}\right] \oplus f \mapsto [\![E]\!]\mathsf{H}\right) \text{ else } \mathsf{H}\left[[\![B_E^{/A.f}]\!]\mathsf{H}\right] \text{ fi}\right)[g]$$

$$= \text{if } [\![A]\!]\mathsf{H} = [\![B_E^{/A.f}]\!]\mathsf{H} \text{ then } \left(\mathsf{H}\left[[\![A]\!]\mathsf{H}\right] \oplus f \mapsto [\![E]\!]\mathsf{H}\right)[g] \text{ else } \mathsf{H}\left[[\![B_E^{/A.f}]\!]\mathsf{H}\right][g] \text{ fi}$$

$$= \text{if } [\![A]\!]\mathsf{H} = [\![B_E^{/A.f}]\!]\mathsf{H} \text{ then } \mathsf{H}\left[[\![A]\!]\mathsf{H}\right][g] \text{ else } \mathsf{H}\left[[\![B_E^{/A.f}]\!]\mathsf{H}\right][g] \text{ fi}$$

$$= \text{if } [\![A]\!]\mathsf{H} = [\![B_E^{/A.f}]\!]\mathsf{H} \text{ then } \mathsf{H}\left[[\![B_E^{/A.f}]\!]\mathsf{H}\right][g] \text{ else } \mathsf{H}\left[[\![B_E^{/A.f}]\!]\mathsf{H}\right][g] \text{ fi}$$

$$= \mathsf{H}\left[[\![B_E^{/A.f}]\!]\right][g] = [\![\left(B_E^{/A.f}\right).g]\!]\mathsf{H}$$

With identical component names:

$$[\![(B.f)_E^{/A.f}]\!]\mathsf{H} = [\![B.f]\!]\mathsf{H}' = \mathsf{H}'\left[[\![B]\!]\mathsf{H}'\right][f] = \mathsf{H}'\left[[\![B_E^{/A.f}]\!]\mathsf{H}\right][f]$$

$$= \left(\mathsf{H} \oplus \left([\![A]\!]\mathsf{H}\right) \mapsto \left(\mathsf{H}\left[[\![A]\!]\mathsf{H}\right] \oplus f \mapsto [\![E]\!]\mathsf{H}\right)\right)\left[[\![B_E^{/A.f}]\!]\mathsf{H}\right][f]$$

[1] In examples Morris deals with multiple occurrences of component names or array names by sometimes introducing nested substitutions, sometimes introducing constants – replacing $b[b[j]] = j$, for example, by $b[b0] = j \wedge b[j] = b0$. It may be that his less than formal statement of the mechanism is misleading.

$$= \left(\text{if } \llbracket A \rrbracket \mathsf{H} = \llbracket B_E^{/A.f} \rrbracket \mathsf{H} \text{ then } \left(\mathsf{H}\left[\llbracket A \rrbracket \mathsf{H}\right] \oplus f \mapsto \llbracket E \rrbracket \mathsf{H}\right) \text{ else } \mathsf{H}\left[\llbracket B_E^{/A.f} \rrbracket \mathsf{H}\right] \text{ fi}\right)[f]$$

$$= \text{if } \llbracket A \rrbracket \mathsf{H} = \llbracket B_E^{/A.f} \rrbracket \mathsf{H} \text{ then } \left(\mathsf{H}\left[\llbracket A \rrbracket \mathsf{H}\right] \oplus f \mapsto \llbracket E \rrbracket \mathsf{H}\right)[f] \text{ else } \mathsf{H}\left[\llbracket B_E^{/A.f} \rrbracket \mathsf{H}\right][f] \text{ fi}$$

$$= \text{if } \llbracket A \rrbracket \mathsf{H} = \llbracket B_E^{/A.f} \rrbracket \mathsf{H} \text{ then } \llbracket E \rrbracket \mathsf{H} \text{ else } \llbracket \left(B_E^{/A.f}\right).f \rrbracket \mathsf{H} \text{ fi}$$

$$= \llbracket \text{ if } A = B_E^{/A.f} \text{ then } E \text{ else } \left(B_E^{/A.f}\right).f \text{ fi} \rrbracket \mathsf{H}$$

In each case the calculated equivalence leads to an axiom which is identical to Morris's except that *B* on the right-hand side of the axiom is replaced by $B_E^{/A.f}$.

3.3 The Component-as-Array Trick

The axioms for object component substitution, following an assignment $A.f := E$, for distinct component names *f* and *g*, are

$$\overline{(B.g)_E^{/A.f} \mathrel{\hat{=}} \left(B_E^{/A.f}\right).g} \qquad \overline{(B.f)_E^{/A.f} \mathrel{\hat{=}} \text{if } A = B_E^{/A.f} \text{ then } E \text{ else } \left(B_E^{/A.f}\right).f \text{ fi}}$$

The standard treatment of an assignment $b[I] := E$ gives us for distinct arrays *b* and *c*

$$c[J]^b_{b \oplus I \mapsto E} = c\left[J^b_{b \oplus I \mapsto E}\right] \qquad \begin{aligned} b[J]^b_{b \oplus I \mapsto E} &= (b \oplus I \mapsto E)\left[J^b_{b \oplus I \mapsto E}\right] \\ &= \text{if } I = J^b_{b \oplus I \mapsto E} \text{ then } E \text{ else } b\left[J^b_{b \oplus I \mapsto E}\right] \text{ fi} \end{aligned}$$

It is clear from the correspondence between these treatments that object-component substitution is formally equivalent to a treatment of object components as pointer-indexed arrays. That is, assignment to component *f* of an object pointed to by *A* can be treated as if it were an assignment to the *A*-indexed component of an array *f*, and access to the *f* component of an object pointed to by *A* as selection of the *A*th component of an array *f*.

This observation is certainly not novel: Burstall gives an equivalent in [6], and it may be that Morris intended this reading in [21]. It is worth stating clearly, however, to clarify what seems otherwise to be imperfectly understood 'folk knowledge'.

The advantages of the component-as-array treatment are considerable. First of all, it is obvious that it enables the calculation of weakest preconditions. Second, it means that we don't need a new structural induction to deal with object component substitution: a considerable advantage when mechanising proof and proof-checking. Finally, and most importantly in the context of this paper, it makes possible a formal treatment of object component substitution into inductively defined formulæ.

I feel it's necessary, despite its advantages, to emphasise that the treatment is doubly a trick. It's a trick built on the array-as-mapping trick of McCarthy and Painter. It's violently at odds with our understanding of how heaps work in practice. It isn't clear how much of it would survive a relaxation of the restrictions we have imposed on our programming language.

$$\frac{Q \to R^{x}_{E}}{\{Q\}x := E\{R\}} \qquad \frac{Q \to R^{f}_{f \oplus A \mapsto E}}{\{Q\}A.f := E\{R\}} \qquad \frac{\{Q\}S\{R'\} \quad R' \to R}{\{Q\}S\{R\}}$$

$$\frac{Q \to P \quad \{P \wedge B\}S\{P\} \quad P \wedge \neg B \to R \quad P \wedge B \to t > 0 \quad \{P \wedge B \wedge t = vt\}S\{t < vt\}}{\{Q\} \text{ while } B \text{ do } S \text{ od } \{R\}}$$

$$\frac{\{Q \wedge B\}S_{then}\{R\} \quad \{Q \wedge \neg B\}S_{else}\{R\}}{\{Q\}\text{if } B \text{ then } S_{then} \text{ else } S_{else} \text{ fi}\{R\}} \qquad \frac{\{Q\}S1\{Q'\} \quad \{Q'\}S2\{R\}}{\{Q\}S1; S2\{R\}}$$

Fig. 1. Hoare-triple rules

3.4 Restricted Global Reasoning

If we treat the heap as a global array, and convert all our object component assignments to heap-array element assignments, then we have *global reasoning*, because substitution of a new mapping for the heap array affects every assertion about any part of the heap. By contrast the Floyd-Hoare treatment of variable assignment gives us *local reasoning*: only those assertions which mention the assigned variable are affected by substitution.

Object component substitution and the component-as-array trick each offer a restricted global reasoning. Some locality is achieved effortlessly, when formulæ which only mention component *f* aren't affected by assignment to component *g*. But on the other hand the interaction of assignment with inductive definitions, discussed below, needs careful treatment if a global reasoning explosion is to be avoided.

4 Hoare-Triple Rules

I consider a language which has assignment to variables and object components, while-do-od, if-then-else-fi and instruction sequence (fig. 1). The usual caveats apply to the while-do-od rule: *vt* must be a fresh variable, *t* must be an integer-valued function of the state. It is useful to include a postcondition-strengthening rule. Component indexing $A.(B \oplus C \mapsto E)$ is equivalent to $(\text{if } A = C \text{ then } E \text{ else } A.B \text{ fi})$.

I omit from the rules anything which requires definedness of formulæ. Definedness is especially important in pointer-manipulating programs, because nil is such a dangerous value – often in use, of pointer type, and yet not dot-suffixable. Nevertheless, it would add little to the discussion in this paper to deal with definedness, if only because my examples don't attempt to traverse nil pointers.

5 Three Small Examples

Cycles in the heap show pointer aliasing in its rawest form. Three small mechanical proofs, calculated in Jape [5], are shown in full detail except for trivial implications.

```
1 :p.b=3→p.b=3                                   {→-I,hyp}
2 :p.b=3→p.(a⊕p↦p).b=3                           A.(B⊕A↦E)≙E 1
3 :p.b=3→p.(a⊕p↦p).(a⊕p↦p).b=3                   A.(B⊕A↦E)≙E 2
4 :p.b=3→p.(a⊕p↦p).(a⊕p↦p).(a⊕p↦p).b=3           A.(B⊕A↦E)≙E 3
5 :{p.b=3}(p.a:=p){p.a.a.a.b=3}                  := 4
```

Fig. 2. A single-step cycle

```
1 :p.d=3→p.d=3                                                    {→-I,hyp}
2 :p.d=3→q.b.(c⊕q.b↦p).d=3                                        A.(B⊕A↦E)≙E 1
3 :p.d=3→p.(a⊕p↦q).b.(c⊕q.b↦p).d=3                                A.(B⊕A↦E)≙E 2
4 :p.d=3→q.b.(c⊕q.b↦p).(a⊕p↦q).b.(c⊕q.b↦p).d=3                    A.(B⊕A↦E)≙E 3
5 :{p.d=3}(q.b.c:=p){q.b.c.(a⊕p↦q).b.c.d=3}                       := 4
6 :q.b.c.(a⊕p↦q).b.c.d=3→q.b.c.(a⊕p↦q).b.c.d=3                    {→-I,hyp}
7 :q.b.c.(a⊕p↦q).b.c.d=3→p.(a⊕p↦q).b.c.(a⊕p↦q).b.c.d=3            A.(B⊕A↦E)≙E 6
8 :{q.b.c.(a⊕p↦q).b.c.d=3}(p.a:=q){p.a.b.c.a.b.c.d=3}             := 7
9 :{p.d=3}(q.b.c:=p;p.a:=q){p.a.b.c.a.b.c.d=3}                    sequence 5,8
```

Fig. 3. A multi-step cycle (version 1)

```
1 :p.d=3→p.d=3                                                                  {→-I,hyp}
2 :p.d=3→q.b.(c⊕q.b↦p).d=3                                                      A.(B⊕A↦E)≙E 1
3 :p.d=3→p.(a⊕p↦q).b.(c⊕q.b↦p).d=3                                              A.(B⊕A↦E)≙E 2
4 :p.d=3→q.b.(c⊕q.b↦p).(a⊕p↦q).b.(c⊕q.b↦p).d=3                                  A.(B⊕A↦E)≙E 3
5 :p.d=3→p.(a⊕p↦q).b.(c⊕q.b↦p).(a⊕p↦q).b.(c⊕q.b↦p).d=3                          A.(B⊕A↦E)≙E 4
6 :{p.d=3}(p.a:=q){p.a.b.(c⊕q.b↦p).a.b.(c⊕q.b↦p).d=3}                           := 5
7 :p.a.b.(c⊕q.b↦p).a.b.(c⊕q.b↦p).d=3→p.a.b.(c⊕q.b↦p).a.b.(c⊕q.b↦p).d=3          {→-I,hyp}
8 :{p.a.b.(c⊕q.b↦p).a.b.(c⊕q.b↦p).d=3}(q.b.c:=p){p.a.b.c.a.b.c.d=3}             := 7
9 :{p.d=3}(p.a:=q;q.b.c:=p){p.a.b.c.a.b.c.d=3}                                  sequence 6,8
```

Fig. 4. A multi-step cycle (version 2)

First a single-step cycle established by a single assignment.

$$\{p.b = 3\}\, p.a := p \,\{p.a.a.a.b = 3\}$$

The proof (fig. 2) consists of a use of the assignment rule (line 5) followed by three applications of component-indexing simplification on lines 4, 3 and 2.

Next, a multi-step cycle established by a sequence of two assignments.

$$\{p.d = 3\}\, q.b.c := p;\; p.a := q \,\{p.a.b.c.a.b.c.d = 3\}$$

The proof (fig. 3) consists of an application of the sequence rule (line 9), two applications of the assignment rule (lines 8 and 5) and various uses of component-indexing simplification. Note that we can't simplify $q.b.c.(a \oplus p \mapsto q)$ on line 6, but the mapping can be eliminated on line 3, once $q.b.c$ has been simplified to p.

If we make the same cycle with the same instructions, but executed in the reverse order (fig. 4), neither of the mappings generated by the assignment rule on line 8 can be eliminated immediately, but they are dealt with eventually, on lines 4 and 2.

6 Substitution and Auxiliary Definitions

When we make proofs of programs which manipulate heap objects using pointers, our assertions will usually call upon auxiliary definitions of data structures. Auxiliary definitions are useful for all kinds of reasons: they allow us to parameterise important assertions in user-defined predicates, they let us define inductive calculations, they shorten descriptions. But the Floyd/Hoare mechanism, even in the absence of pointer aliasing, has difficulty with assertions which contain arbitrary auxiliary defined formulæ (henceforth *adfs*).

If, for example, we define a predicate

$$\mathrm{F}(z) \mathrel{\hat{=}} (x = z)$$

then $\mathrm{F}(y)$ both asserts that y has the same value as x and contains an implicit occurrence of x. Substitution deals only with explicit occurrences, so the variable assignment rule would seem to allow us to conclude mistakenly

$$\{\mathrm{F}(y)\}x := x + 1\{\mathrm{F}(y)\}$$

The difficulty would disappear if we were to insist that adfs are expanded from their definitions before we use substitution. That would not merely be inconvenient but impossible in general, because inductive definitions can expand indefinitely. If we insist, however, that definitions are program-variable closed – that is, that they have no free occurrences of variable names which can be the target of substitution – then we can deal with them in unexpanded form.

For example, we might define

$$\mathrm{F}(u, v) \mathrel{\hat{=}} u = v$$

and the assignment rule would correctly calculate

$$\{\mathrm{F}(x + 1, y)\}x := x + 1\{\mathrm{F}(x, y)\}$$

The problem of auxiliary definitions of heap data structures is not quite so easy to solve. Reynolds [23] deals with *assertion procedures*, but his intentions and his approach are quite distinct from that developed below.

6.1 Inductive Auxiliary Definitions of Data Structures

Pointer-linked data-structure definitions are usually inductively defined. Indeed it is difficult to see how we could do without induction when specifying programs which manipulate data structures via pointers. Making them program-variable closed doesn't deal with object component assignment, but the component-as-array trick allows us to make them component-array closed as well.

Following Burstall, Morris and Reynolds, an example data structure which uses a component array parameter is the sequence of objects[1] $A \Rightarrow_f B$ generated by starting

[1] It's helpful to think of this formula as generating a sequence of objects, but of course it actually generates a sequence of pointer values.

$$A \Rightarrow_f B \hat{=} \text{if } A = B \text{ then } \langle\,\rangle \text{ else } \langle A\rangle @ \left(A.f \Rightarrow_f B\right) \text{ fi}$$

Fig. 5. An inductive definition of *f*-linked sequences in the heap

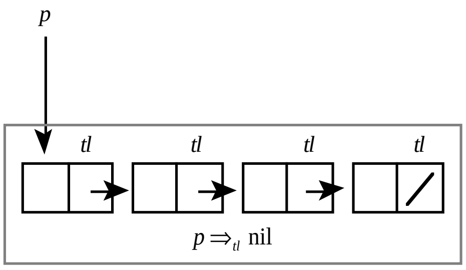

Fig. 6. The sequence $p \Rightarrow_{tl}$ nil

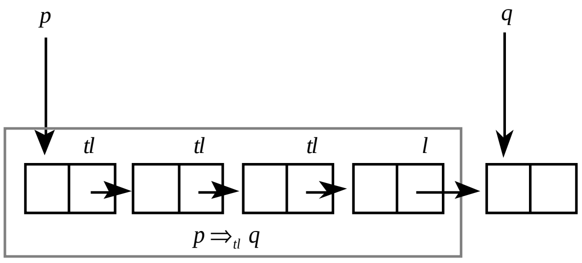

Fig. 7. The sequence $p \Rightarrow_{tl} q$

at the object pointed to by *A* and following links in *f* components until a pointer equal to *B* is encountered.

For various reasons (not least the fact that nil doesn't point to an object), *B* isn't included in the sequence. The definition is both program-variable and component-array closed: it mentions nothing but constants and its parameters *A*, *B* and *f*. The sequence it generates isn't necessarily a list – that is, a finite sequence with no repetitions – because cycles in the heap may mean that we never reach *B*. It will not be defined if we attempt to traverse a nil pointer before reaching *B*.

If we consider objects with *hd* and *tl* components, as in the list reversal and list merge examples below, then $p \Rightarrow_{tl}$ nil can represent what we normally think of as 'the list *p*' (fig. 6). If *p* and *q* point to distinct objects, then $p \Rightarrow_{tl} q$ describes a list fragment, the sequence of objects linked by *tl* components starting with the one pointed to by *p* up to but not including the one pointed to by *q* (fig. 7). In each of these examples I have assumed that the data structure is acyclic.

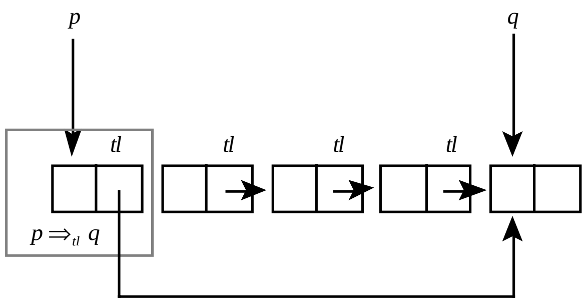

Fig. 8. Effect of the assignment $p.tl := q$ on the sequence $p \Rightarrow_{tl} q$

7 Spatial Separation, Global Reasoning and Inductive Definitions

What imperative graph, tree and list algorithms all seem to do is to work with disjoint bits of data structures, nibbling away at the edges, moving a node from here to there, swinging pointers, altering values in exposed components. The locality of assignment means that changes made in one area of the heap don't affect objects elsewhere. But because pointer aliasing is always a possibility, our reasoning must continually take into account the possibility that the location we are altering is referred to under another name. In practice we take that possibility into account most often in order to dismiss it. Our logic must make this easy to do, and we must choose our assertions to exploit this capability.

Substitution into $x \Rightarrow_f y$, as a result of the assignment $A.f := E$, generates $x \Rightarrow_{f \oplus A \mapsto E} y$. That expands to give a formula which contains indefinitely many occurrences of the mapping $f \oplus A \mapsto E$, in $x.(f \oplus A \mapsto E)$, $x.(f \oplus A \mapsto E).(f \oplus A \mapsto E)$, and so on. This explosion of effects, produced by an assignment which affects only a single location, must either be avoided or effectively dealt with. It arises with any inductive auxiliary definition which has a component-array parameter.

By contrast, assignment is operationally local. Consider, for example the $p \Rightarrow_{tl} q$ data structure of fig. 7. Executing $p.tl := q$ will swing the pointer in the first box to point at the last, changing the data structure to that in fig. 8.

Only one component of the heap changes because of the assignment. The challenge is to imitate this simplicity in our reasoning, to avoid an explosion of mappings and combat what Hoare and Jifeng [12] call 'the complexity of pointer-swing'.

In this particular case the result of substitution is easy to unravel:

$$\begin{aligned} (p \Rightarrow_{tl} q)^{tl}_{tl \oplus p \mapsto q} &= p \Rightarrow_{tl \oplus p \mapsto q} q \\ &= \text{if } p = q \text{ then } \langle\,\rangle \text{ else } \langle p \rangle @ p.(tl \oplus p \mapsto q) \Rightarrow_{tl \oplus p \mapsto q} q \text{ fi} \\ &= \text{if } p = q \text{ then } \langle\,\rangle \text{ else } \langle p \rangle @ q \Rightarrow_{tl \oplus p \mapsto q} q \text{ fi} \end{aligned}$$

$$= \text{if } p = q \text{ then } \langle\,\rangle \text{ else } \langle p\rangle @ \langle\,\rangle \text{ fi}$$

$$= \text{if } p = q \text{ then } \langle\,\rangle \text{ else } \langle p\rangle \text{ fi}$$

Not every example is so straightforward. Consider the effect of the assignment $x.tl := y$ on the postcondition $x \neq \text{nil} \wedge \text{list}(x \Rightarrow_{tl} \text{nil})$, where the list predicate[1] asserts that a sequence is finite and non-repetitive. We may proceed with substitution, expansion and simplification as before:

$$\left(x \neq \text{nil} \wedge \text{list}(x \Rightarrow_{tl} \text{nil})\right)^{tl}_{tl \oplus x \mapsto y} = x \neq \text{nil} \wedge \text{list}\left(x \Rightarrow_{tl \oplus x \mapsto y} \text{nil}\right)$$

$$= x \neq \text{nil} \wedge \text{list}\left(\text{if } x = \text{nil then } \langle\,\rangle \text{ else } \langle x\rangle @ x.(tl \oplus x \mapsto y) \Rightarrow_{tl \oplus x \mapsto y} \text{nil fi}\right)$$

$$= x \neq \text{nil} \wedge \text{list}\left(\text{if } x = \text{nil then } \langle\,\rangle \text{ else } \langle x\rangle @ y \Rightarrow_{tl \oplus x \mapsto y} \text{nil fi}\right)$$

$$= x \neq \text{nil} \wedge \text{list}\left(\langle x\rangle @ y \Rightarrow_{tl \oplus x \mapsto y} \text{nil}\right)$$

Without further information we can go no further, but we can be sure that this is the weakest precondition, because all we've done is to use component-as-array substitution, expand a definition and evaluate a conditional.

We've by no means achieved local reasoning yet, because there is a mapping $tl \oplus x \mapsto y$ in the argument to an inductively defined auxiliary formula. Observe, however, that when x doesn't point to any of the components of $y \Rightarrow_{tl} \text{nil}$, assignment to $x.tl$ can't affect the meaning of that formula. In those circumstances, therefore, $y \Rightarrow_{tl} \text{nil} = y \Rightarrow_{tl \oplus x \mapsto E} \text{nil}$ for any formula E. (This is easily shown formally by induction on the length of finite sequences.) We might, that is, be content to assume separation of objects in the heap, and to prove the simpler precondition

$$x \neq \text{nil} \wedge x \notin y \Rightarrow_{tl} \text{nil} \wedge \text{list}\left(\langle x\rangle @ y \Rightarrow_{tl} \text{nil}\right)$$

Note that the assumption ought to hold, because if x does point into $y \Rightarrow_{tl} \text{nil}$ – that is, if $y \Rightarrow_{tl} \text{nil}$ $y \Rightarrow_{tl} x @ x \Rightarrow_{tl} \text{nil}$ – then we don't have a non-repeating sequence and the precondition is false:

$$x \neq \text{nil} \wedge \text{list}\left(\langle x\rangle @ y \Rightarrow_{tl \oplus x \mapsto y} \text{nil}\right)$$

$$= x \neq \text{nil} \wedge \text{list}\left(\langle x\rangle @ y \Rightarrow_{tl \oplus x \mapsto y} x @ x \Rightarrow_{tl \oplus x \mapsto y} \text{nil}\right)$$

$$= x \neq \text{nil} \wedge \text{list}\left(\langle x\rangle @ y \Rightarrow_{tl} x @ \langle x\rangle @ y \Rightarrow_{tl \oplus x \mapsto y} \text{nil}\right)$$

What's being appealed to by adding $x \notin y \Rightarrow_{tl} \text{nil}$ to the precondition, thereby eliminating a mapping, is the *principle of spatial separation*. There is a set of objects whose components help define the meaning of $y \Rightarrow_{tl} \text{nil}$, and if the object x points to isn't one of that set, then the assignment $x.tl := E$ can't have any effect on $y \Rightarrow_{tl} \text{nil}$.

In practice spatial-separation assertions like $x \notin y \Rightarrow_{tl} \text{nil}$ aren't plucked out of the air – they are found in invariants, as shown in the list reversal, list merge and graph marking examples below.

[1] Defined by heap-independent axioms: see section 9.1.

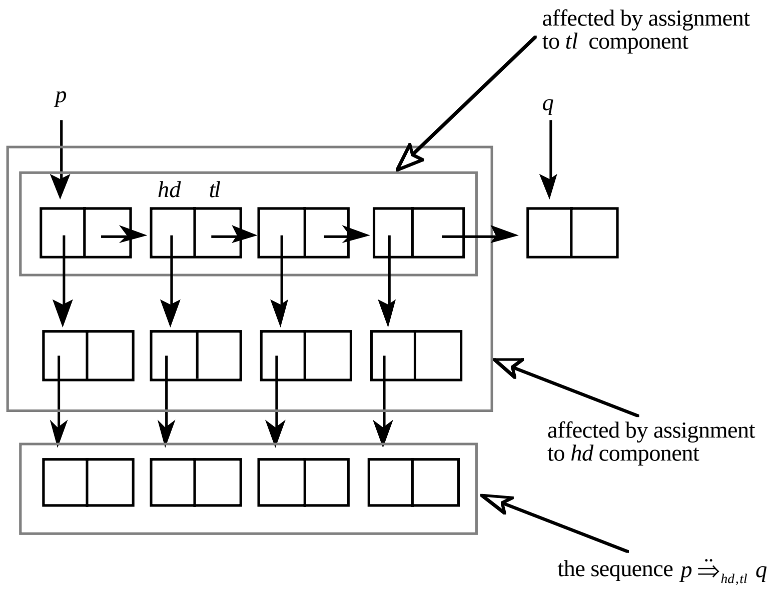
affected by assignment to *tl* component
p
q
hd *tl*
affected by assignment to *hd* component
the sequence $p \overset{\cdot\cdot}{\Rightarrow}_{hd,tl} q$

Fig. 9. Offset sequence with different support sets for different components

What's surprising is that lists and sequences aren't a special case. Spatial separation is exactly what we need to deal in general with inductively defined heap data structures. Any data structure in a finite heap depends on the component values of a finite set of objects. If we assign to a component of an object outside that set, the data structure can't be affected. Faced with an adf whose arguments include a mapping $B \oplus A \mapsto E$, therefore, we can either expand it until all the affected objects are exposed – as, for example, we expanded $x \Rightarrow_{tl \oplus x \mapsto y}$ nil above – or we can show that because of spatial separation the mapping is never used, no matter how far we expand the formula – as in the case of $y \Rightarrow_{tl \oplus x \mapsto y}$ nil above – or we can do both steps, one after the other – or we are stuck for the moment, and we have to leave the mapping alone.

7.1 Spatial Separation Examples

Spatial separation of a data structure and an object A can always be expressed as $A \notin S$, where S is a set of objects whose component values define the data structure. In straightforward cases, like the $\Rightarrow_f$ definition of fig. 5, the set of objects is just the data structure itself.

Offset Sequences. Consider the definition

$$A \overset{\cdot\cdot}{\Rightarrow}_{f,g} B \mathrel{\hat{=}} \text{if } A = B \text{ then } \langle\,\rangle \text{ else } \langle A.f.f \rangle @ A.g \overset{\cdot\cdot}{\Rightarrow}_{f,g} B \text{ fi}$$

The formula $p \overset{..}{\Rightarrow}_{hd,tl}\, q$ describes a sequence of objects not necessarily linked together, and offset by two steps from $p \Rightarrow_{tl} q$ (fig. 9). Assignment to $A.hd$ or $A.tl$ may alter the sequence, but the sets of objects which support the description differ between cases, and neither of them is the sequence itself.

Ordered Sequences. The predicate olist_f asserts that a particular sequence of objects is ordered by $(\leq)$ in its *f* component:

$$\text{olist}_f \langle\,\rangle \qquad\qquad \text{olist}_f \langle A \rangle$$

$$\text{olist}_f \big(R \mathbin{@} \langle A, B\rangle \mathbin{@} S\big) \mathrel{\hat{=}} \text{olist}_f \big(R \mathbin{@} \langle A \rangle\big) \wedge A.f \leq B.f \wedge \text{olist}_f \big(\langle B \rangle \mathbin{@} S\big)$$

The sequence of objects S which $\text{olist}_f(S)$ asserts is ordered don't have to be linked together in the heap. An assignment $A.f := E$ doesn't affect the assertion just when $A \notin S$.

If we combine definitions then we have to be careful. The assertion $\text{olist}_{hd}\big(x \Rightarrow_{tl} \text{nil}\big)$ uses both the olist_f and the $\Rightarrow_f$ definitions to state that a particular linked sequence in the heap is ordered: the assertion is unaffected by $A.tl := E$ when $A \notin x \Rightarrow_{tl} \text{nil}$ – because that is the condition which makes sure that the sequence $x \Rightarrow_{tl} \text{nil}$ remains the same – and unaffected by $A.hd := E$ in exactly the same circumstances, even though that assignment doesn't alter the object-sequence itself.

Cyclic Graphs. The set of nodes in a directed binary graph reachable through a pointer *A* is

$$A*_{l,r} \mathrel{\hat{=}} \text{if } A = \text{nil then } \{\,\} \text{ else } \{A\} \cup A.l*_{l,r} \cup A.r*_{l,r} \text{ fi}$$

The assignment $p.l := q$, given the postcondition $p \neq \text{nil} \wedge \mathrm{G}\big(p*_{l,r}\big)$, where G is some predicate, will generate the precondition

$$p \neq \text{nil} \wedge \mathrm{G}\Big(\{p\} \cup q*_{l \oplus p \mapsto q, r} \cup p.r*_{l \oplus p \mapsto q, r}\Big)$$

We can expect our graphs to be cyclic, so the occurrences of the mapping $l \oplus p \mapsto q$ in this formula will give us problems.

In this case it is helpful to rewrite the definition to make spatial separation easier to establish. It's simple to calculate reachable nodes using a directed acyclic graph (DAG), breaking cycles by including an exclusion set *S* in the definition:

$$A*_{l,r,S} = \text{if } A = \text{nil} \vee A \in S \text{ then } \{\,\} \text{ else } \{A\} \cup A.l*_{l,r,S \cup \{A\}} \cup A.r*_{l,r,S \cup \{A\}} \text{ fi}$$

This automatically gives spatial separation between the root node A and the subgraphs generated from the child nodes $A.l$ and $A.r$. Now the assignment $p.l := q$. given the postcondition $p \neq \text{nil} \wedge p \notin S \wedge \mathrm{G}\big(p*_{l,r,S}\big)$, will generate

$$p \neq \text{nil} \wedge p \notin S \wedge \mathrm{G}\Big(\{p\} \cup q*_{l \oplus p \mapsto q, r, S \cup \{p\}} \cup p.r*_{l \oplus p \mapsto q, r, S \cup \{p\}}\Big)$$

When $A \in S$ the graph $B*_{l,r,S}$ won't include A – that is, we have spatial separation. Formally we have the equivalence (provable by induction on the height of finite DAGs)

$$A \in S \rightarrow \left(B*_{l \oplus A \mapsto E,r,S} = B*_{l,r,S} \right)$$

Since p belongs to $S \cup \{p\}$, the precondition can be simplified to

$$p \neq \text{nil} \wedge p \notin S \wedge \mathrm{G}\left(\{p\} \cup q*_{l,r,S\cup\{p\}} \cup p.r*_{l,r,S\cup\{p\}} \right).$$

8 Heap Reasoning and Structural Reasoning

We can make assertions about a value represented by a heap data structure which are about the value itself and which don't involve knowledge of the heap. If it's a sequence we may remark about its length, its non-cyclic nature, the non-occurrence of 0 elements in its *hd* components, and so on. If it's a tree we may want to talk about its height, the balance between its subtrees, the ordering of its tips, and so on. If it's a graph we may remark about its spanning tree, its connectedness, its colouring, and so on. If we are careful, 'pure' structural reasoning can carry much of the weight of a specification and proof. On the other hand, spatial separation concerns require our heap data structure definitions to be written to expose information about the objects that support particular data structures.

It seems, therefore, that specifications ought to be written at three separate levels:

1. Remarks about the contents of particular variables and objects.
2. Inductive definitions of data structures, including rules which interpret non-hiding assertions.
3. The specification of the problem, in terms of (1) and (2).

We have to make our definitions carefully, both to fit our problem and to permit the easiest and most local forms of reasoning. The $\Rightarrow_f$ definition of fig. 5 is good for acyclic sequences, and makes it easy to reason about assignments near the head of a sequence. It's less good (see the list merge example below) when we assign to the tail of a sequence. It's not very good at all if the sequence can be cyclic, and in that case it seems reasonable to devise a special adf which deals easily with cycles.

It is interesting that remarks about individual heap cells – 'pictures of memory' – seem relatively unimportant in this treatment, in contrast to Reynolds [24]. On the other hand, spatial separation of data structures, a clear echo of Burstall's DNRLS [6] and Reynolds' spatial conjunction, is an essential part of every proof.

9 A Worked Example: In-Place List-Reversal

The in-place list-reversal algorithm (fig. 10) is the lowest hurdle that a pointer-aliasing formalism ought to be able to jump. Burstall deals with it in [6] with essentially the same specification as that below.

$$\{Q\}\ r := p;\ p := \text{nil};$$
$$\{P\}\ \text{while } r \neq \text{nil do } q := r;\ r := r.tl;\ q.tl := p;\ p := q \text{ od } \{R\}$$

Fig. 10. The in-place list-reversal algorithm

9.1 Definitions

I have not written inductive definitions of functions for the most part. Instead I give useful facts as axioms.

In the invariant I need append (@) and rev, and I need to be able to say that a particular cell-sequence S is a list: that is, it is finite length and has no repetitions. I can only reverse finite sequences. I need disjointness ($\pitchfork$) of sequences.

$$\langle\,\rangle @ S \mathrel{\hat=} S \qquad S @ \langle\,\rangle \mathrel{\hat=} S \qquad\qquad \text{rev}\,\langle\,\rangle \mathrel{\hat=} \langle\,\rangle \qquad \text{rev}\,\langle A\rangle \mathrel{\hat=} \langle A\rangle$$
$$S @ (T @ U) \mathrel{\hat=} (S @ T) @ U \qquad\qquad \text{list } A \wedge \text{list } B \rightarrow \text{rev}(A @ B) = \text{rev } B @ \text{rev } A$$

$$\langle\,\rangle \pitchfork S \qquad \langle A\rangle \pitchfork \langle B\rangle \mathrel{\hat=} A \neq B \qquad\qquad \text{list}\,\langle\,\rangle \qquad \text{list}\,\langle A\rangle$$
$$S \pitchfork T \mathrel{\hat=} T \pitchfork S \qquad\qquad \text{list}(S @ T) \mathrel{\hat=} \text{list } S \wedge \text{list } T \wedge S \pitchfork T$$
$$(S @ T) \pitchfork U \mathrel{\hat=} (S \pitchfork U) \wedge (T \pitchfork U)$$

For proof of termination of any list algorithm, it's necessary to do arithmetic on lengths of lists. I haven't attempted a mechanical treatment: instead I've appealed to some obvious facts.

$$\text{list } S \rightarrow \text{length}\,(\langle A\rangle @ S) > \text{length } S \qquad \text{length}\,(\langle A\rangle @ S) > 0$$
$$\text{list } S \rightarrow \exists n{:}(n \geq 0 \wedge \text{length } S = n)$$

I use the $\Rightarrow_f$ definition (fig. 5) to describe sequences in the heap. It's possible to prove by induction on the length of finite sequences that

$$\left(\text{list}\left(B \Rightarrow_{tl} C\right) \wedge \langle A\rangle \pitchfork B \Rightarrow_{tl} C\right) \rightarrow \left(B \Rightarrow_{tl \oplus A \mapsto E} C = B \Rightarrow_{tl} C\right)$$

– a fact that is appealed to several times in the proof.

9.2 Specification

The algorithm is given in variable p a pointer to a *tl*–linked list S.

$$Q \mathrel{\hat=} \text{list}\left(p \Rightarrow_{tl} \text{nil}\right) \wedge p \Rightarrow_{tl} \text{nil} = S$$

The invariant of the loop is that there is a *tl*-linked list from p to nil, a similar but distinct list from r to nil, and the reverse of the r list, followed by the p list, is the reverse of the original input.

$$P \mathrel{\hat=} \begin{pmatrix} \text{list}\left(p \Rightarrow_{tl} \text{nil}\right) \wedge \text{list}\left(r \Rightarrow_{tl} \text{nil}\right) \wedge p \Rightarrow_{tl} \text{nil} \pitchfork r \Rightarrow_{tl} \text{nil} \wedge \\ \text{rev}\left(r \Rightarrow_{tl} \text{nil}\right) @ p \Rightarrow_{tl} \text{nil} = \text{rev } S \end{pmatrix}$$

The loop measure is the length of the r list.

$$t \mathrel{\hat=} \text{length}\left(r \Rightarrow_{tl} \text{nil}\right)$$

On termination the p list is the reverse of the original input.

$$R \mathrel{\hat{=}} p \Rightarrow_{tl} \text{nil} = \text{rev}\ S$$

This specification doesn't say that on termination the hd components of S are what they originally were, though it is obvious from the program that this is so. It would be easy to change the specification to make this point by adding $hd = HD$ to precondition, invariant and postcondition, but it would be just a drop in the ocean of the frame problem.

9.3 Semi-Formal Proofs

These proofs are written backwards: postcondition first, working towards precondition and several substitution steps are compressed into one. The machine-checked proofs available from the web site [4], calculated with the aid of Jape [5], are less compressed.

Initialisation.

$\{P\}$

$r := p;\ p := \text{nil}$

[substitution]

$$\left\{\begin{array}{l}\text{list}(p \Rightarrow_{tl} \text{nil}) \wedge \text{list}(\text{nil} \Rightarrow_{tl} \text{nil}) \wedge p \Rightarrow_{tl} \text{nil} \pitchfork \text{nil} \Rightarrow_{tl} \text{nil} \wedge \\ \text{rev}(p \Rightarrow_{tl} \text{nil})@\text{nil} \Rightarrow_{tl} \text{nil} = \text{rev}\ S\end{array}\right\}$$

[replace $\text{nil} \Rightarrow_{tl} \text{nil}$ by $\langle\ \rangle$; then straightforward sequence calculation]

$$\{\text{list}(p \Rightarrow_{tl} \text{nil}) \wedge p \Rightarrow_{tl} \text{nil} = S\}$$

Loop Body Preserves Invariant

$\{P\}$

$q := r;\ r := r.tl;\ q.tl := p;\ p := q$

[substitution]

$$\left\{\begin{array}{l}\text{list}(r.tl \Rightarrow_{tl \oplus r \mapsto p} \text{nil}) \wedge \text{list}(r \Rightarrow_{tl \oplus r \mapsto p} \text{nil}) \wedge r.tl \Rightarrow_{tl \oplus r \mapsto p} \text{nil} \pitchfork r \Rightarrow_{tl \oplus r \mapsto p} \text{nil} \wedge \\ \text{rev}(r.tl \Rightarrow_{tl \oplus r \mapsto p} \text{nil})@r \Rightarrow_{tl \oplus r \mapsto p} \text{nil} = \text{rev}\ SS\end{array}\right\}$$

[Use $r \neq \text{nil}$ and $\text{list}(r \Rightarrow_{tl} \text{nil})$ from invariant to obtain $\text{list}(\langle r\rangle @ r.tl \Rightarrow_{tl} \text{nil})$; thence $\text{list}(r.tl \Rightarrow_{tl} \text{nil})$ and $\langle r\rangle \pitchfork r.tl \Rightarrow_{tl} \text{nil}$, and therefore $r.tl \Rightarrow_{tl \oplus r \mapsto p} \text{nil} = r.tl \Rightarrow_{tl} \text{nil}$. Use $r \neq \text{nil}$ and $p \Rightarrow_{tl} \text{nil} \pitchfork r \Rightarrow_{tl} \text{nil}$ to obtain $p \Rightarrow_{tl} \text{nil} \pitchfork (\langle r\rangle @ r.tl \Rightarrow_{tl} \text{nil})$; thence $\langle r\rangle \pitchfork p \Rightarrow_{tl} \text{nil}$ and since $\text{list}(p \Rightarrow_{tl} \text{nil})$, $r \Rightarrow_{tl \oplus r \mapsto p} \text{nil} = \langle r\rangle @ p \Rightarrow_{tl} \text{nil}$. Once all the assignment-induced mappings are eliminated, it's a straightforward sequence calculation.]

$$\left\{\begin{array}{l}\text{list}(p \Rightarrow_{tl} \text{nil}) \wedge \text{list}(r \Rightarrow_{tl} \text{nil}) \wedge p \Rightarrow_{tl} \text{nil} \pitchfork r \Rightarrow_{tl} \text{nil} \wedge \\ \text{rev}(r \Rightarrow_{tl} \text{nil})@p \Rightarrow_{tl} \text{nil} = \text{rev}\ S \wedge r \neq \text{nil}\end{array}\right\}$$

```
$\{Q\}$
if $q = \text{nil} \vee\!\!\vee (p \neq \text{nil} \wedge\!\!\wedge p.hd \leq q.hd)$ then $r := p;\ p := p.tl$ else $r := q;\ q := q.tl$ fi; $s := r$
$\{P\}$
while $p \neq \text{nil} \vee q \neq \text{nil}$ do
   if $q = \text{nil} \vee\!\!\vee (p \neq \text{nil} \wedge\!\!\wedge p.hd \leq q.hd)$ then $s.tl := p;\ p := p.tl$ else $s.tl := q;\ q := q.tl$ fi;
   $s := s.tl$
od
$\{R\}$
```

Fig. 11. The in-place list merge algorithm

Loop Body Reduces Measure

$\{\text{length}(r \Rightarrow_{tl} \text{nil}) < vt\}$
$q := r;\ r := r.tl;\ q.tl := p;\ p := q$

[substitution]

$\{\text{length}(r.tl \Rightarrow_{tl \oplus r \mapsto p} \text{nil}) < vt\}$

[use $r \neq \text{nil}$ and $\text{list}(r \Rightarrow_{tl} \text{nil})$ from precondition to obtain $\langle r \rangle \pitchfork r.tl \Rightarrow_{tl} \text{nil}$; thence $r.tl \Rightarrow_{tl \oplus r \mapsto p} \text{nil} = r.tl \Rightarrow_{tl} \text{nil}$; then straightforward sequence calculation]

$$\left\{\begin{array}{l} \text{list}(p \Rightarrow_{tl} \text{nil}) \wedge \text{list}(r \Rightarrow_{tl} \text{nil}) \wedge p \Rightarrow_{tl} \text{nil} \pitchfork r \Rightarrow_{tl} \text{nil} \wedge \\ \text{rev}(r \Rightarrow_{tl} \text{nil}) @ p \Rightarrow_{tl} \text{nil} = \text{rev}\ S \wedge r \neq \text{nil} \wedge \text{length}(r \Rightarrow_{tl} \text{nil}) = vt \end{array}\right\}$$

The While Loop

$\{P\}$ while $r \neq \text{nil}$ do $q := r;\ r := r.tl;\ q.tl := p;\ p := q$ od $\{R\}$

Not shown: it appeals to the invariant and measure proofs above, and is otherwise straightforward sequence calculation.

The Whole Algorithm

$\{Q\}$ $r := p;\ p := \text{nil};$ while $r \neq \text{nil}$ do $q := r;\ r := r.tl;\ q.tl := p;\ p := q$ od $\{R\}$

Trivial, given the previous proofs.

10 An Illustration: In-Place List Merge

The proof of in-place list reversal above is no advance on Burstall's version [6]. In-place list merge (fig. 11) is a more challenging problem, because it works at the tail end of a list. The tests in the program are designed to avoid traversal of nil pointers: $A \vee\!\!\vee B$ is equivalent to (if A then true else B fi); $A \wedge\!\!\wedge B$ is equivalent to (if A then B else false fi).

The program is given two disjoint ordered *tl*-linked nil-terminated lists via pointers *p* and *q*, at least one of which is not empty.

$$Q \mathrel{\hat=} \begin{pmatrix} p \Rightarrow_{tl} \text{nil} = SP \wedge q \Rightarrow_{tl} \text{nil} = SQ \wedge \text{olist}_{hd}(p \Rightarrow_{tl} \text{nil}) \wedge \text{olist}_{hd}(q \Rightarrow_{tl} \text{nil}) \wedge \\ p \Rightarrow_{tl} \text{nil} \pitchfork q \Rightarrow_{tl} \text{nil} \wedge (p \neq \text{nil} \vee q \neq \text{nil}) \end{pmatrix}$$

Writing $A \Rightarrow_f^+ B$ for $A \Rightarrow_f B @\langle B\rangle$, the invariant states that we can put the list fragment $r \Rightarrow_{tl}^+ s$ before either the *p* list or the *q* list to produce an ordered result; that fragment joined with the two lists is a permutation of the original input; and the *p* and *q* lists remain disjoint. In order to prove that the result on termination is what we wish, it also has to assert the non-occurrence of nil in the fragment $r \Rightarrow_{tl}^+ s$ and a linkage between the *s* object and the one or other of the lists.

$$P \mathrel{\hat=} \begin{pmatrix} \text{olist}_{hd}\left(r \Rightarrow_{tl}^+ s @ p \Rightarrow \text{nil}\right) \wedge \text{olist}_{hd}\left(r \Rightarrow_{tl}^+ s @ q \Rightarrow \text{nil}\right) \wedge \\ \text{perm}\left(r \Rightarrow_{tl}^+ s @ p \Rightarrow \text{nil} @ q \Rightarrow \text{nil}, SP @ SQ\right) \wedge \\ p \Rightarrow \text{nil} \pitchfork q \Rightarrow \text{nil} \wedge \text{nil} \notin r \Rightarrow_{tl}^+ s \wedge (s.tl = p \vee s.tl = q) \end{pmatrix}$$

The measure is the sum of the lengths of the *p* and *q* lists.

$$t \mathrel{\hat=} \text{length}(p \Rightarrow_{tl} \text{nil} @ q \Rightarrow_{tl} \text{nil})$$

On termination *r* points to an ordered *tl*-linked nil-terminated list which is a permutation of the original input.

$$R \mathrel{\hat=} \text{olist}_{hd}\left(r \Rightarrow_{tl} \text{nil}\right) \wedge \text{perm}\left(r \Rightarrow_{tl} \text{nil}, SP @ SQ\right)$$

Problems arise during the proof in showing and exploiting spatial separation between the input lists $p \Rightarrow_{tl} \text{nil}$ and $q \Rightarrow_{tl} \text{nil}$ on the one hand, and the list fragment $r \Rightarrow_{tl}^+ s$ on the other. For example, moving the invariant backward through the sequence $s.tl := p; p := p.tl; s := s.tl$ – the course of the loop, given that the guard in the if-then-else-fi is true – produces several instances of the fearsome formula $r \Rightarrow_{tl \oplus s \mapsto p}^+ p$. Given

$$\text{list}\left(r \Rightarrow_{tl}^+ s\right),\ p \neq \text{nil},\ p \notin r \Rightarrow_{tl}^+ s$$

– each of which is implied by the invariant plus the if guard – then a bit of finite expansion, plus the fact that

$$A \Rightarrow_{f \oplus B \mapsto E}^+ B = A \Rightarrow_f^+ B$$

makes it possible to show that

$$r \Rightarrow_{tl \oplus s \mapsto p}^+ p = r \Rightarrow_{tl}^+ s @\langle p\rangle$$

The rest of the proof is straightforward manipulation.

11 An Illustration: Graph Marking

The Schorr-Waite algorithm is the first mountain that any formalism for pointer aliasing should climb. It's dealt with semi-formally by Morris [21] and mechanically by Suzuki [27]; Kowaltowski [17] gives a wonderful informal proof in pictures of a

```
$\{Q\}$
$t := root;\ p := \text{nil}$
$\{P\}$
while $p \neq \text{nil} \vee (t \neq \text{nil} \wedge\!\!\wedge \neg t.m)$ do
  if $t = \text{nil} \vee\!\!\vee t.m$ then
    if $p.c$ then
      $q := t;\ t := p;\ p := p.r;\ t.r := q$ /* POP */
    else
      $q := t;\ t := p.r;\ p.r := p.l;\ p.l := q;\ p.c := \text{true}$ /* SWING */
    fi
  else
    $q := p;\ p := t;\ t := t.l;\ p.l := q;\ p.m := \text{true};\ p.c := \text{false}$ /* PUSH */
  fi
od
$\{R\}$
```

Fig. 12. The Schorr-Waite graph-marking algorithm

tree-marking variant. Morris's version of the algorithm, modified to permit nil pointers, is shown in fig. 12.

Definitions. The stack starting at *A*:

$$A\uparrow_{l,r,c} \mathrel{\hat{=}} \text{if } A = \text{nil then } \langle\ \rangle \text{ else } \langle A\rangle @ \left(\text{if } A.c \text{ then } A.r \text{ else } A.l \text{ fi}\uparrow_{l,r,c}\right) \text{ fi}$$

The binary DAG reachable from *A* but containing none of the objects in set *S*:

$$A*_{l,r,S} = \text{if } A = \text{nil} \vee A \in S \text{ then } (\ ) \text{ else } \left(A, A.l*_{l,r,S\cup\{A\}}, A.r*_{l,r,S\cup\{A\}}\right) \text{ fi}$$

The binary DAG reachable from *A* which contains only unmarked nodes and none of the objects in set *S*:

$$A**_{l,r,m,S} \mathrel{\hat{=}} \begin{pmatrix} \text{if } A = \text{nil} \vee A \in S \vee A.m \text{ then } (\ ) \\ \text{else } \left(A, A.l**_{l,r,m,S\cup\{A\}}, A.r**_{l,r,m,S\cup\{A\}}\right) \text{ fi} \end{pmatrix}$$

Two sequences zipped together:

$$\langle\ \rangle \,|||\, S \mathrel{\hat{=}} \langle\ \rangle \qquad R \,|||\, \langle\ \rangle \mathrel{\hat{=}} \langle\ \rangle$$
$$(\langle A\rangle @ R) \,|||\, (\langle B\rangle @ S) \mathrel{\hat{=}} \langle (A,B)\rangle @ \left(R \,|||\, S\right)$$

Converting sequences and DAGs to sets:

$$\text{set}\langle\ \rangle \mathrel{\hat{=}} \{\ \} \qquad \text{set}(\ ) \mathrel{\hat{=}} \{\ \}$$
$$\text{set}\langle A\rangle \mathrel{\hat{=}} \{A\} \qquad \text{set}(A, T1, T2) \mathrel{\hat{=}} \{A\} \cup \text{set } T1 \cup \text{set } T2$$
$$\text{set}(R @ S) \mathrel{\hat{=}} \text{set } R \cup \text{set } S$$

Specification. The graph reachable from the root is *iG*. The whole of the heap is unmarked. Initial values of the control-bit mapping, the left and right subnode mappings, are *iC*, *iL* and *iR*.

$$Q \mathrel{\hat{=}} root^{*}_{l,r,\{\,\}} = iG \wedge \forall x{:}\, \neg x.m \wedge c = iC \wedge l = iL \wedge r = iR$$

Throughout the marking process the stack *p* is a list, and every node on the stack is marked. Every node in the original graph is reachable from the tip *t* and/or the stack. Unmarked nodes in the graph are reachable from the tip and/or the right sub-nodes of elements of the stack. If a node is marked then it's in the graph; if it's not marked then its control bit is unchanged; if it's not on the stack then its left and right components are as they were originally. If a node is on the stack then we can reconstruct its left and right components by considering its predecessor node and its control bit.

$$P \mathrel{\hat{=}} \left(\begin{array}{l} list\left(p\uparrow_{l,r,c}\right) \wedge \forall x{:}\left(x \in \mathrm{set}\left(p\uparrow_{l,r,c}\right) \rightarrow x.m\right) \wedge \\ \mathrm{set}\left(t*_{l,r,\{\,\}}\right) \cup \mathrm{set}\left(p*_{l,r,\{\,\}}\right) = \mathrm{set}\, iG \wedge \forall x{:}\left(x.m \rightarrow x \in \mathrm{set}\, iG\right) \wedge \\ \forall x{:}\left(\begin{array}{l} x \in \mathrm{set}\, iG \wedge \neg x.m \rightarrow \\ x \in \left(\mathrm{set}\left(t**_{l,r,m,\{\,\}}\right) \cup \bigcup\left\{\mathrm{set}\left(y.r**_{l,r,m,\{\,\}}\right) \,\middle|\, x \in \mathrm{set}\left(p\uparrow_{l,r,c}\right)\right\}\right) \end{array}\right) \wedge \\ \forall x{:}\left(\left(\neg x.m \rightarrow x.c = x.iC\right) \wedge \left(x \notin p\uparrow_{l,r,c} \rightarrow x.l = x.iL \wedge x.r = x.iR\right)\right) \wedge \\ \forall x, y{:}\left(\begin{array}{l} (x, y) \in \left(p\uparrow_{l,r,c} \,|||\, \left(\langle t\rangle \mathbin{@} p\uparrow_{l,r,c}\right)\right) \rightarrow \\ \text{if } x.c \text{ then } x.l = x.iL \wedge y = x.iR \text{ else } y = x.iL \wedge x.r = x.iR \text{ fi} \end{array}\right) \end{array}\right)$$

The measure of the loop is a triple of the number of unmarked nodes in the graph, the number of nodes on the stack with control bit set to true, and the length of the stack.

On termination all and only the original graph is marked. Unmarked nodes have an unchanged control bit, and the left and right components are as they were on input.

$$R \mathrel{\hat{=}} \forall x{:}\left(\left(x.m \leftrightarrow x \in \mathrm{set}\, iG\right) \wedge \left(\neg x.m \rightarrow x.c = x.iC\right) \wedge x.l = x.iL \wedge x.r = x.iR\right)$$

Proof. DAGs based on the definitions above are finite height, given a finite heap (provable by induction on the difference between the restriction set *S* and the heap). That permits inductive proof of all kinds of interesting equivalences, including, for example

$$B \in S \rightarrow \left(A**_{l \oplus B \mapsto E, r, m, S} = A**_{l,r,m,S}\right)$$

$$T \subseteq S \rightarrow \left(A*_{l,r,m,S} \cup B*_{l,r,m,T} = A*_{l,r,m,S \cup \{B\}} \cup B*_{l,r,m,T}\right)$$

$$E \rightarrow \left(A**_{l,r,m \oplus B \mapsto E, S} = A**_{l,r,m,S \cup \{B\}}\right)$$

The marvellous character of the algorithm is not that it reaches every node – a simple recursion would do that – but that it does so without using a stack, modifying nodes it has passed through and – most marvellously of all – restoring them afterwards. Corresponding parts of the proof are large. We have to prove in the SWING arm, for example, that

$$
\begin{gathered}
p \neq nil, \quad \neg p.c, \quad \forall x{:}\left(x \in \mathrm{set}\left(p\uparrow_{l,r,c}\right) \rightarrow x.m\right), \quad list\left(p.l\uparrow_{l,r,c}\right), \quad \langle p\rangle \pitchfork p.l\uparrow_{l,r,c}, \\
\forall x{:}\left(\begin{array}{l}
(\neg x.m \rightarrow x.c = x.iC) \wedge \left(x \notin \mathrm{set}\left(p\uparrow_{l,r,c}\right) \rightarrow (x.l = x.iL \wedge x.r = x.iR)\right) \wedge \\
\forall y{:}\left(\begin{array}{l}
(x,y) \in \mathrm{set}\left(p\uparrow_{l,r,c} \mathbin{|||} \left(\langle \mathrm{t}\rangle @ p\uparrow_{l,r,c}\right)\right) \rightarrow \\
\text{if } x.c \text{ then } x.l = x.iL \wedge y = x.iR \text{ else } y = x.iL \wedge x.r = x.iR \text{ fi}
\end{array}\right)
\end{array}\right) \\
\vdash \forall x{:}\left(\begin{array}{l}
\left(\neg x.m \rightarrow x.(c \oplus p \mapsto \mathrm{true}) = x.iC\right) \wedge \\
\left(\begin{array}{l}
x \notin \mathrm{set}\left(p\uparrow_{l,r,c}\right) \rightarrow \\
\left(x.(l \oplus p \mapsto t) = x.iL \wedge x.(r \oplus p \mapsto p.l) = x.iR\right)
\end{array}\right) \wedge \\
\forall y{:}\left(\begin{array}{l}
(x,y) \in \mathrm{set}\left(p\uparrow_{l,r,c} \mathbin{|||} \left(\langle p.r\rangle @ p\uparrow_{l,r,c}\right)\right) \rightarrow \\
\left(\begin{array}{l}
\text{if } x.(c \oplus p \mapsto \mathrm{true}) \text{ then } x.(l \oplus p \mapsto t) = x.iL \wedge y = x.iR \\
\text{else } y = x.iL \wedge x.(r \oplus p \mapsto p.l) = x.iR \text{ fi}
\end{array}\right)
\end{array}\right)
\end{array}\right)
\end{gathered}
$$

It's for the most part straightforward, but it's tedious manipulation. In Burstall's phrase, a great deal of work for such a simple matter.

From the DAG definition it's possible to prove that

$$t*_{l,r,S\cup\{p\}} \cup p*_{l,r,S} = t*_{l,r,S} \cup p*_{l,r,S} = t*_{l,r,S} \cup p*_{l,r,S\cup\{t\}}$$

That gives enough spatial separation to make it easy to prove that the nodes of the original graph are always reachable. It's a little harder to show that the unmarked nodes are always directly reachable: we have to show when dealing with the PUSH arm, for example, that

$$
\begin{gathered}
t \neq nil, \neg t.m, list\left(p\uparrow_{l,r,c}\right), \\
\forall x{:}\left(\begin{array}{l}
x \in \mathrm{set}\, iG \wedge \neg x.m \rightarrow \\
x \in \left(\mathrm{set}\left(t**_{l,r,m,\{\,\}}\right) \cup \bigcup\left\{\mathrm{set}\left(y.r**_{l,r,m,\{\,\}}\right) \,\middle|\, x \in p\uparrow_{l,r,c}\right\}\right)
\end{array}\right) \\
\vdash \forall x{:}\left(\begin{array}{l}
x \in \mathrm{set}\, iG \wedge \neg x.(m \oplus t \mapsto true) \rightarrow \\
x \in \left(\begin{array}{l}
\mathrm{set}\left(t.l**_{l\oplus t\mapsto p,r,m\oplus t\mapsto true,\{\,\}}\right) \cup \\
\bigcup\left\{\mathrm{set}\left(y.r**_{l\oplus t\mapsto p,r,m\oplus t\mapsto true,\{\,\}}\right) \,\middle|\, x \in p\uparrow_{l\oplus t\mapsto p,r,c}\right\}
\end{array}\right)
\end{array}\right)
\end{gathered}
$$

The proof that the stack is invariantly a list sometimes has to deal with some superficially frightening formulæ. The SWING arm, for example, generates

$$p\uparrow_{l\oplus p\mapsto t,r\oplus p\mapsto p.l,c\oplus p\mapsto true}$$

but, with a single step of expansion, plus $p.c$ from the guard and $p \neq \mathrm{nil}$ and $\mathrm{list}\left(p\uparrow_{l,r,c}\right)$ from the invariant, this simplifies to $p\uparrow_{l,r,c}$.

12 Conclusion

Burstall showed a way towards practical proofs of pointer programs which relatively few have followed. This paper shows that it is possible to reason in Hoare logic about small but moderately complicated pointer programs, using the principle of spatial separation and whatever data structure and other auxiliary definitions suit the problem.

Burstall’s DNLRS mechanism achieved local reasoning by restricting itself to particular data structures and particular problems. The treatment of general data structures in this paper doesn’t yet approach the elegance of his solution: it substitutes first, tidies up second, and remains rather low-level. To make it more elegant and more practically useful, it will be necessary to make the substitution mechanism mimic the locality of assignment, dealing only with genuine potential aliases and ignoring those which can be dealt with by spatial separation assumptions. If we can do this for a wide range of problems, building on a relatively small collection of data structure inductions, this goal may perhaps be reached.

Acknowledgements. This work springs from discussions with (in temporal order) Keith Clarke, Peter O’Hearn, Dave Streader, Stefano Guerrini, Samin Ishtiaq, Hayo Thielecke, Edmund Robinson and Cristiano Calgagno[1] at QMW, fuelled by the QMW Verified Bytecode project [EPSRC GR/L54578]. Tony Hoare first pointed me to Morris’s work, and John Reynolds emphasised the importance of Burstall’s. Uday Reddy and Richard Bird impressed upon me the usefulness of the component-as-array trick. Jape is a joint effort with Bernard Sufrin, and the developments which support the Jape-rendered proofs in this document arose out of discussions between us.

References

[1] Appel A.W., Felty A.P.: A Semantic Model of Types and Machine Instructions for Proof-Carrying Code 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '00), pp. 243-253, January 2000.

[2] Arnold K., Gosling, J.: The Java programming language. Addison-Wesley, 1997.

[3] Bijlsma A.: Calculating with Pointers. Science of Computer Programming **12** (1989) 191-205

[4] Bornat R., Machine-checked proofs of list reversal, list merge, Schorr-Waite. Available from http://www.dcs.qmw.ac.uk/~richard/pointers

[5] Bornat R., Sufrin B.A.: Animating formal proof at the surface: the Jape proof calculator. The Computer Journal, 43, 3, 1999, 177-192.

[6] Burstall R.M.: Some techniques for proving correctness of programs which alter data structures. Machine Intelligence 7, D. Michie (ed), American Elsevier, 1972, 23-50.

[7] Cousot P.: Methods and Logics for Proving Programs. In Formal Models and Semantics, J. van Leeuwen (ed.); Volume B of Handbook of Theoretical Computer Science. Elsevier (1990) 843--993

[1] Calcagno was on loan from Genova.

[8] Floyd R.W.: Assigning meaning to programs. Proc. Symp. in App. Math., American Mathematical Society, Vol.19, 1967, 19-32.

[9] Gries D.: The Schorr-Waite graph marking algorithm. Acta Informatica Vol 11, 1979, 223-232.

[10] Gries D., Levin G.M.: Assignment and procedure call proof rules. ACM Transactions on Programming Languages and Systems **2** (1980) 564-579

[11] Hoare C.A.R.: An axiomatic basis for computer programming, Comm. ACM Vol 12, no 10, 1969, 576-580 and 583.

[12] Hoare C.A.R., Jifeng H.: A Trace Model for Pointers and Objects. Object-Oriented Programming 13th European Conference (ECOOP '99), Lecture Notes in Computer Science, Vol. 1628. Springer Verlag (1999) 1-17

[13] Hoare C.A.R., Wirth N.: An Axiomatic Definition of the Programming Language Pascal. Acta Informatica **2** (1973) 335-355

[14] Jensen K., Wirth N.: Pascal user manual and report. Springer-Verlag (1975)

[15] Kernighan B.W., Ritchie D.M.: The C programming language. Prentice-Hall (1978)

[16] Kowaltowski T.: Data Structures and Correctness of Programs. Journal of the ACM **2**, (1979) 283-301

[17] Kowaltowski T.: Examples of Informal but Rigorous Correctness Proofs for Tree Traversing Algorithms. Technical report TR-DCC-92-10, University of Campinas, Brazil.

[18] Leino R.: Toward Reliable Modular Programs. PhD Thesis, California Institute of Technology (1995)

[19] Luckham D.C., Suzuki N.: Verification of Array, Record, and Pointer Operations in Pascal. ACM Transactions on Programming Languages and Systems, Examples of Informal but Rigorous Correctness Proofs for Tree Traversing Algorithms **1** (1979),226-244

[20] McCarthy J., Painter J.A.: Correctness of a Compiler for Arithmetic Expressions. Proceedings Symposium in Applied Mathematics, Mathematical Aspects of Computer Science **19** (1967) 33-41

[21] Morris J.M. A general axiom of assignment. Assignment and linked data structure. A proof of the Schorr-Waite algorithm. In: Theoretical Foundations of Programming Methodology (Proceedings of the 1981 Marktoberdorf Summer School), M Broy and G. Schmidt (eds.) , Reidel (1982) 25-51

[22] Necula G. and Lee P.: Safe, Untrusted Agents using Proof-Carrying Code. Mobile Agents and Security, Lecture Notes in Computer Science, Vol. 1419. Springer-Verlag (1998) 61-91

[23] Reynolds J.C.: The Craft of Programming. Prentice-Hall International (1981)

[24] Reynolds J.C.: Reasoning about Shared Mutable Data Structure, given at the symposium in celebration of the work of C.A.R. Hoare, Oxford, September 1999.

[25] Schorr H., Waite W.M.: An efficient machine-independent procedure for garbage collection in various list structures. Comm. ACM **10** (1967) 501-506

[26] Strachey C.: Towards a Formal Semantics. In: Formal Language Description Languages for Computer Programming, T.B. Steel Jr (ed). North-Holland (1964).

[27] Suzuki N.: Automatic verification of programs with complex data structure. PhD Thesis, Stanford U (1976)

[28] Topor R.W.: The correctness of the Schorr-Waite list marking algorithm. Acta Informatica **11** (1979) 211-221