Local reasoning, separation and aliasing

Richard Bornat
School of Computing Science
Middlesex University.
R.Bornat@mdx.ac.uk

Cristiano Calcagno
Department of Computing
Imperial College
University of London
ccris@doc.ic.ac.uk

Peter O'Hearn
Department of Computer Science
Queen Mary College
University of London
ohearn@dcs.qmul.ac.uk

Abstract

Structures built by pointer aliasing, such as DAGs and graphs, are notoriously tricky to deal with. The mechanisms of separation logic can deal with these structures, but so far this has been done by the maintenance of a global invariant. Specifications and proofs which use local reasoning, and which may point the way to a structured programming for pointers, are discussed. An idiom for inclusion sharing, where one structure is included in another, is presented. A notion of 'partial graphs' – graphs with dangling pointers – is used to facilitate proof.

1 Introduction

The purpose of a formal logic is to codify reasoning, to enable simple mechanical steps that correspond to reliable but intricate reasoning in a semantic domain. It's reasonable to demonstrate that a novel logic helps us to solve old problems in new ways. A sound logic, however, will prove nothing that is not already a semantic consequence so in a sense it adds nothing to knowledge: all that you can prove with it you could already prove without it. In that sense it's futile to demonstrate its utility, but it's still important to try, because ease of reasoning matters.

The notion of structured programming is inextricably intertwined with the properties of formal logics for reasoning about programs. Proofs of structured programs came apart at the same seams as the programs themselves, and composed likewise. Build your programs from these components in this way, the message went, and it will be easier to build a correct program. It was so effective that nowadays it's hard to find programmers who don't write structured programs. Designers of novel logics can quite reasonably be asked if their work will lead to similarly powerful reasoning and similarly effective advice.

Separation logic [9, 4, 5, 8], which derives from BI [6, 7], aims to simplify reasoning about programs that mutate data structures in a heap (pointer programs for short). Pointers have been thought of as the gotos of data structuring, and so they can be, but if we can organise things so that our data structures and our programs come apart neatly in similar ways – structured reasoning for the heap – we shall perhaps be able to use pointers reliably. Separation logic facilitates *local reasoning* about parts of data structures: if an assignment or a procedure call alters only one corner of one data structure, then we can easily express and exploit the fact that it doesn't alter any other part of it nor any part of any separate data structure. The advantages can be seen most clearly in Yang's proof of the Schorr-Waite algorithm [12, 13], which elegantly avoids the struggles of [1] to show the same result using Hoare logic and array assignment. Apart from that work and Torp-Smith's proof of the Cheney algorithm [11], most work using separation logic has focussed on lists and trees, where separation is easily described.

In this paper we discuss three formal proofs[1] of algorithms which deal with internal sharing of structure; two of the proofs deal with aliasing as well. Our aim has been to make the reasoning as local as possible. Although we can see considerable possibilities for further improvement, we can already claim significant success in extending the range of separation logic to new and difficult areas.

2 A brief introduction to separation logic

Assignment to a variable affects just that variable. Hoare's variable assignment rule

$$\{R^x_E\}\, x := E\, \{R\}$$

elegantly captures that property in a substitution that affects only one variable. The rule supports local reasoning because substitution has no effect on any other variable. Assignment to an array element affects just that element, but Hoare's array element assignment rule

$$\{R^a_{a\oplus(I:E)}\}\, a[I] := E\, \{R\}$$

isn't correspondingly local. Instead, the assignment affects every formula that mentions the array, not merely those which mention the assigned element. As a consequence, the difficulty of reasoning in Hoare logic about programs that use arrays revolves around the arithmetic of array indexing. A program heap is a giant array, and is usually sparse. Reasoning about heaps in Hoare logic is just too hard; if we are to do the job, we must find an easier way.

A program's heap contains cells addressed by integers (not necessarily a contiguous range of addresses, nor necessarily starting at 0). Heap cells contain values. In particular, they may contain integers

[1]The proofs have not yet been mechanically checked, but we have used formal reasoning, without recourse to the semantics, throughout.

which should be read as pointers, and thus we can have recursive and cyclic data structures.

Heap space can be acquired and, once acquired, released. The heap a program is allowed to use is limited to what was provided when execution starts plus what was acquired during execution and not yet released. It follows that only some integers point into the heap, while others are 'dangling pointers'. Dereferencing a dangling pointer is always a mistake and can be a disaster; for simplicity we suppose that it always crashes the program. Avoiding such crashes is one of the practical difficulties of programming with pointers. Dangling pointers are not a difficulty for separation logic, however: there's a real sense in which they are the point, the engine that makes the reasoning tick.

A program has a 'stack' of named variables (think: collection of registers) outside the heap. There are no address operations, implicit or explicit, on the stack, so aliasing within it is impossible. Variables may contain integers and thus point into the heap (or dangle).

Our programming language is '*while*' programs with recursive procedures, using a restricted range of assignment instructions, intended to correspond closely to the operation of a conventional von Neumann machine. We use Hoare triples $\{Q\}C\{R\}$ to specify partial correctness: program C, started in a state (heap + stack) which satisfies Q, is guaranteed (i) not to crash by referencing a heap cell outside its footprint and (ii) if it terminates, to arrive at a state satisfying R. The novelty of the separation logic approach is the interpretation of formulas describing the heap in Q and R.

The simplest heap is **emp**, which contains no cells and allows no dereferencing at all. A single-cell heap, for example $x \mapsto V$, allows dereferencing and update of just one location. In this heap $[x]$ (read 'contents of x') gives the value in the heap cell indexed by x, and delivers V; the instruction $[x] := E$ (see below) updates that same cell, and after it the heap is $x \mapsto E$. The construction $x \mapsto \_$ is a useful shorthand for $\exists v \cdot x \mapsto v$.

The $(\star)$ connective deals with *disjoint* heaps (heaps with disjoint address domains). $A \star B$ holds of a heap just when it can be divided into two disjoint heaps, where simultaneously A holds in one and B in the other. $(\star)$ is associative and commutative; in BI it's a multiplicative version of conjunction. As you would expect, $A \star \mathbf{emp} = \mathbf{emp} \star A = A$. You can deduce disjointness of address domains when heaps are combined: for example $(x \mapsto \_ \star y \mapsto \_) \vdash x \neq y$, and $x \mapsto V \star x \mapsto W$ must be false whatever the values of V and W, since the integer x can't point to two separate cells at the same time. Note that disjoint heaps may contain dangling pointers across the divide, so $x \mapsto y \star y \mapsto x$ is acceptable, but $x \mapsto [y] \star y \mapsto x$ is invalid because the heap $x \mapsto [y]$ refers to a cell outside its domain.

Address arithmetic is allowed, since pointers are really just integers. In particular the construction $x \mapsto V_1, V_2, ..., V_n$ is shorthand for $x \mapsto V_1 \star x+1 \mapsto V_2 \star ... \star x+n-1 \mapsto V_n$. In our programs we are content to use field-access formulae $x.\mathit{field}$ as shorthand for $[x+K]$ for some agreed fixed integer K.

The $(\wedge)$ connective deals with *identical* heaps. $A \wedge B$, normal additive conjunction, holds of a heap if A holds of the whole, and similarly B. In proofs below we make use of the construction $(A \star \text{True}) \wedge B$: True holds of any heap, so this says that A holds in part of the heap, and B holds in the whole, making a kind of subheap relation. We can also use $A \wedge B$ if either A or B is 'pure', saying nothing about the heap at all.

$(-\!\!\star)$ is the 'magic wand' connective. $A -\!\!\star B$ is true of a heap if, whenever it is joined to a disjoint heap which makes A true, B holds in the result. In the particular case that there is only one heap which satisfies A, $A -\!\!\star B$ describes a B-heap with an A-shaped hole. It is a multiplicative form of implication, and $A \star (A -\!\!\star B) \vdash B$.

In $\{Q\}C\{R\}$, C starts in a state satisfying Q and generates a state satisfying R. Q describes the whole heap used by C, unless it gets some more from a resource-generator like 'new'. There are also resource-destructors like 'dispose', and thus it follows that R may describe a larger or a smaller heap than Q does. Because of this resource interpretation, and in contrast to our conventional treatment of assignment to stack variables, we can read Q as circumscribing C's initial footprint in the heap.

The most important inference rule is the Frame Rule, which allows us to localise reasoning about resources (it requires a proviso that C doesn't assign stack variables free in P because of our conventional treatment of stack-variable assignment).

$$\frac{\{Q\}C\{R\}}{\{P \star Q\}C\{P \star R\}} \ \left(\mathit{modifies}\ C \cap \mathit{vars}\ P = \emptyset\right)$$

If C doesn't go outside the heap described by Q, and if P describes a disjoint heap, then clearly C leaves the heap described by P unchanged. In effect we can pull Q out of the precondition, convert it to R with C, and plug R back in to the postcondition.

We recognise three forms of assignment. It may seem irksome to have a limited collection, but it's easy to see that more complicated constructions can be compiled down into the permitted forms by using extra assignments to stack variables. Assignment to stack variables is treated as in ordinary Hoare logic. In these rules E is a 'pure' formula which doesn't refer to the heap. Because of the frame rule, assignments that affect the heap can be defined by giving a minimum resource context. (The naming restrictions on variable assignment can be overcome with $\exists$ bindings of initial values.)

$$\begin{array}{rll} \{R^x_E\} & x{:=}E & \{R\} \\ \{x \mapsto \_\} & [x]{:=}E & \{x \mapsto E\} \\ \{y \mapsto E\} & x{:=}[E'] & \{y \mapsto E \wedge x = E'\}\ (x \text{ not free in } E, E') \end{array}$$

We can treat directly the kind of primitive storage allocation which underlies more sophisticated forms.

$$\begin{array}{rcl} \{\mathbf{emp}\} & x := \text{new}(E) & \{x \mapsto E\} \\ \{x \mapsto \_\} & \text{dispose}(x) & \{\mathbf{emp}\} \end{array}$$

These pre- and post-conditions give minimum resource contexts. Together with the frame rule they tell us that new delivers a pointer which is distinct from any in use in any other context (but not necessarily distinct from any other which may have been disposed in the past, nor from any integer values which may be lying around in stack or heap) and that dispose denies us future access to a currently-accessible cell.

In reasoning about procedures we follow [5]. Our procedures don't alter any of their arguments, and they have no free variables. Where we have written an assignment to a variable that isn't in the procedure's parameter list, for example $l := ...$ in (4), the cautious reader should read an automatic local variable definition. In dealing with recursive steps, we have appealed to a instance of the specification as an inductive hypothesis. We haven't given proofs of termination of any of the procedures, but structural induction arguments based on the formula that is represented in the heap (see section 4 below) would seem to be straightforward.

3 Separation and aliasing

The major practical difficulty of reasoning in a conventional Hoare logic about programs that mutate arrays is array-index aliasing. Different index formulae denote the same array element, and formulae change their reference as the values of variables change. Program proofs must therefore focus on index formula arithmetic, which is difficult to deal with logically.

The message of separation logic is that in dealing with programs that mutate data structures interconnected by pointers, separation trumps aliasing. The list, tree and graph algorithms investigated in [1], for example, depend on separation between structures. To verify those algorithms with array-index reasoning, it's necessary to describe that separation in predicate logic and to laboriously prove after each algorithmic step that despite changes to stack variables and the heap, the separation is preserved. The $(\star)$ connective of separation logic, by contrast, allows explicit description of structure separation and the frame rule effortlessly exploits that separation.

When we are dealing with a single list or tree, aliasing doesn't matter, because it simply can't occur. Even in cases where there is substructure sharing – when two trees point to the same list, for example – changes to the substructure don't affect the structures that point to it. When we are dealing with a DAG or a graph via a global invariant, aliasing is straightforward to deal with once separation has been described. Following pointers and changing pointers, which is the name of the game in very many algorithms, is perfect separation logic territory. That's why we say that reasoning in separation logic embraces the dangling pointer: it doesn't matter what is attached to the end of a dangling pointer, because some other part of the program will separately preserve an invariant of that separate data structure.

A program that manipulates a cyclic graph, or even a directed acyclic graph, has to deal explicitly with the possibility that there can be multiple pointers to a substructure. In a cyclic graph it's obviously necessary to avoid infinite recursions or endless repetitions through the same sequence of vertices. Even in an acyclic graph it's normally a good idea to avoid considering the same subgraph many times. In effect, pointers are being used as surrogates for the structures they index, and the existence of other identical pointers – aliases – matters a great deal. Programs which deal with structures that can have internal sharing must keep some kind of record of substructures encountered, either by marking them as they are encountered, or by recording them in a list or a spanning tree (or whatever).

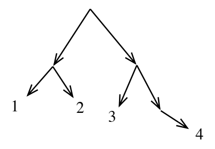
Our canonical problem is a recursive algorithm which copies a rooted, directed, ordered, possibly cyclic graph. In such a graph there can be several pointers to the same subgraph. In the copy there must be just the same number of pointers, in similar positions, to a similar substructure: that is, the aliasing relationships of the original have to be reproduced in the copy. Our specifications therefore have to describe aliasing. Because we want to specify the algorithm in terms of its effect on a subgraph which may include aliased pointers, we need to specify some of the properties of the graph outside the footprint, but we want to keep that part of the specification to a minimum, to make the reasoning as local as possible.

Previous successful uses of separation logic in this area, notably Yang's treatment [13] of the Schorr-Waite graph-marking algorithm [10] and Torp-Smith's proof [11] of the Cheney two-space garbage-collection algorithm [3], have used global invariants to describe the total heap being manipulated by a program. A particular micro-action of the program, such as marking a heap cell or assigning a pointer, uses the frame rule to pick out and modify those parts of the invariant that are affected. Aliasing structures are described by describing sets of locations. In these problems a global invariant is appropriate: there is no recursion, the whole heap is available at every point of the program, arithmetic is sometimes necessary to distinguish one sort of pointer from another, and the program always needs to able to access marks or pointers assigned at an earlier stage.

When dealing with recursive algorithms that deal with DAGs and graphs, it is clear that it isn't satisfactory to deal with global invariants. The algorithms are compact, they obviously operate on substructures, and it surely should be possible to reason locally in the same sort of way that we deal with list and tree algorithms. It's likely that the specifications will be significantly more complicated than with lists and trees, just because we have to describe aliasing. It's in that spirit that the work described below was undertaken. It's important to stress that if our reasoning is more 'local' than [13] and [11], it's because our problem algorithms are chosen to make it possible.

4 Specification style

In specifying programs, following Reynolds [8], we describe data structures by using a predicate which holds when a heap contains a representation of a data structure described by a formula. For example the tree



is described by the formula

$$\begin{aligned}&\mathsf{Node}\ (\mathsf{Node}\ (\mathsf{Tip}\ 1)\ (\mathsf{Tip}\ 2))\\ &\qquad(\mathsf{Node}\ (\mathsf{Tip}\ 3)\ (\mathsf{Node}\ \mathsf{Empty}\ (\mathsf{Tip}\ 4)))\end{aligned} \tag{1}$$

A heap can represent a Node by a triple of 1 with two tree pointers, a Tip by a pair of 0 with the tip value, and Empty by nil. These decisions can be set out in a heap predicate

$$\begin{aligned}\mathrm{tree\ nil}\ \mathsf{Empty} &\mathrel{\hat{=}} \mathbf{emp}\\ \mathrm{tree}\ t\ (\mathsf{Tip}\ \alpha) &\mathrel{\hat{=}} t \mapsto 0, \alpha\\ \mathrm{tree}\ t\ (\mathsf{Node}\ \lambda\ \rho) &\mathrel{\hat{=}} \exists l, r \cdot (t \mapsto 1, l, r \star \mathrm{tree}\ l\ \lambda \star \mathrm{tree}\ r\ \rho)\end{aligned} \tag{2}$$

This predicate describes only the shape of a heap which represents a tree and not in general its domain (i.e. its position). There are usually many heaps which make a predicate formula true. For example, the heap

$$\begin{gathered}6 \mapsto 1, \mathrm{nil}, 80 \star 10 \mapsto 1, 48, 19 \star 19 \mapsto 1, 105, 6 \star\\ 48 \mapsto 0, 1 \star 80 \mapsto 0, 4 \star 105 \mapsto 0, 4\end{gathered}$$

satisfies tree 10 T where T is the tree formula of (1). The same formula is also satisfied by

$$\begin{gathered}4 \mapsto 0, 1 \star 10 \mapsto 1, 4, 19 \star 19 \mapsto 1, 105, 66 \star\\ 66 \mapsto 1, \mathrm{nil}, 80 \star 80 \mapsto 0, 4 \star 105 \mapsto 0, 4\end{gathered}$$

It's then possible to specify simple recursive algorithms like *copytree*, which is given a pointer to a tree in the heap and produces a similarly shaped copy:

$$\{\mathrm{tree}\ t\ \tau\}\ t' := \mathit{copytree}\ t\ \{\mathrm{tree}\ t\ \tau \star \mathrm{tree}\ t'\ \tau\}$$

Because the pre- and post-conditions specify the shape but not the domain of the heap, this specification doesn't require that the input heap is part of the output. A perverse *copytree* would be permitted to throw away part or all of its input and even assign to t, provided only that it leaves two separate τ-shaped heaps on exit. For the rest of this paper we ignore that kind of problem, which can be circumvented in ways that would add no illumination to our discussion.

5 Inclusion sharing

Our first example of internal structure sharing is the inclusion of one structure within another: a list within a tree. The idiom of the solution is widely useful, and we exploit it in our treatment of cyclic graphs with forwarding pointers below.

Consider a binary tree with values at its tips and no null subtrees. The grammar of formulae describing such trees is

$$T ::= \mathsf{Node}\ T\ T \mid \mathsf{Tip}\ \mathrm{val}$$

The fringe of such a tree is a sequence of the values at its tips, which can be computed from a tree formula:

$$\begin{aligned} \mathit{fringe}\ (\mathsf{Tip}\ \ v) &\mathrel{\hat{=}} \langle v\rangle \\ \mathit{fringe}\ (\mathsf{Node}\ \lambda\ \rho) &\mathrel{\hat{=}} \mathit{fringe}\ \lambda +\!\!+\ \mathit{fringe}\ \rho \end{aligned}$$

Richard Bird posed us the problem of deriving 'without fuss' a program which links together the fringe of this kind of tree. He observed that if nodes and tips are both represented in the heap by a triple (l,v,r), tips would naturally have two nil subtree pointers l and r, either of which would be enough to distinguish them from nodes, and the other pointer slot could be used to link together the tips. Nodes would have an unused value slot. That corresponds to the heap predicate

$$\begin{aligned} \mathrm{tree}\ t\ (\mathsf{Tip}\ \ v) &\mathrel{\hat{=}} t \mapsto \mathrm{nil}, v, \_ \\ \mathrm{tree}\ t\ (\mathsf{Node}\ \lambda\ \rho) &\mathrel{\hat{=}} \exists l, r \cdot (t \mapsto l, \_, r \star \mathrm{tree}\ l\ \lambda \star \mathrm{tree}\ r\ \rho) \end{aligned} \qquad (3)$$

A list is conventionally represented by a linked sequence of (value, pointer) records, the last of which contains a nil pointer. In the heap representation of a Tip, the second two slots can be viewed as a component of a list. Rather than defining a nil-terminated list, we need to define a list segment – a sequence of records the last of which contains a continuation pointer y, the whole representing a sequence *vs*.

$$\begin{aligned} \mathrm{lseg}\ y\ y\ \langle\,\rangle &\mathrel{\hat{=}} \mathbf{emp} \\ \mathrm{lseg}\ x\ y\ (\langle v\rangle +\!\!+\ vs) &\mathrel{\hat{=}} \exists x' \cdot (x \mapsto v, x' \star \mathrm{lseg}\ x'\ y\ vs) \end{aligned}$$

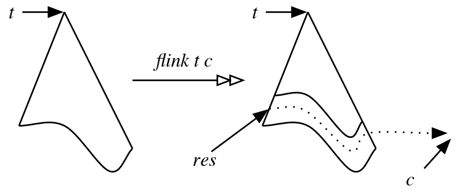
Note that a segment is a sequence of separated records, but the continuation pointer y is unconstrained. In our examples y will always point to another segment or be nil, but we don't have to rely on that when checking whether a heap satisfies the predicate.

5.1 Algorithm and specification

Since our business, at this stage of our understanding of separation logic, is post-hoc verification rather than derivation, we ignored the problems of derivation and focussed on what seemed to us to be the obvious algorithm:

```
$\mathit{fringelink}\ t\ c \mathrel{\hat{=}}$ if $[t] = \mathrm{nil}$ then
                      $[t+2] := c;\ t+1$
                   else
                      $\mathit{fringelink}\ [t]\ (\mathit{fringelink}\ [t+2]\ c)$
                   fi
```

The continuation pointer argument c allows it to link the fringes of adjacent subtrees together, and overall *fringelink t* nil returns a pointer to the fringe list of tree t. The input footprint of *fringelink t c* is clearly the tree pointed to by t, and the result points to a list taking up part of the tree space:



A heap which was both fringe F and tree T would be $F \wedge T$. But the fringe is never larger than the tree, so the output can be split into fringe and non-fringe heaps: that is, the output is $(F \star \mathrm{True}) \wedge T$. The specification of the algorithm is

```
{tree $t\ \tau$}
   $res := \mathit{fringelink}\ t\ c$
{(lseg $res\ c\ (\mathit{fringe}\ \tau) \star \mathrm{True}) \wedge \mathrm{tree}\ t\ \tau$}
```

5.2 Proof

To clarify the proof it's helpful to compile the nested function call of the else arm into a sequence of assignments. The proof of the modified algorithm must then show

```
{tree $t\ \tau$}
if $[t] = \mathrm{nil}$ then
   {tree $t$ ($\mathsf{Tip}\ v$)}
   $[t+2] := c;\ t+1$
   {(lseg $(t+1)\ c\ \langle v\rangle \star \mathrm{True}) \wedge \mathrm{tree}\ t\ (\mathsf{Tip}\ v)$}
else
   {tree $t$ ($\mathsf{Node}\ \lambda\ \rho$)}
   $res1 := \mathit{fringelink}\ [t+2]\ c$;
   $res2 := \mathit{fringelink}\ [t]\ res1$;
   $res2$
   $\left\{\begin{array}{l}(\mathrm{lseg}\ res2\ c\ (\mathit{fringe}\ \lambda +\!\!+\ \mathit{fringe}\ \rho) \star \mathrm{True}) \wedge \\ \mathrm{tree}\ t\ (\mathsf{Node}\ \lambda\ \rho)\end{array}\right\}$
fi
{(lseg $res\ c\ (\mathit{fringe}\ \tau) \star \mathrm{True}) \wedge \mathrm{tree}\ t\ \tau$}
```

The frame rule is used several times in the proof itself, and we've introduced a notational device to clarify those steps. The deduction

$$\frac{\{Q\}C\{R\}}{\{P \star Q\}C\{P \star R\}}$$

is shown linearly as

$$\begin{array}{l} \{P \star Q\} \\ \text{Framed:} \left\langle \{Q\}\ C\ \{R\} \right\rangle \\ \{P \star R\} \end{array}$$

The Tip case constructs a singleton list. The only remarkable steps

```
else
  $\{\text{tree } t\ \tau \wedge [t] \neq \text{nil}\} \therefore \{\text{tree } t\ (\mathsf{Node}\ \lambda\ \rho)\} \therefore \{t \mapsto l, \_, r \star \text{tree } l\ \lambda \star \text{tree } r\ \rho\}$
  Framed: $\langle \{\text{tree } r\ \rho\}\ res1 := fringelink\ [t+2]\ c;\ \{(\text{lseg } res1\ c\ (fringe\ \rho) \star \text{True}) \wedge \text{tree } r\ \rho\} \rangle$
  $\{t \mapsto l, \_, r \star \text{tree } l\ \lambda \star ((\text{lseg } res1\ c\ (fringe\ \rho) \star \text{True}) \wedge \text{tree } r\ \rho)\}$
  Framed: $\langle \{\text{tree } l\ \lambda\}\ res2 := fringelink\ [t]\ res1;\ \{(\text{lseg } res2\ res1\ (fringe\ \lambda) \star \text{True}) \wedge \text{tree } l\ \lambda\} \rangle$
  $\{t \mapsto l, \_, r \star ((\text{lseg } res2\ res1\ (fringe\ \lambda) \star \text{True}) \wedge \text{tree } l\ \lambda) \star ((\text{lseg } res1\ c\ (fringe\ \rho) \star \text{True}) \wedge \text{tree } r\ \rho)\} \therefore$
  $\{(\text{lseg } res2\ res1\ (fringe\ \lambda) \star \text{lseg } res1\ c\ (fringe\ \rho) \star \text{True}) \wedge (t \mapsto l, \_, r \star \text{tree } l\ \lambda \star \text{tree } r\ \rho)\} \therefore$
  $\{(\text{lseg } res2\ c\ (fringe\ \lambda +\!\!+ fringe\ \rho) \star \text{True}) \wedge \text{tree } t\ (\mathsf{Node}\ \lambda\ \rho)\}$
  $res2$
  $\{(\text{lseg } res\ c\ (fringe\ \lambda +\!\!+ fringe\ \rho) \star \text{True}) \wedge \text{tree } t\ (\mathsf{Node}\ \lambda\ \rho)\}$
```

Figure 1. ***fringelink* proof, Node case**

are the introduction of $\wedge$ and the replacement of $t \mapsto \text{nil}$ by True:

```
if $[t] = \text{nil}$ then
  $\{\text{tree } t\ \tau \wedge [t] = \text{nil}\} \therefore \{\text{tree } t\ (\mathsf{Tip}\ v)\} \therefore$
  $\{t \mapsto \text{nil}, v, \_\}$
  Framed: $\langle \{t+2 \mapsto \_\}[t+2] := c; \{t+2 \mapsto c\} \rangle$
  $\{t \mapsto \text{nil}, v, c\} \therefore$
  $\{t \mapsto \text{nil}, v, c \wedge t \mapsto \text{nil}, v, c\} \therefore$
  $\{(t+1 \mapsto v, c \star t \mapsto \text{nil}) \wedge t \mapsto \text{nil}, v, c\} \therefore$
  $\{(\text{lseg } (t+1)\ c\ \langle v \rangle \star \text{True}) \wedge \text{tree } t\ (\mathsf{Tip}\ v)\}$
  $t+1$
  $\{(\text{lseg } res\ c\ \langle v \rangle \star \text{True}) \wedge \text{tree } t\ (\mathsf{Tip}\ v)\}$
```

The Node case (figure 1) is not quite so straightforward. After the second framed step, the postcondition is of the form $A \star ((B \star \text{True}) \wedge C) \star ((D \star \text{True}) \wedge E)$. This can be rearranged, using the rules of BI plus the fact that $\text{True} \star \text{True} = \text{True}$, into $(B \star D \star \text{True}) \wedge (A \star C \star E)$. It's possible to show by induction on α that $\text{lseg}\ x\ x'\ \alpha \star \text{lseg}\ x'\ y\ \beta \vdash \text{lseg}\ x\ y\ (\alpha +\!\!+ \beta)$, which justifies the final conclusion.

6 DAGs and graphs

Cyclic graphs are a major challenge to separation logic. The heap representation of a graph involves, in principle, a tangle of pointers between interpenetrating structures. Records aren't labelled to say that they are shared. Cycles of edges aren't marked.

Despite that, graph algorithms are remarkably simple. A few basic tricks – e.g. marking, forwarding pointers – appear to cover most eventualities. It surely ought to be possible to deal with the problem as simply as the algorithms do.

DAGs – directed acyclic graphs – are halfway between trees and graphs: substructures are shared but there are no cycles. The essential difference between a DAG and a graph is that the root of a DAG is separate from its children. In search of simple solutions, we begin by considering the problem of copying a DAG, producing a copy which has exactly the same aliasing relationships as the original. As with the *copytree* example above, we ignore the necessity to state that the input DAG is undisturbed.

Although it would be possible to justify an algorithm by rational reconstruction from a specification, in truth we started with a program. One way – the slow way – to copy a DAG is to scan the structure, building up an association list of (original pointer, copy pointer) pairs as you go. Each time you come across a substructure, you check to see if it's recorded in the list: if it is, you use the already-recorded copy pointer; if it's not, you copy it and record the address of the copy in the list. Abstracting from an association list to a forwarding function, we have the following procedure, which takes as argument a pointer d to a binary DAG,[2] represented in the heap very like the binary trees of (2), and a forwarding function f representing copies made so far. It returns a pointer to a copy of the DAG, plus an updated forwarding function.

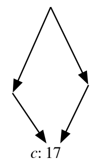
```
$copydag\ d\ f\ \hat{=}$ if $d = \text{nil}$ then nil, $f$
                 elsf $d \in \text{dom } f$ then $f\ d, f$
                 elsf $d.tag = 0$ then
                   $d' := new(0, d.val);\ d', f \oplus (d : d')$
                 else
                   $l, f' := copydag\ d.left\ f;$                          (4)
                   $r, f'' := copydag\ d.right\ f';$
                   $d' := new(1, l, r);$
                   $d', f'' \oplus (d : d')$
                 fi
```

We describe this below as an aliasing-discovery copying algorithm. Line 2 deals with aliasing; the rest of the algorithm is just tree copying except that in the else case the recursive calls augment the original forwarding function f as they go. In a DAG substructures can't share with the root, and to emphasise the point we copy the root after dealing with its children.

The ideal specification of this algorithm would be that given a pointer to a closed DAG – one with no dangling pointers – and a null forwarding function, it produces a heap which contains the original DAG and a separate identically-aliased copy, with a forwarding function which describes an isomorphism between original and copy. We approach that ideal in stages.

6.1 Describing a DAG

To specify and verify (4) we must first be able to give a syntactic description of a DAG. Our binary DAGs, like binary trees, can contain nodes and tips. We allow null subDAGs. Binding formulae 'let $x = D$ in D'', with $\mathsf{Ptr}\ x$ references to defined names, are an obvious way to describe sharing of structures. For example, the DAG



is described by the formula

$$\text{let } c = \mathsf{Tip}\ 17 \text{ in } \mathsf{Node}\ (\mathsf{Node}\ \mathsf{Empty}\ (\mathsf{Ptr}\ c))\ (\mathsf{Node}\ (\mathsf{Ptr}\ c)\ \mathsf{Empty}) \qquad (5)$$

[2]It wouldn't be difficult to generalise to n-ary DAGs, but it wouldn't add anything to the discussion.

$$\begin{aligned}
\text{dag nil } \mathsf{Empty}\; U\; U &\mathrel{\hat{=}} \textbf{emp}\\
\text{dag } d\ (\mathsf{Ptr}\ x)\ U\ U &\mathrel{\hat{=}} U\ x = d \wedge \textbf{emp}\\
\text{dag } d\ (x : \mathsf{Tip}\ \alpha)\ U\ V &\mathrel{\hat{=}} x \notin \mathrm{dom} U \wedge V = U \oplus (x : d) \wedge d \mapsto 0, \alpha\\
\text{dag } d\ (x : \mathsf{Node}\ \lambda\ \rho)\ U\ V &\mathrel{\hat{=}} \exists l, r, U', V' \cdot (x \notin \mathrm{dom} U \wedge x \notin \mathrm{dom} V' \wedge V = V' \oplus (x : d) \wedge d \mapsto 1, l, r \star \text{dag } l\ \lambda\ U\ U' \star \text{dag } r\ \rho\ U'\ V')
\end{aligned}$$

Figure 2. Heap predicate for partial DAGs

The grammar of this kind of DAG formula is

$$\begin{aligned} D ::= &\ \mathsf{Empty} \mid \mathsf{Tip}\ \text{int} \mid \mathsf{Node}\ D\ D \mid \mathsf{Ptr}\ \text{var}\\ \mid &\ \text{let var} = D \text{ in } D \end{aligned} \qquad (6)$$

The corresponding heap predicate takes a pointer d and a formula δ as arguments, just as (2) and (3) do. The third argument V is a mapping from bound names to dangling pointers which index shared structures in the heap – in effect, continuation pointers for each of the undefined names in δ. Unreferenced definitions would make disconnected structures in the heap, so we impose a relevance condition on binding formulae.

$$\begin{aligned}
\text{dag nil } \mathsf{Empty}\ V &\mathrel{\hat{=}} \textbf{emp}\\
\text{dag } d\ (\mathsf{Tip}\ \alpha)\ V &\mathrel{\hat{=}} d \mapsto 0, \alpha\\
\text{dag } d\ (\mathsf{Node}\ \lambda\ \rho)\ V &\mathrel{\hat{=}} \exists l, r \cdot \begin{pmatrix} d \mapsto 1, l, r \star \text{dag } l\ \lambda\ V \star \\ \text{dag } r\ \rho\ V \end{pmatrix}\\
\text{dag } d\ (\mathsf{Ptr}\ x)\ V &\mathrel{\hat{=}} V\ x = d \wedge \textbf{emp}\\
\text{dag } d\ (\text{let } x = \delta' \text{ in } \delta)\ V &\mathrel{\hat{=}} \exists d' \cdot \begin{pmatrix} x \text{ free in } \delta \wedge \text{dag } d'\ \delta'\ V \star \\ \text{dag } d\ \delta\ (V \oplus (x : d')) \end{pmatrix}
\end{aligned} \qquad (7)$$

Unfortunately, this predicate doesn't suit our purposes. In the heap there is no advance signalling of internal sharing – nothing to correspond to a let-in construction. The algorithm of (4) discovers internal sharing by noting substructures as it comes across them and recognising when the same substructure is referenced again. It would be possible to construct a two-stage copy process, with a first pass which discovered the aliasing structure and a second, based on the predicate of (7), which did the copying. But then the first pass would have to be based on a heap predicate which doesn't assume knowledge of aliasing, which is just the problem that we started with. It's obvious that it's no harder to verify an aliasing-discovery copying algorithm like (4), which discovers aliasing as it goes, than to take the two-stage approach.

7 Partial graphs

Any adequate DAG copying algorithm must in effect discover a spanning tree of its argument. The algorithm of (4) finds a spanning tree biased to the left – that is, it treats the first occurrence of a subDAG in a left-to-right scan as the defining instance, and cuts links which it comes across later. It's possible to transform a let-in formula like (5) into one which suits this biased search. Take the innermost let-in formula: push it inwards, preserving the relevance condition (D in let $x = D'$ in D must contain $\mathsf{Ptr}\ x$), and pushing into the left subDAG of a Node if there is a choice. The let binding will necessarily come to rest against the leftmost occurrence of a reference to its name. Replace let $x = D$ in $\mathsf{Ptr}\ x$ with $x : D$, repeat with other let bindings in turn, and the result is a formula like this:

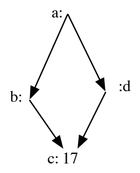
$$\mathsf{Node}\ (\mathsf{Node}\ \mathsf{Empty}\ (c : \mathsf{Tip}\ 17))\ (\mathsf{Node}\ (\mathsf{Ptr}\ c)\ \mathsf{Empty})$$

We can read this formula left-to-right: $x : D$ labels a structure on first occurrence, and $\mathsf{Ptr}\ x$ later refers to it.

Then observe that when a program scans the heap, it can't tell whether or not a substructure it has come across for the first time will be referenced later, so label *every* element, giving

$$\begin{aligned} a : \mathsf{Node}\ (b : \mathsf{Node}\ \mathsf{Empty}\ (c : \mathsf{Tip}\ 17))\\ (d : \mathsf{Node}\ (\mathsf{Ptr}\ c)\ \mathsf{Empty}) \end{aligned}$$

This formula describes the fully-labelled DAG



The grammar of these formulae is

$$D ::= \mathsf{Empty} \mid \mathsf{Ptr}\ \text{var} \mid \text{var} : \mathsf{Tip}\ \text{int} \mid \text{var} : \mathsf{Node}\ D\ D \qquad (8)$$

These formulae describe *partial graphs*: graphs consisting of named vertices in which not every name maps to a successor set, because there is no requirement that $x : D$ must occur to define $\mathsf{Ptr}\ x$. They are extremely similar to the structures described by Cardelli, Gardner and Ghelli in [2] though the motivation is different: their trees are given and the pointers defined as secondary; ours must be discovered, and which pointers are part of the tree and which are aliases depends on the direction of a scan.

We stumbled across partial graphs as a means of describing the footprint of (4), plucking them out of the air as a transformation of the let-in formulae of (6), but they do appear to be semantically interesting. They suit graphs even better than they do DAGs, as we shall see. We haven't yet had time to investigate their properties. They may be awkward to work with – it's not quite so easy to say that a partial graph is closed, for example, as it is with a let-in formula, and it's necessary to impose a unique-labelling restriction – but they do suit our purposes rather well.

8 Describing partial DAGs

The heap predicate for DAGs based on the partial graph formulae of (8) is shown in figure 2. The algorithm (4) accumulates a forwarding function; the predicate describes a left-to-right evaluation that similarly accumulates an environment. By contrast, the enviroment of (7) is built up recursively. U is the environment mapping names which are not defined in the partial DAG (once again, providing a collection of continuation pointers); V is the output environment which extends U to include also the names that are defined in the partial DAG. In the Node case, since we are dealing with a DAG, there mustn't be occurrences of $\mathsf{Ptr}\ x$ within the node. It would be sufficient to say either that x isn't mapped by environment U or that it isn't mapped by V', but we explicitly exclude it from both as a proof convenience. The unique labelling condition is imposed by the same restriction.

This predicate is specifically designed to support a left to right scan, as are the formulae on which it is based. It seems difficult to avoid this complication.

```
$\{\mathrm{dag}\ d\ \delta\ U\ V \wedge \mathrm{ran}\,U = \mathrm{dom}\ f \wedge d \neq \mathrm{nil} \wedge d \notin \mathrm{dom}\ f\}$
elsf $d.tag = 0$ then
  $\{\mathrm{dag}\ d\ (x : \mathsf{Tip}\ \alpha)\ U\ (U \oplus (x : d)) \wedge \mathrm{ran}\,U = \mathrm{dom}\ f\} \therefore$
  $\{d \mapsto 0, \alpha \wedge \mathrm{ran}\,U = \mathrm{dom}\ f \wedge x \notin \mathrm{dom}\,U\}$
  $d' := new(0, d.val);$
  $\{d \mapsto 0, \alpha \star d' \mapsto 0, \alpha \wedge \mathrm{ran}\,U = \mathrm{dom}\ f \wedge x \notin \mathrm{dom}\,U \wedge x \notin \mathrm{dom}(f \bullet U)\} \therefore$
  $\{\mathrm{dag}\ d\ (x : \mathsf{Tip}\ \alpha)\ U\ (U \oplus (x : d)) \star \mathrm{dag}\ d'\ (x : \mathsf{Tip}\ \alpha)\ (f \bullet U)\ ((f \bullet U) \oplus (x : d')) \wedge \mathrm{ran}\,U = \mathrm{dom}\ f\}$
  $d', f \oplus (d : d')$
  $\{\mathrm{dag}\ d\ (x : \mathsf{Tip}\ \alpha)\ U\ (U \oplus (x : d)) \star \mathrm{dag}\ d\ (x : \mathsf{Tip}\ \alpha)\ (f \bullet U)\ ((f \oplus (d : d')) \bullet (U \oplus (x : d))) \wedge \mathrm{ran}(U \oplus (x : d)) = \mathrm{dom}(f \oplus (d : d'))\}$
```

Figure 3. Slow *copydag*, Tip case

```
else
  $\{\mathrm{dag}\ d\ (x : \mathsf{Node}\ \lambda\ \rho)\ U\ V \wedge \mathrm{ran}\,U = \mathrm{dom}\ f\} \therefore$
  $\{d \mapsto 1, L, R \star \mathrm{dag}\ L\ \lambda\ U\ U' \star \mathrm{dag}\ R\ \rho\ U', V' \wedge \mathrm{ran}\,U = \mathrm{dom}\ f \wedge x \notin \mathrm{dom}\,U \wedge x \notin \mathrm{dom}\,V'\}$
  Framed: $\langle \{\mathrm{dag}\ L\ \lambda\ U\ U' \wedge \mathrm{ran}\,U = \mathrm{dom}\ f\}\ l, f' := copydag\ d.left\ f\ \{\mathrm{dag}\ L\ \lambda\ U\ U' \star \mathrm{dag}\ l\ \lambda\ (f \bullet U)\ (f' \bullet U') \wedge \mathrm{ran}\,U' = \mathrm{dom}\ f'\} \rangle;$
  $\{d \mapsto 1, L, R \star \mathrm{dag}\ L\ \lambda\ U\ U' \star \mathrm{dag}\ R\ \rho\ U', V' \star \mathrm{dag}\ l\ \lambda\ (f \bullet U)\ (f' \bullet U') \wedge \mathrm{ran}\,U' = \mathrm{dom}\ f' \wedge$
   $\mathrm{ran}\,U = \mathrm{dom}\ f \wedge x \notin \mathrm{dom}\,U \wedge x \notin \mathrm{dom}\,V'\}$
  Framed: $\left\langle \{\mathrm{dag}\ R\ \rho\ U', V' \wedge \mathrm{ran}\,U' = \mathrm{dom}\ f'\}\ r, f'' := copydag\ d.right\ f'\ \left\{\begin{array}{l}\mathrm{dag}\ R\ \rho\ U'\ V' \star \mathrm{dag}\ r\ \rho\ (f' \bullet U')\ (f'' \bullet V') \wedge \\ \mathrm{ran}\,V' = \mathrm{dom}\ f''\end{array}\right\} \right\rangle;$
  $\{d \mapsto 1, L, R \star \mathrm{dag}\ L\ \lambda\ U\ U' \star \mathrm{dag}\ R\ \rho\ U', V' \star \mathrm{dag}\ l\ \lambda\ (f \bullet U)\ (f' \bullet U') \star \mathrm{dag}\ r\ \rho\ (f' \bullet U')\ (f'' \bullet V') \wedge$
   $\mathrm{ran}\,V' = \mathrm{dom}\ f'' \wedge \mathrm{ran}\,U' = \mathrm{dom}\ f' \wedge \mathrm{ran}\,U = \mathrm{dom}\ f \wedge x \notin \mathrm{dom}\,U \wedge x \notin \mathrm{dom}\,V'\}$
  $d' := new(1, l, r);$
  $\{d \mapsto 1, L, R \star \mathrm{dag}\ L\ \lambda\ U\ U' \star \mathrm{dag}\ R\ \rho\ U', V' \star d' \mapsto 1, l, r \star \mathrm{dag}\ l\ \lambda\ (f \bullet U)\ (f' \bullet U') \star \mathrm{dag}\ r\ \rho\ (f' \bullet U')\ (f'' \bullet V') \wedge$
   $\mathrm{ran}\,V' = \mathrm{dom}\ f'' \wedge \mathrm{ran}\,U' = \mathrm{dom}\ f' \wedge \mathrm{ran}\,U = \mathrm{dom}\ f \wedge x \notin \mathrm{dom}\,U \wedge x \notin \mathrm{dom}\,V'\}$
  $d', f'' \oplus (d, d')$
  $\{\mathrm{dag}\ d\ (x : \mathsf{Node}\ \lambda\ \rho)\ U\ V \star \mathrm{dag}\ d'\ (x : \mathsf{Node}\ \lambda\ \rho)\ (f \bullet U)\ ((f'' \oplus (d, d')) \bullet V) \wedge \mathrm{ran}(V' \oplus (x : d) = \mathrm{dom}(f'' \oplus (d : d')\}$
```

Figure 4. Slow *copydag*, Node case

8.1 Copying a partial DAG slowly

If we disingenuously imagine that the representation of a forwarding function takes no heap space,[3] the specification of (4) is

```
$\{\mathrm{dag}\ d\ \delta\ U\ V \wedge \mathrm{ran}\,U = \mathrm{dom}\ f\}$
  $d', f' := copydag\ d\ f$
$\{\mathrm{dag}\ d\ \delta\ U\ V \star \mathrm{dag}\ d'\ \delta\ (f \bullet U)\ (f' \bullet V) \wedge \mathrm{ran}\,V = \mathrm{dom}\ f'\}$
```

where $f \bullet g$ is function composition – i.e. $(f \bullet g)\ x = f\ (g\ x)$.

We show a proof in stages. The Empty case is trivial:

```
$\{\mathrm{dag}\ d\ \delta\ U\ V \wedge \mathrm{ran}\,U = \mathrm{dom}\ f\}$
if $d = \mathrm{nil}$ then
  $\{\mathrm{dag}\ \mathrm{nil}\ \delta\ U\ V \wedge \mathrm{ran}\,U = \mathrm{dom}\ f\} \therefore$
  $\{\mathrm{dag}\ \mathrm{nil}\ \mathsf{Empty}\ U\ U \wedge \mathrm{ran}\,U = \mathrm{dom}\ f\} \therefore$
  $\{\mathbf{emp} \wedge \mathrm{ran}\,U = \mathrm{dom}\ f\} \therefore$
  $\{\mathbf{emp} \star \mathbf{emp} \wedge \mathrm{ran}\,U = \mathrm{dom}\ f\}$
  $\mathrm{nil}, f$
  $\{\mathrm{dag}\ \mathrm{nil}\ \mathsf{Empty}\ U\ U \star \mathrm{dag}\ \mathrm{nil}\ \mathsf{Empty}\ (f \bullet U)\ (f \bullet U) \wedge$
   $\mathrm{ran}\,U = \mathrm{dom}\ f\}$
```

In the Ptr case we need to argue $d \in \mathrm{dom}\ f \wedge \mathrm{dom}\ f = \mathrm{ran}\,U \rightarrow d \in \mathrm{ran}\,U \rightarrow \exists x \cdot U\ x = d$.

```
$\{\mathrm{dag}\ d\ \delta\ U\ V \wedge \mathrm{ran}\,U = \mathrm{dom}\ f \wedge d \neq \mathrm{nil}\}$
elsf $d \in \mathrm{dom}\ f$ then
  $\{\mathrm{dag}\ d\ \delta\ U\ V \wedge \mathrm{ran}\,U = \mathrm{dom}\ f \wedge d \neq \mathrm{nil} \wedge d \in \mathrm{ran}\,U\} \therefore$
  $\{\mathrm{dag}\ d\ \delta\ U\ V \wedge \mathrm{ran}\,U = \mathrm{dom}\ f \wedge d \neq \mathrm{nil} \wedge U\ x = d\} \therefore$
  $\{\mathrm{dag}\ d\ (\mathsf{Ptr}\ x)\ U\ U \wedge \mathrm{ran}\,U = \mathrm{dom}\ f \wedge U\ x = d\} \therefore$
  $\{\mathbf{emp} \wedge \mathrm{ran}\,U = \mathrm{dom}\ f \wedge U\ x = d\} \therefore$
  $\{\mathbf{emp} \wedge \mathrm{ran}\,U = \mathrm{dom}\ f \wedge U\ x = d \wedge (f \bullet U)\ x = f\ d\} \therefore$
  $\{\mathbf{emp} \star \mathbf{emp} \wedge \mathrm{ran}\,U = \mathrm{dom}\ f \wedge U\ x = d \wedge (f \bullet U)\ x = f\ d\}$
  $f\ d, f$
  $\{\mathrm{dag}\ d\ (\mathsf{Ptr}\ x)\ U\ U \star \mathrm{dag}\ (f\ d)\ (\mathsf{Ptr}\ x)\ (f \bullet U)\ (f \bullet U) \wedge$
   $\mathrm{ran}\,U = \mathrm{dom}\ f\}$
```

The Tip case (figure 3) is straightforward. The Node case (figure 4) is long, but it's unsubtle.

Writing *null* for the null forwarding function, we now have the result we need: *copydag d null* delivers a copy of the DAG pointed to by d, plus a forwarding function which we can discard (or, more realistically, dispose).

8.2 Disposing of a partial DAG slowly

Trees can be disposed recursively: subtree, subtree, ..., root. DAGs need more care, because we mustn't follow a pointer to a shared but already-disposed structure. The development so far is enough to support an algorithm for disposing DAGs. Rather than a forwarding function we keep a single list *ds* of disposed nodes. For simplicity, we suppose that a single 'dispose' gets rid of both cells at once, just

[3] This is merely a simplifying assumption. Being realistic about association lists would add nothing to the discussion: the values in the list are only compared and copied, never dereferenced.

$$
\begin{aligned}
\text{graph nil } \mathsf{Empty}\ U\ U &\mathrel{\hat=} \mathbf{emp}\\
\text{graph } g\ (\mathsf{Ptr}\ x)\ U\ U &\mathrel{\hat=} U\ x = g \wedge \mathbf{emp}\\
\text{graph } g\ (x : \mathsf{GNode}\ \alpha\ \lambda\ \rho)\ U\ V &\mathrel{\hat=} \exists l, r, U' \cdot \left(x \notin \text{dom}\, U \wedge g \mapsto \alpha, l, r \star \text{graph } l\ \lambda\ (U \oplus (x : g))\ U' \star \text{graph } r\ \rho\ U'\ V\right)
\end{aligned}
$$

Figure 5. Heap predicate for partial graphs

as C's free deals with a complete malloc record.

```
disposedag d ds  ≙ if $d = \text{nil} \vee\!\!\vee\ d \in ds$ then ds
                   elsf $d.tag = 0$ then
                      dispose d; $d :: ds$
                   else
                      $ds' := disposedag\ d.left\ ds$;
                      $ds'' := disposedag\ d.right\ ds'$;
                      dispose d;
                      $d :: ds''$
                   fi
```

The specification – this time we don't pretend to ignore the space taken up by the list, so to get rid of a DAG you need *disposelist* (*disposedag d* nil) – is

$$
\begin{aligned}
&\{\text{dag } d\ \delta\ U\ V \star \text{lseg } ds\ c\ (\text{ran}\, U)\}\\
&\quad ds' := disposedag\ d\ ds\\
&\{\text{lseg } ds'\ c\ (\text{ran}\, V)\}
\end{aligned}
$$

The proof is extremely similar to the proof of *copydag* and is omitted.

8.3 Copying and disposing a partial graph, slowly

It's as easy to copy a cyclic graph as it is to copy a DAG. In a graph we put values at nodes and there is no need for tips

$$G ::= \mathsf{Empty} \mid \mathsf{Ptr}\ \text{var} \mid \text{var} : \mathsf{GNode}\ \text{int}\ G\ G \tag{9}$$

and use a representation predicate (figure 5) which allows for the fact that we can have cycles. The graph-copying algorithm is then a minor modification of *copydag*, extending the forwarding function before copying subgraphs in order to deal with cycles.

```
copygraph g f  ≙ if $g = \text{nil}$ then nil, f
                 elsf $g \in \text{dom}\ f$ then $f\ g, f$
                 else
                    $g' := new(g.val, \text{nil}, \text{nil})$;
                    $g'.left, f' := copygraph\ g.left\ (f \oplus (g : g'))$;
                    $g'.right, f'' := copygraph\ g.right\ f'$;
                    $g', f''$
                 fi
                                                                  (10)
```

The specification is very similar to *copydag*'s.

$$
\begin{aligned}
&\{\text{graph } g\ \gamma\ U\ V \wedge \text{ran}\, U = \text{dom}\ f\}\\
&\quad g', f' := copygraph\ g\ f\\
&\{\text{graph } g\ \gamma\ U\ V \star \text{graph } g'\ \gamma\ (f \bullet U)\ (f' \bullet V) \wedge \text{ran}\, V = \text{dom}\ f'\}
\end{aligned}
$$

The proof is extremely similar too, so it's omitted.

This problem was relatively easy to solve, once we had partial graphs. It would have been very difficult to solve had we not first analysed DAGs: instead of single let bindings we would have needed recursive letr, and we would have had to deal with parallel bindings so as to describe mutually interpenetrating subgraphs.

9 Graphs with forwarding pointers

It's clear from the development so far that graphs are no harder to deal with than DAGs. In some ways they are easier: the heap predicates are simpler and have fewer cases, for example. Algorithms that deal with cyclic graphs will certainly deal with DAGs, and now that we have the partial graph notion, we need not consider DAGs any further.

9.1 Graph copying with forwarding pointers

The algorithm of (10) searches an association list as it encounters each subgraph. That takes time, and the association list takes space – at least three pointers per entry, or four if you aren't careful or you do it in ML. It's much faster, and more efficient in space if graphs are ever to be copied or disposed, to store a forwarding pointer in each node as it is copied. Once again, we started with an algorithm:

```
copyfwdgraph g ≙ if $g = \text{nil}$ then nil
                 elsf $g.fwd \neq \text{nil}$ then $g.fwd$
                 else
                    $g.fwd := new(nil, g.val, nil, nil)$;
                    $g.fwd.left := copyfwdgraph\ g.left$;           (11)
                    $g.fwd.right := copyfwdgraph\ g.right$;
                    $g.fwd$
                 fi
```

Nodes in the heap now have an extra slot for a forwarding pointers. Uncopied subgraphs must have nil forwarding pointers; copied subgraphs point to their copy. To cut cycles we must copy the root of a new subgraph before copying its children (first line of the else case); necessarily the copy must initially have dummy subgraphs, replaced by copies as they are made.

Given a 'clean' graph (one with all nil forwarding pointers) this algorithm will produce a clean copy and a separate 'dirty' original (one whose forwarding pointers define an isomorphism between the two graphs).[4] As before, we approach this ideal specification slowly, but this time armed with the partial graph notation.

The heap representation predicate of our graphs (fwdgraph) is identical to that of figure 5 except that nodes are now $g \mapsto \_, \alpha, l, r$, leaving a slot for a forwarding pointer or (as we shall see) a disposal link.

The specification of (11) needs care if we are to observe separation between the different regions of the heap. There is no longer a forwarding function argument, but the same forwarding information is present, distributed on input across those parts of the graph that have already been copied – ran$U$ in the heap predicate – and on output distributed across ran$V$. Those nodes of the input graph which have not yet been copied – (ran$(V - U)$) – must have nil forwarding pointers, and are part of the input graph in the same way that the fringe list was part of the output tree in section 5.1. The specification (figure 6) is intricate, but not unreasonably so. Our

[4]If you want the clean original back you have to traverse the input graph again with a cleanup procedure which we don't discuss here.

$$\begin{array}{l}\{(\text{fwdgraph}\, g\, \gamma\, U\, V \land ((\forall_\star x \in \text{ran}(V-U) \cdot x \mapsto \text{nil}) \star \text{True})) \star (\forall_\star y \in \text{ran}\, U \cdot y \mapsto f\, y) \land \text{dom}\, f = \text{ran}\, U \land \text{nil} \notin \text{ran}\, f\} \\ \quad g' := \mathit{copyfwdgraph}\; g \\ \left\{\begin{array}{l}(\text{fwdgraph}\, g\, \gamma\, U\, V \land ((\forall_\star x \in \text{ran}(V-U) \cdot x \mapsto f'\, x) \star \text{True})) \star \text{fwdgraph}\, g'\, \gamma\, (f \bullet U)\, (f' \bullet V) \star (\forall_\star y \in \text{ran}\, U \cdot y \mapsto f\, y \land y \mapsto f'\, y) \land \\ \text{dom}\, f' = \text{ran}\, V \land \text{nil} \notin \text{ran}\, f'\end{array}\right\}\end{array}$$

Figure 6. Specification of fast copygraph

formulation makes use of $\forall_\star$, the iterated conjunction introduced in [11].

The proof is straightforward. The Empty case (figure 7) is immediate. In the Ptr case (figure 8) we know that $g \in \text{ran}\, V$, and that $U \subseteq V$ – both from the heap predicate – and so $g.\mathit{fwd} \neq \text{nil}$, together with the precondition, tells us that $g \in \text{ran}\, U$. In presenting the GNode case (figure 9) we have compiled multi-step instructions into sequences of assignment, run together some compound assignments and omitted to indicate obvious uses of the frame rule.

The specification does support our ideal specification. A clean graph is

$$\text{fwdgraph}\, g\, \gamma\, \mathit{null}\, V \land ((\forall_\star x \in \text{ran}\, V \cdot x \mapsto \text{nil}) \star \text{True})$$

and a dirty graph is

$$\text{fwdgraph}\, g\, \gamma\, \mathit{null}\, V \land ((\forall_\star x \in \text{ran}\, V \cdot \exists y \cdot (x \mapsto y \land y \neq \text{nil})) \star \text{True})$$

The overall specification is

$$\{\text{clean}\; g\; \gamma\}\; g' := \mathit{copyfwdgraph}\; g\; \{\text{dirty}\; g\; \gamma \star \text{clean}\; g'\; \gamma\}$$

9.2 Graph disposal with forwarding pointers

Graph disposal using forwarding pointers takes more ingenuity than graph copying, because a program can't access already-disposed structures: that is, we can't dispose of the graph in a recursive scan without using an auxiliary list as in *disposegraph* above. The trick is to link together the nodes of the graph using the forwarding pointer slots.

```
$linknodes\ g\ c \mathrel{\hat{=}}$ if $g = \text{nil} \lor\!\lor g.fwd \neq \text{nil}$ then $c$
                  else
                    $g.fwd := c$;
                    $c' := linknodes\ g.left\ g$;                 (12)
                    $c'' := linknodes\ g.right\ c'$;
                    $c''$
                  fi
```

The second trick is to make a list that doesn't end in nil, because nil is used to distinguish unvisited nodes. Rejecting the temptation to invent another kind of nil – no true programmer would succumb! – we construct a sequence in which the last element points to itself. Then *disposenodes* (*linknodes g g*) does both tricks.[5]

```
$disposenodes\ g \mathrel{\hat{=}}$ if $g \neq \text{nil}$ then
                        do $oldg := g$; $g := g.fwd$; dispose $g$
                        until $g = oldg$
                      fi
```

In effect, (12) 'linearises' the graph, converting a structure in which there can be many pointers to a node into a list in which there is only one. But it doesn't really linearise it: all the subgraph pointers are still there, but we simply ignore them; and the loop at the end of the 'list' means that it isn't a list at all. Truly, separation logic proof embraces the dangling pointer!

[5]Once again, we suppose that dispose gets rid of an entire node rather than a single cell.

The specification of *linknodes* (figure 10) uses the same sort of approach as figure 6 and appeals to a heap predicate nseg which is analogous to lseg, but which represents a list of node addresses.

$$\begin{array}{r}\text{nseg}\; c\; c\; \langle\rangle \mathrel{\hat{=}} \textbf{emp} \\ \text{nseg}\; g\; c\; (\langle g\rangle +\!\!+ gs) \mathrel{\hat{=}} \exists g' \cdot (g \mapsto g', \_, \_, \_ \star \text{nseg}\; g'\; c\; gs)\end{array}$$

Proof is omitted.

10 Summary

We have attempted to extend the reach of separation logic by considering difficult examples, and to a considerable extent we have succeeded. We have found an idiomatic form, $A \land (B \star \text{True})$, for inclusion sharing. We have discovered 'partial graphs' in the grammar of (9). Our proofs have punched a hole in a significant obstacle: until now these problems hadn't been dealt with by local reasoning.

This is by no means the last word, or even our last word, on the subject. One of our aims is to find ways in which pointer-mutating programs, their data structure and their proofs come apart and compose along the same seams, so that we can begin to develop a structured programming for mutated pointer structures with sharing. In the fringe linking example we feel that we have achieved that aim. In the case of algorithms even as simple as those which copy and dispose DAGs and graphs, we can see that there is more work to do. The specifications require too much information about the environment U and its relation to V – by contrast the *fringelink* argument c can be any pointer at all.

We are particularly excited about the potential of partial graphs, both as a syntactic specification mechanism and as a semantics for graphs. We would like to know if the idea which we developed to solve a particular problem might have deeper more useful mathematical properties.

We hope that that further work will result in more than just a number of tidier proofs. It may lead to a greater understanding of graphs and how to reason about them in general. It surely ought to help us to extend the reach of separation logic to larger examples.

Acknowledgements

This work springs from endless discussion within the London separation logic community between ourselves, Josh Berdine, Ivana Mijajlovic and Noah Torp-Smith. We are particularly grateful to Torp-Smith for inspiring us, by his proof of the Cheney algorithm, to work again on graphs and for loan of the iterated conjunction operator. We are grateful for advice from John Reynolds and Hongsoek Yang. Christian Huyck pointed out to one of us how simple and fast graph copying is if you use forwarding pointers. Tony Hoare and Philippa Gardner provided useful advice on presentation of specifications and proofs.

$$\left\{(\mathrm{fwdgraph}\ g\ \gamma\ U\ V \wedge ((\forall_\star x \in \mathrm{ran}(V-U)\cdot x \mapsto \mathrm{nil}) \star \mathrm{True})) \star (\forall_\star y \in \mathrm{ran}\, U \cdot y \mapsto f\ y) \wedge \mathrm{dom}\ f = \mathrm{ran}\, U \wedge \mathrm{nil} \notin \mathrm{ran}\ f\right\}$$

if $g = \mathrm{nil}$ then

$$\left\{(\mathrm{fwdgraph\ nil}\ \mathsf{Empty}\ U\ U \wedge ((\forall_\star x \in \mathrm{ran}(U-U)\cdot x \mapsto \mathrm{nil}) \star \mathrm{True})) \star (\forall_\star y \in \mathrm{ran}\, U \cdot y \mapsto f\ y) \wedge \mathrm{dom}\ f = \mathrm{ran}\, U \wedge \mathrm{nil} \notin \mathrm{ran}\ f\right\}$$

nil

$$\left\{\begin{array}{l}(\mathrm{fwdgraph\ nil}\ \mathsf{Empty}\ U\ U \wedge ((\forall_\star x \in \mathrm{ran}(U-U)\cdot x \mapsto f\ x) \star \mathrm{True})) \star \mathrm{fwdgraph\ nil}\ \mathsf{Empty}\ (f \bullet U)\ (f \bullet U) \star \\ (\forall_\star y \in \mathrm{ran}\, U \cdot y \mapsto f\ y) \wedge \mathrm{dom}\ f = \mathrm{ran}\, U \wedge \mathrm{nil} \notin \mathrm{ran}\ f\end{array}\right\}$$

Figure 7. Fast copy graph, Empty **case**

$$\left\{(\mathrm{fwdgraph}\ g\ \gamma\ U\ V \wedge ((\forall_\star x \in \mathrm{ran}(V-U)\cdot x \mapsto \mathrm{nil}) \star \mathrm{True})) \star (\forall_\star y \in \mathrm{ran}\, U \cdot y \mapsto f\ y) \wedge \mathrm{dom}\ f = \mathrm{ran}\, U \wedge \mathrm{nil} \notin \mathrm{ran}\ f \wedge g \neq \mathrm{nil}\right\}$$

elsf $g.\mathit{fwd} \neq \mathrm{nil}$ then

$$\left\{\begin{array}{l}(\mathrm{fwdgraph}\ g\ \gamma\ U\ V \wedge ((\forall_\star x \in \mathrm{ran}(V-U)\cdot x \mapsto \mathrm{nil}) \star \mathrm{True})) \star (\forall_\star y \in \mathrm{ran}\, U \cdot y \mapsto f\ y) \wedge \\ \mathrm{dom}\ f = \mathrm{ran}\, U \wedge \mathrm{nil} \notin \mathrm{ran}\ f \wedge g \in \mathrm{ran}\, U\end{array}\right\} \therefore$$

$$\left\{\begin{array}{l}(\mathrm{fwdgraph}\ g\ (\mathsf{Ptr}\ x)\ U\ U \wedge ((\forall_\star x \in \mathrm{ran}(U-U)\cdot x \mapsto \mathrm{nil}) \star \mathrm{True})) \star (\forall_\star y \in \mathrm{ran}\, U \cdot y \mapsto f\ y) \wedge \\ \mathrm{dom}\ f = \mathrm{ran}\, U \wedge \mathrm{nil} \notin \mathrm{ran}\ f \wedge g \in \mathrm{ran}\, U\end{array}\right\}$$

$$\left\{\begin{array}{l}(\mathrm{fwdgraph}\ g\ (\mathsf{Ptr}\ x)\ U\ U \wedge ((\forall_\star x \in \mathrm{ran}(U-U)\cdot x \mapsto f\ x) \star \mathrm{True})) \star \mathrm{fwdgraph}\ (f\ g)\ (\mathsf{Ptr}\ x)\ (f \bullet U)\ (f \bullet U) \star \\ (\forall_\star y \in \mathrm{ran}\, U \cdot y \mapsto f\ y) \wedge \mathrm{dom}\ f = \mathrm{ran}\, U \wedge \mathrm{nil} \notin \mathrm{ran}\ f\end{array}\right\}$$

$g.\mathit{fwd}$

$$\left\{\begin{array}{l}(\mathrm{fwdgraph}\ g\ (\mathsf{Ptr}\ x)\ U\ U \wedge ((\forall_\star x \in \mathrm{ran}(U-U)\cdot x \mapsto f\ x) \star \mathrm{True})) \star \mathrm{fwdgraph}\ \mathit{res}\ (\mathsf{Ptr}\ x)\ (f \bullet U)\ (f \bullet U) \star \\ (\forall_\star y \in \mathrm{ran}\, U \cdot y \mapsto f\ y) \wedge \mathrm{dom}\ f = \mathrm{ran}\, U \wedge \mathrm{nil} \notin \mathrm{ran}\ f\end{array}\right\}$$

Figure 8. Fast copy graph, Ptr **case**

$$\left\{\begin{array}{l}(\mathrm{fwdgraph}\ g\ (x : \mathsf{GNode}\ \alpha\ \lambda\ \rho)\ U\ V \wedge ((\forall_\star x \in \mathrm{ran}(V-U)\cdot x \mapsto \mathrm{nil}) \star \mathrm{True})) \star (\forall_\star y \in \mathrm{ran}\, U \cdot y \mapsto f\ y) \wedge \mathrm{dom}\ f = \mathrm{ran}\, U \wedge \\ \mathrm{nil} \notin \mathrm{ran}\ f \wedge g \neq \mathrm{nil}\end{array}\right\} \therefore$$

$$\left\{\begin{array}{l}((g \mapsto \mathrm{nil}, \alpha, l, r \star \mathrm{fwdgraph}\ l\ \lambda\ (U \oplus (x:g))\ U' \star \mathrm{fwdgraph}\ r\ \rho\ U'\ V) \wedge ((\forall_\star x \in \mathrm{ran}(V-U)\cdot x \mapsto \mathrm{nil}) \star \mathrm{True})) \star \\ (\forall_\star y \in \mathrm{ran}\, U \cdot y \mapsto f\ y) \wedge \mathrm{dom}\ f = \mathrm{ran}\, U \wedge \mathrm{nil} \notin \mathrm{ran}\ f\end{array}\right\}$$

$g' := \mathit{new}(\mathit{nil}, g.\mathit{val}, \mathit{nil}, \mathit{nil});$

$$\left\{\begin{array}{l}((g \mapsto \mathrm{nil}, \alpha, l, r \star \mathrm{fwdgraph}\ l\ \lambda\ (U \oplus (x:g))\ U' \star \mathrm{fwdgraph}\ r\ \rho\ U'\ V) \wedge ((\forall_\star x \in \mathrm{ran}(V-U)\cdot x \mapsto \mathrm{nil}) \star \mathrm{True})) \star \\ g' \mapsto \mathrm{nil}, \alpha, \mathrm{nil}, \mathrm{nil} \star (\forall_\star y \in \mathrm{ran}\, U \cdot y \mapsto f\ y) \wedge \mathrm{dom}\ f = \mathrm{ran}\, U \wedge \mathrm{nil} \notin \mathrm{ran}\ f\end{array}\right\}$$

$g.\mathit{fwd} := g';$

$$\left\{\begin{array}{l}((g \mapsto g', \alpha, l, r \star \mathrm{fwdgraph}\ l\ \lambda\ (U \oplus (x:g))\ U' \star \mathrm{fwdgraph}\ r\ \rho\ U'\ V) \wedge (g \mapsto g' \star (\forall_\star x \in (\mathit{ran}(V-U) - \{g\})\cdot x \mapsto \mathrm{nil}) \star \mathrm{True})) \star \\ (\forall_\star y \in U \cdot y \mapsto (f \oplus (g:g'))\ y) \star g' \mapsto \mathrm{nil}, \alpha, \mathrm{nil}, \mathrm{nil} \wedge \mathrm{dom}\ f = \mathrm{ran}\, U \wedge \mathrm{nil} \notin \mathrm{ran}\ f \wedge \mathrm{dom}\ f' = \mathrm{ran}\, U' \wedge \mathrm{nil} \notin \mathrm{ran}\ f'\end{array}\right\}$$

$l' := \mathit{copyfwdgraph}\ g.\mathit{left};$

$$\left\{\begin{array}{l}\left(\begin{array}{l}(g \mapsto g', \alpha, l, r \star \mathrm{fwdgraph}\ l\ \lambda\ (U \oplus (x:g))\ U' \star \mathrm{fwdgraph}\ r\ \rho\ U'\ V) \wedge \\ ((\forall_\star x \in \mathrm{ran}(U'-U)\cdot x \mapsto f'\ x) \star (\forall_\star x \in \mathrm{ran}(V-U')\cdot x \mapsto \mathrm{nil}) \star \mathrm{True})\end{array}\right) \star (\forall_\star y \in \mathrm{ran}\, U \cdot y \mapsto f'\ y) \star g' \mapsto \mathrm{nil}, \alpha, \mathrm{nil}, \mathrm{nil} \star \\ (\mathrm{fwdgraph}\ l'\ \lambda\ ((f \oplus (g:g') \bullet (U \oplus (x:g)))\ (f' \bullet U') \wedge \mathrm{dom}\ f = \mathrm{ran}\, U \wedge \mathrm{nil} \notin \mathrm{ran}\ f \wedge \mathrm{dom}\ f' = \mathrm{ran}\, U' \wedge \mathrm{nil} \notin \mathrm{ran}\ f'\end{array}\right\}$$

$g.\mathit{fwd}.\mathit{left} := l';$

$$\left\{\begin{array}{l}\left(\begin{array}{l}(g \mapsto g', \alpha, l, r \star \mathrm{fwdgraph}\ l\ \lambda\ (U \oplus (x:g))\ U' \star \mathrm{fwdgraph}\ r\ \rho\ U'\ V) \wedge \\ ((\forall_\star x \in \mathrm{ran}(U'-U)\cdot x \mapsto f'\ x) \star (\forall_\star x \in \mathrm{ran}(V-U')\cdot x \mapsto \mathrm{nil}) \star \mathrm{True})\end{array}\right) \star (\forall_\star y \in \mathrm{ran}\, U \cdot y \mapsto f'\ y) \star g' \mapsto \mathrm{nil}, \alpha, l', \mathrm{nil} \star \\ (\mathrm{fwdgraph}\ l'\ \lambda\ ((f \oplus (g:g') \bullet (U \oplus (x:g)))\ (f' \bullet U') \wedge \mathrm{dom}\ f = \mathrm{ran}\, U \wedge \mathrm{nil} \notin \mathrm{ran}\ f \wedge \mathrm{dom}\ f' = \mathrm{ran}\, U' \wedge \mathrm{nil} \notin \mathrm{ran}\ f'\end{array}\right\}$$

$r' := \mathit{copyfwdgraph}\ g.\mathit{right};$

$$\left\{\begin{array}{l}\left(\begin{array}{l}(g \mapsto g', \alpha, l, r \star \mathrm{fwdgraph}\ l\ \lambda\ (U \oplus (x:g))\ U' \star \mathrm{fwdgraph}\ r\ \rho\ U'\ V) \wedge \\ ((\forall_\star x \in \mathrm{ran}(V-U)\cdot x \mapsto f''\ x) \star (\forall_\star x \in \mathrm{ran}(V-V)\cdot x \mapsto \mathrm{nil}) \star \mathrm{True})\end{array}\right) \star (\forall_\star y \in \mathrm{ran}\, U \cdot y \mapsto f''\ y) \star g' \mapsto \mathrm{nil}, \alpha, l', \mathrm{nil} \star \\ (\mathrm{fwdgraph}\ l'\ \lambda\ ((f \oplus (g:g') \bullet U \oplus (x:g))\ (f' \bullet U') \star (\mathrm{fwdgraph}\ r'\ \rho\ (f' \bullet U)\ (f'' \bullet V) \wedge \\ \mathrm{dom}\ f = \mathrm{ran}\, U \wedge \mathrm{nil} \notin \mathrm{ran}\ f \wedge \mathrm{dom}\ f' = \mathrm{ran}\, U' \wedge \mathrm{nil} \notin \mathrm{ran}\ f' \wedge \mathrm{dom}\ f'' = \mathrm{ran}\, V \wedge \mathrm{nil} \notin \mathrm{ran}\ f''\end{array}\right\}$$

$g.\mathit{fwd}.\mathit{right} := r';$

$$\left\{\begin{array}{l}\left(\begin{array}{l}(g \mapsto g', \alpha, l, r \star \mathrm{fwdgraph}\ l\ \lambda\ (U \oplus (x:g))\ U' \star \mathrm{fwdgraph}\ r\ \rho\ U'\ V) \wedge \\ ((\forall_\star x \in \mathrm{ran}(V-U)\cdot x \mapsto f''\ x) \star (\forall_\star x \in \mathrm{ran}(V-V)\cdot x \mapsto \mathrm{nil}) \star \mathrm{True})\end{array}\right) \star (\forall_\star y \in \mathrm{ran}\, U \cdot y \mapsto f''\ y) \star g' \mapsto \mathrm{nil}, \alpha, l', r' \star \\ (\mathrm{fwdgraph}\ l'\ \lambda\ ((f \oplus (g:g') \bullet U \oplus (x:g))\ (f' \bullet U') \star (\mathrm{fwdgraph}\ r'\ \rho\ (f' \bullet U)\ (f'' \bullet V) \wedge \\ \mathrm{dom}\ f = \mathrm{ran}\, U \wedge \mathrm{nil} \notin \mathrm{ran}\ f \wedge \mathrm{dom}\ f' = \mathrm{ran}\, U' \wedge \mathrm{nil} \notin \mathrm{ran}\ f' \wedge \mathrm{dom}\ f'' = \mathrm{ran}\, V \wedge \mathrm{nil} \notin \mathrm{ran}\ f''\end{array}\right\} \therefore$$

$$\left\{\begin{array}{l}(\mathrm{fwdgraph}\ g\ (\mathsf{GNode}\ \alpha\ \lambda\ \rho) \wedge ((\forall_\star x \in \mathrm{ran}(V-U)\cdot x \mapsto f''\ x) \star \mathrm{True})) \star \\ (\forall_\star y \in \mathrm{ran}\, U \cdot y \mapsto f''\ y) \star \mathrm{fwdgraph}\ (f''\ g)\ (\mathsf{GNode}\ \alpha\ \lambda\ \rho)\ (f \bullet U)\ (f'' \bullet V) \wedge \mathrm{dom}\ f'' = \mathrm{ran}\, V \wedge \mathrm{nil} \notin \mathrm{ran}\ f''\end{array}\right\}$$

$g.\mathit{fwd}$

$$\left\{\begin{array}{l}(\mathrm{fwdgraph}\ g\ (\mathsf{GNode}\ \alpha\ \lambda\ \rho) \wedge ((\forall_\star x \in \mathrm{ran}(V-U)\cdot x \mapsto f''\ x) \star \mathrm{True})) \star (\forall_\star y \in \mathrm{ran}\, U \cdot y \mapsto f''\ y) \star \\ \mathrm{fwdgraph}\ \mathit{res}\ (\mathsf{GNode}\ \alpha\ \lambda\ \rho)\ (f \bullet U)\ (f'' \bullet V) \wedge \mathrm{dom}\ f'' = \mathrm{ran}\, V \wedge \mathrm{nil} \notin \mathrm{ran}\ f''\end{array}\right\}$$

Figure 9. Fast copy graph, GNode **case**

$$\left\{(\mathrm{fwdgraph}\ g\ \gamma\ U\ V \wedge ((\forall_\star x \in \mathrm{ran}(V-U)\cdot x \mapsto \mathrm{nil}) \star \mathrm{True})) \star (\forall_\star y \in \mathrm{ran}\, U \cdot \exists z \cdot (y \mapsto z \wedge z \neq \mathrm{nil})) \wedge g \neq \mathrm{nil} \rightarrow c \neq \mathrm{nil}\right\}$$

$c' := \mathit{linknodes}\ g\ c$

$$\left\{\begin{array}{l}((c' = c \wedge \mathbf{emp}) \vee (\mathrm{nseg}\ c'\ g\ \_ \star \mathrm{nseg}\ g\ c\ \langle g \rangle)) \wedge ((\forall_\star x \in \mathrm{ran}(V-U)\cdot \exists z' \cdot (x \mapsto z' \wedge z' \neq \mathrm{nil})) \star \mathrm{True})) \star \\ (\forall_\star y \in \mathrm{ran}\, U \cdot \exists z \cdot (y \mapsto z \wedge z \neq \mathrm{nil}))\end{array}\right\}$$

Figure 10. Specification of *linknodes*

11 References

[1] R. Bornat. Proving pointer programs in Hoare logic. In R. C. Backhouse and J. N. Oliveira, editors, *Mathematics of Program Construction, 5th International Conference*, LNCS, pages 102–126. Springer, 2000.

[2] L. Cardelli, P. Gardner, and G. Ghelli. Querying trees with pointers. Unpublished notes, 2003.

[3] C. J. Cheney. A nonrecursive list compacting algorithm. *Comm. ACM*, 13(11):677–678, 1970.

[4] S. Ishtiaq and P. O'Hearn. BI as an assertion language for mutable data structures. In *28th POPL*, pages 14–26, London, January 2001.

[5] P. O'Hearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In L. Fribourg, editor, *CSL 2001*, pages 1–19. Springer-Verlag, 2001. LNCS 2142.

[6] P. W. O'Hearn and D. J. Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5(2):215–244, June 1999.

[7] D. Pym. *The Semantics and Proof Theory of the Logic of Bunched Implications*, volume 26 of *Applied Logic Series*. Kluwer Academic Publishers, 2002.

[8] J. Reynolds. Separation logic: a logic for shared mutable data structures. Invited Paper, LICS'02, 2002.

[9] J. C. Reynolds. Intuitionistic reasoning about shared mutable data structure. In J. Davies, B. Roscoe, and J. Woodcock, editors, *Millennial Perspectives in Computer Science*, pages 303–321. Palgrave, 2000.

[10] H. Schorr and W. Waite. An efficient machine independent procedure for garbage collection in various list structures. *Comm. ACM*, 10(8):501–506, 1967.

[11] N. Torp-Smith. *Proving a garbage collecter using local reasoning*. Master's thesis, IT University of Copenhagen, 2003.

[12] H. Yang. An example of local reasoning in BI pointer logic: the Schorr-Waite graph marking algorithm. Proceedings of the SPACE Workshop, 2001.

[13] H. Yang. *Local Reasoning for Stateful Programs*. Ph.D. thesis, University of Illinois, Urbana-Champaign, Illinois, USA, 2001.