Datatypes in Memory

David Aspinall[1] and Piotr Hoffman[2]

[1] LFCS, School of Informatics, University of Edinburgh, U.K.
[2] Institute of Informatics, Warsaw University, Poland

Abstract. Besides functional correctness, specifications must describe other properties of permissible implementations. We want to use simple algebraic techniques to specify *resource usage* alongside functional behaviour. In this paper we examine the space behaviour of datatypes, which depends on the representation of values in memory. In particular, it varies according to how much values are allowed to overlap, and how much they must be kept apart to ensure correctness for destructive space-reusing operations.
We introduce a mechanism for specifying datatypes represented in a memory, with operations that may be destructive to varying degrees. We start from an abstract model notion for data-in-memory and then show how to specify the observable behaviour of models. The method is demonstrated by specifications of lists-in-memory and pointers; with a suitable definition of implementation, we show that lists-in-memory may be implemented by pointers. We then present a method for proving implementations correct and show that it is sound and, under certain assumptions, complete.

1 Introduction

This paper is part of an investigation into using simple algebraic techniques to write specifications of *resource usage* alongside functional correctness, where resources are quantitative measures such as time, space, power, and the like. Resource usage is of course a relative notion, and depends on the computation mechanism of an underlying machine as well as the representation of data on that machine. We would like to write specifications which are as abstract as possible with respect to these low-level details, but which nonetheless are able to distinguish usefully between different algorithms and representations which are not distinguished by classical algebraic specifications.

We start off here by considering *memory* as the prototypical resource, and consider the behaviour of datatype operations which are implemented in memory. Many standard algorithms make use of shared mutable data structures; these algorithms have quite different resource usage behaviour compared with functional versions that copy data instead. For mutating algorithms to work correctly, the layout in memory of the data structures must satisfy certain conditions; for example, some parts of these structures may occupy the same memory cells, some may not. We provide a mechanism to specify layout constraints, which enable

or prevent mutating algorithms, and so restrict the class of models of our specifications to ones which have certain resource behaviours.

We specify layout constraints by using *preservation* and *disjointness* predicates which restrict the use of sharing in implementations. Intuitively, a memory-altering operation *preserves* some data object in memory if after executing it the object is still available in the new memory. Otherwise, the object is destroyed and the operation considered *destructive*. If two objects are *disjoint* (separate) in the memory, then destructive operations on the first object cannot affect the second. Motivating examples follow in the body of the paper.

A key insight is that we need only be concerned by the *behavioural*, or observable, consequences of a given data layout, not by the layout itself in a concrete model. Both preservation and disjointness have behavioural characterisations, and it turns out that for standard heap models these abstract characterisations coincide with the natural, naive notions. We get sensible results for other memory models as well, which allow comparison between functional and imperative implementations within the same logical framework, for example.

As well as mechanisms for specifying data structures, we define a notion of *implementation*. Using a simple form of program, we show how to implement one data structure in terms of another. We then give some basic ideas for proving implementations correct by a form of equational reasoning. We show that this method is sound, and, with restrictions, complete.

Contributions and related work. As far as we are aware, this is the first explicit attempt to study an approach to datatype space usage using algebraic specification methods. There has been a wealth of recent activity in program logics for pointer implementations datatypes in concrete memory models and type theories or analyses for shape and layout description (to mention only a few, e.g. [1–5]). Notably, Bunched Implications, BI, provides an abstract model theory for resources, as well as a substructural logic for describing models [6]. Although we also aim at an abstract approach, we intentionally work from first principles within the algebraic framework, rather than try to recast lines of work based on different semantic foundations. More comments on related work are in the conclusions.

Outline. The structure of the rest of this paper is as follows. In Sect. 2 we introduce the abstract algebraic framework and two canonical example algebras. In Sect. 3 we define the central behavioural equivalence relation which we use as a basis for both specification and reasoning. The relation can express preservation of data at the same time as behaviour of operations. We apply this to a specification of lists in Sect. 4, which we write in a specially defined behavioural version of conditional equational logic. Sect. 5 then shows how to use the behavioural approach to define a natural *disjointness relation* which is also useful in specifications. Sect. 6 gives a specification for pointers and in Sect. 7 we define a notion of implementation and show how pointers may be used to implement lists; we sketch how this may be proved formally and prove that our approach is sound and, in certain cases, complete. Sect. 8 concludes and gives some comparison with the related work.

2 Memory Signatures and Algebras

Definition 1. *A* memory signature *consists of the following components:*

- *disjoint sets of* abstract sorts *and* memory sorts*,*
- *a set of* abstract operations *of the form* $f : s_1 \times \cdots \times s_n \to t_1 \times \cdots \times t_k$ *where* $n, k \geq 0$ *and* $s_1, \ldots, s_n, t_1, \ldots, t_k$ *are abstract sorts,*
- *a set of* memory operations *of the form* $f : \mu \times (s_1 \times \cdots \times s_n) \to \mu \times (t_1 \times \cdots \times t_k)$ *where* μ *is a special symbol representing the memory and where* $n, k \geq 0$ *and* $s_1, \ldots, s_n, t_1, \ldots, t_k$ *are arbitrary sorts.*

The idea here is that objects of abstract sorts are directly observable, whereas objects of memory sorts can only be interpreted in the context of a memory via the memory operations. Abstract operations have a purely auxiliary function and are used in specifications. A memory operation is a form of "machine instruction", representing the actual steps of computation of the considered machine. The machine is modelled by a *memory algebra* over the signature.

Definition 2. *A memory algebra* A *consists of the following components:*

- *a non-empty set* $A[\mu]$ *of* memories*,*
- *for any sort* s*, a set* $A[s]$ *of* objects of type s*,*
- *for any operation* f*, a partial function* $A[f]$ *of appropriate type,*
- *for any memory sort* s*, a* validity predicate $A_V[s] \subseteq A[\mu] \times A[s]$.

If $(m, o) \in A_V[s]$ *we say* o is valid in m *and write* $o \in m$*. Validity in a memory is extended pointwise to tuples of objects of arbitrary sort, considering objects of abstract sort to be valid in any memory. A memory algebra must ensure that* memory operations preserve validity*, i.e., whenever* $\alpha \in m$ *and* $A[f](m, \alpha)$ *is defined and equal to* (m', β)*, then* $\beta \in m'$.

The partiality of memory operations is intended to represent errors or non-termination, but not out-of-memory exceptions. Although our approach is designed to deal with out-of-memory conditions, in this paper we assume that they do not occur. Out-of-memory exceptions can be included at the cost of some extra complexity, by adding another form of undefinedness so that non-termination and lack of memory can be distinguished.

Validity allows us to model the destruction of data. Any memory operation $f(m, \alpha)$ must produce a memory m' and output β valid in m', but we do not require the input α to remain valid in the new memory m'. Destructive operations, such as disposing a pointer, can destroy their own arguments.

We illustrate these definitions with concrete examples. Consider the memory signature with abstract sort `bool` and operations `t, f : bool` and with memory sort `list` and the following memory operations:

$\texttt{nil} : \mu \to \mu \times \texttt{list}$	$\texttt{isnil} : \mu \times \texttt{list} \to \mu \times \texttt{bool}$
$\texttt{cons} : \mu \times \texttt{bool} \times \texttt{list} \to \mu \times \texttt{list}$	$\texttt{hd} : \mu \times \texttt{list} \to \mu \times \texttt{bool}$
$\texttt{tl} : \mu \times \texttt{list} \to \mu \times \texttt{list}$	$\texttt{delete} : \mu \times \texttt{list} \to \mu$

Define a memory algebra A over the above signature as follows. Let the abstract, boolean components be defined as usual, and let the memories all be sequences

of pairs of natural numbers, which we treat as addresses: $A[\mu] \subseteq \mathbb{N} \to \mathbb{N}^2$. In other words, a memory is an infinite address space with two addresses, the first representing a boolean, stored at any location. If a is an address and m is a memory, then the *a-sequence in m* is the sequence $\{a_i\}_{i\in\mathbb{N}}$ defined by $a_0 = a$ and $a_{i+1} = \pi_2(m(a_i))$. Now define $A[\mu]$ to contain all $m \in \mathbb{N} \to \mathbb{N}^2$ such that the 0-sequence in m does not contain any repetitions. The 0-sequence is called the *free list*. Addresses in this sequence are called *free addresses*. Note that 0 is always free. Finally, let $A[\texttt{list}] = \mathbb{N}$ and define a list a to be valid in a memory m if in the a-sequence in m a free address occurs somewhere, and if the first such address is 0.

Now the memory operations are defined as follows on valid arguments:

- $\texttt{nil}(m) = (m, 0)$, and $\texttt{isnil}(m, a)$ is true if $a = 0$ and false otherwise;
- $\texttt{cons}(m, b, a) = (m', a')$, where m' is m with some free a' removed from the free list, with $m'(a') = (0, a)$ if b is false, and $m'(a') = (1, a)$ if b is true;
- $\texttt{hd}(m, a) = (m, \pi_1(m(a)))$ if $a \neq 0$; and $\texttt{tl}(m, a) = (m, \pi_2(m(a)))$ if $a \neq 0$;
- $\texttt{delete}(m, a) = m'$, where m' is m with all addresses from the a-sequence in m added to the free list.

Here, the free list at the 0-sequence is treated as a pool of memory for allocation and deallocation. In all cases not covered, the memory operations are undefined. We call this model of lists the *pointer model*.

Another model of lists is the algebra B with the same boolean component as A, but with $B[\mu] = \{*\}$ a singleton set and $B[\texttt{list}]$ the set of all finite sequences of booleans. Then all the list operations work just as regular list operations, except that they additionally return $*$ as the new memory. In particular, $\texttt{delete}(m, l) = m$ for all lists l. This model of lists is called the *algebraic model*.

3 Behavioural Equivalence

We now define a notion of behavioural equivalence for values in memory algebras. We do this by conceiving a memory algebra as a machine which contains a memory and a finite number of variables. The variables may keep data which is directly observable (of an abstract sort), or data that may only be interpreted using the memory (of a memory sort). The machine computes by applying a memory operation to the existing memory and data, thereby obtaining a new memory and new data. Two states of a machine should be considered equivalent if no sequence of steps of the machine leads to any difference in observable data.

Formally, a *state* of a memory algebra is a pair (m, γ), where m is a memory and $\gamma = (\gamma_1 : s_1, \ldots, \gamma_n : s_n)$ is a tuple of objects valid in m. Then n is the *length* of the state, and $(s_1, \ldots, s_n)$ is the *type* of the state. If $F : \{1, \ldots, N\} \to \{1, \ldots, n\}$ is any function, with n being the length of a state (m, γ) and N arbitrary, then the composition $(m, F(\gamma))$ defined by $F(\gamma)_i = \gamma_{F(i)}$ for $1 \leq i \leq N$ is a state as well. This is simply a rearrangement of the state (m, γ), possibly reordering, removing and duplicating objects. For any state (m, γ) and memory m', let $\gamma|_{m'}$ be the tuple obtained by removing from γ any objects not valid in m'. Of course, $(m', \gamma|_{m'})$ is a state. Finally, let "+" denote concatenation of tuples.

Definition 3. *The* behavioural equivalence $\sim$ *in a memory algebra A is the greatest relation on states of equal type such that if $(m_1, \gamma_1) \sim (m_2, \gamma_2)$ then:*

1. *if v_1, v_2 are abstract values in corresponding positions in γ_1, γ_2, then $v_1 = v_2$.*
2. *if v is abstract, then $(m_1, \gamma_1 + (v)) \sim (m_2, \gamma_2 + (v))$.*
3. *if γ_1 and γ_2 have length n and $F : \{1, \ldots, N\} \to \{1, \ldots, n\}$ is any function, then $(m_1, F(\gamma_1)) \sim (m_2, F(\gamma_2))$.*
4. *if $\gamma_1 = \eta_1 + \delta_1$ and $\gamma_2 = \eta_2 + \delta_2$, η_1 and η_2 have the same length and f is an appropriately typed memory operation, then $A[f](m_1, \eta_1)$ is defined iff $A[f](m_2, \eta_2)$ is. In this case, let $A[f](m_1, \eta_1) = (m_1', \eta_1')$ and $A[f](m_2, \eta_2) = (m_2', \eta_2')$; it is required that for all indices i, $(\gamma_1)_i$ is valid in m_1' iff $(\gamma_2)_i$ is valid in m_2' and $(m_1', \eta_1' + \gamma_1|_{m_1'}) \sim (m_2', \eta_2' + \gamma_2|_{m_2'})$.*

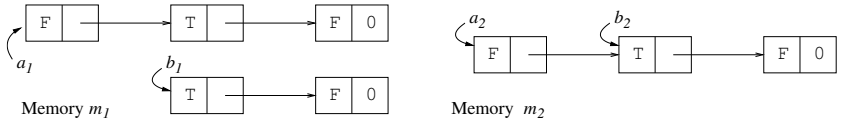
The above relation is well-defined and is an equivalence. Intuitively:

1. abstract sorts are observable.
2. one may at any moment add arbitrary variables of abstract sort.
3. one may rearrange the variables.
4. one may apply memory operations to valid objects, but any objects invalidated in doing so are removed from the state; both undefinedness and destruction of data are observable.

Because invalid variables are removed from the state in clause 4, we forbid a computation that holds a "dangling" pointer which later becomes valid again. This doesn't imply that some form of on-the-fly garbage collection is involved; it just means that programs cannot assume anything about invalidated objects, and specifications cannot express such assumptions.

In the algebraic model of lists there cannot be any interaction between two lists; we have $(m_1, \gamma_1) \sim (m_2, \gamma_2)$ iff $(m_1, (\gamma_1)_i) \sim (m_2, (\gamma_2)_i)$ for all $1 \le i \le n$, where n is the length of γ_1 and γ_2. In this case $\sim$ is simply the identity relation, because any pair of non-equal lists is differentiated by an appropriate number of `tl` operations and then a `hd` or `isnil` operation. This is true for any "non-destructive" model, even if non-equal but equivalent lists exist in it.

Things are different in the pointer model, where we have a destructive operation and there can be overlaps between lists. Consider the two memories:



Here $(m_1, (a_1, b_1)) \sim (m_2, (a_2, b_2))$ doesn't hold, although $(m_1, (a_1)) \sim (m_2, (a_2))$ and $(m_1, (b_1)) \sim (m_2, (b_2))$ both hold. This is because the lists in m_2 interfere in a way that can be observed by performing a `delete` operation.

For any state $\zeta = (m, (a_1 : \texttt{list}, \ldots, a_n : \texttt{list}))$ in the pointer model, let Φ_ζ take any pair $1 \le i, j \le n$ to the list of booleans which is kept in the maximal common part of a_i and a_j in m (in particular, $\Phi_\zeta(i, i)$ is the list of booleans corresponding to the list a_i in m). Then $\sim$ is the kernel of Φ, that is, for any tuples ζ_1 and ζ_2 of length n of lists we have $\zeta_1 \sim \zeta_2$ iff $\Phi_{\zeta_1} = \Phi_{\zeta_2}$.

4 A Specification of Boolean Lists

We can specify memory algebras in two parts: a specification of the abstract part, and a specification of the memory part. We suppose that the abstract part (i.e., the booleans) is already specified, and concentrate on the memory part.

We use axioms of a very simple form, similar to conditional equational logic. Of course, this does not mean that more complex logics cannot be used in our approach. Formulae are given by the following grammar:

$$\begin{aligned}\phi ::= {} & (m,\alpha) \sim (m,\alpha) \mid f(m,\alpha) = \bot \mid f(m,\alpha) \neq \bot \mid \\ & x \in m \mid \forall m \cdot \phi \mid \forall x \cdot \phi \mid \phi \wedge \phi \mid \\ & x \in m \implies \phi \mid f(m,\alpha) \to (m,\alpha) \implies \phi\end{aligned}$$

Here m ranges over variables binding memories, x over other variables, and α over tuples of variables. Variable typing is assumed but left implicit. The variables x may be bound either to objects of a single sort or to finite tuples of objects; such variables will usually be named γ, δ, etc. So the quantification $\forall \gamma \cdot \gamma \in m \implies \phi$ says that for all tuples γ of objects, if all these objects are valid in m, then ϕ holds; we abbreviate this by writing $\forall \gamma \in m \cdot \phi$. The formula $f(m, x_1, \ldots, x_n) \to (m', y_1, \ldots, y_k) \implies \phi$ is true if whenever $f(m, x_1, \ldots, x_n)$ is defined, then after binding the result to m' and $y_1, \ldots, y_k$, the formula ϕ holds. If a variable y_i is not used in ϕ, we may write $\_$ instead of y_i.

As syntactic sugar we use equality, $x = y$, if x and y are of an abstract sort. This can be expressed using the relation $\sim$ as $\forall m \cdot (m, x) \sim (m, y)$. For memory sorts, our axioms make more use of the behavioural equivalence: we write $m \lesssim m'$ as a shorthand for the *non-destructiveness assertion* $\forall \gamma \in m \cdot (m, \gamma) \sim (m', \gamma)$, which says that all objects of m are preserved (observationally) in m'. To specify additionally that some object a in m behaves equivalently to a' in m' we write $(m, a) \lesssim (m', a')$ as a shorthand for $\forall \gamma \in m \cdot (m, a, \gamma) \sim (m', a', \gamma))$. This generalises in the obvious way to a tuple of objects.
The specification of boolean lists begins with the following three axioms:

$$\forall m \cdot \mathtt{nil}(m) \neq \bot \ \wedge \ (\mathtt{nil}(m) \to (m', \_) \implies m \lesssim m') \tag{1}$$

$$\forall m \forall l \in m \cdot \mathtt{isnil}(m, l) \neq \bot \ \wedge \ (\mathtt{isnil}(m, l) \to (m', \_) \implies m \lesssim m') \tag{2}$$

$$\forall m \forall b \forall l \in m \cdot \mathtt{cons}(m, b, l) \neq \bot \ \wedge \ (\mathtt{cons}(m, b, l) \to (m', \_) \implies m \lesssim m') \tag{3}$$

These say that `nil`, `isnil` and `cons` are always defined on valid arguments, and that they are non-destructive. This does not mean that preexisting objects can't be changed at all: that can happen, so long as the result is behaviourally equivalent to the original. Next we specify the behaviour of `isnil`, `hd` and `tl`:

$$\forall m \cdot \mathtt{nil}(m) \to (m', l) \implies \mathtt{isnil}(m', l) \to (\_, b) \implies b = \mathtt{t} \tag{4}$$

$$\forall m \forall b \forall l \in m \cdot \mathtt{cons}(m, b, l) \to (m', l') \implies \mathtt{isnil}(m', l') \to (\_, b) \implies b = \mathtt{f} \tag{5}$$

$$\forall m \forall b \forall l \in m \cdot \mathtt{cons}(m, b, l) \to (m', l') \implies \mathtt{hd}(m', l') \neq \bot \wedge (\mathtt{hd}(m', l') \to (m'', b') \implies (m', b) \lesssim (m'', b')) \tag{6}$$

$$\forall m \forall b \forall l \in m \cdot \texttt{cons}(m,b,l) \to (m',l') \implies \texttt{tl}(m',l') \neq \bot \wedge (\texttt{tl}(m',l') \to (m'',l'') \implies (m',l) \lesssim (m'',l'')) \quad (7)$$

Note that axiom (6) says not only that the correct boolean value is returned, but also that `hd` is non-destructive. Axiom (7) is even stronger: `tl` is non-destructive and the produced tail must fully share with the original tail.

We can give alternative axioms for `hd` and `tl` that specify different amounts of destructiveness. If instead of axiom (6) we wrote:

$$\forall m \forall b \forall l \in m \cdot \texttt{cons}(m,b,l) \to (m',l') \implies \texttt{hd}(m',l') \neq \bot \wedge (\texttt{hd}(m',l') \to (m'',b') \implies (m,b) \lesssim (m'',b'))$$

then we would obtain lists in which `hd` is allowed (though not forced) to destroy or modify the head of the list. Similarly, we could allow `tl` to destroy or modify the head of the list when computing the tail. Yet more possibilities exist, e.g., one could allow `tl` to fully destroy the old list. This would need an axiom somewhat similar to the (forthcoming) axioms for `delete`, plus an axiom of the form:

$$\forall m \forall b \forall l \in m \cdot \texttt{cons}(m,b,l) \to (m',l') \implies \texttt{tl}(m',l') \to (m'',l'') \implies (m,l) \sim (m'',l'')$$

As for `delete`, the task would be easy with a *disjointness predicate* $o_1 \bot_m o_2$ stating that two given objects are disjoint in a memory m (with respect to a set of operations, see next section). Separation here means that manipulating o_1 cannot have an effect on o_2 and vice versa. Using this we could write:

$$\forall m \forall l \in m \cdot \texttt{delete}(m,l) \neq \bot$$
$$\forall m_0 \cdot \texttt{nil}(m_0) \to (m,l) \implies \forall \gamma \in m_0 \cdot \gamma \bot_m l$$

The second formula states that a nil list and any further manipulation of it (e.g. by `cons` and then `delete`), cannot have any effect on preexisting objects. The separation predicate is introduced in the next section. However, without it we can specify `delete` by the two axioms:

$$\forall m_0 \cdot \texttt{nil}(m_0) \to (m,l) \implies \texttt{delete}(m,l) \neq \bot \wedge (\texttt{delete}(m,l) \to m' \implies m_0 \lesssim m') \quad (8)$$

$$\forall m \forall b \forall l \in m \cdot \texttt{cons}(m,b,l) \to (m',l') \implies \texttt{delete}(m,l) \to m_0'' \implies \texttt{delete}(m',l') \neq \bot \wedge (\texttt{delete}(m',l') \to m'' \implies m_0'' \lesssim m'') \quad (9)$$

Axiom (8) says that deleting the nil list does not destroy preexisting objects. Axiom (9) says that if we add an element and then delete the list, then objects that wouldn't have been destroyed if we deleted the list without adding the element will be left intact. Thus, axiom (9) allows us to show that deleting a longer list is like deleting a nil list, and axiom (8) shows that deleting a nil list does not destruct unrelated objects. Together, they guarantee that preexisting

objects will be retained. Clearly these axioms do not force the model to be a destructive one; `delete` may be a dummy operation, as in the algebraic list model. But this would change if we introduced methods for counting the amount of used memory, for example; then we could specify that `delete` decreases the amount of memory used.

Our axiomatization of lists of booleans can be easily extended to an axiomatization of lists in which both booleans and other lists may be stored. In effect, this would be an axiomatization of directed acyclic graphs (dags) — a datatype in which sharing is really essential.

One could argue that our axioms are apparently rather complex for such a simple datatype as lists. However, many of these axioms have a regular form (e.g., idioms for non-destructiveness) and we could use further shorthands. More importantly, we would claim that there is a range of non-equivalent specifications of lists-in-memory, usefully describing different degrees of destructiveness, so the axioms need to be complex enough to capture these differences.

5 Specifying Disjointness

A notion of disjointness is useful in specifying memory operations. The previous section demonstrated this for the `delete` operation. Another example is a `copy` operation, which should produce a new and disjoint copy of a given list. Using our predicate for disjointness, this is captured by:

$$\forall m \forall l \in m \cdot \mathtt{copy}(m,l) \neq \bot \ \wedge (\mathtt{copy}(m,l) \to (m', \_) \implies m \lesssim m')$$

$$\forall m \forall l \in m \cdot \mathtt{copy}(m,l) \to (m', l') \implies (m,l) \sim (m', l') \wedge \forall \gamma \in m \cdot l' \perp_{m'} \gamma$$

In a concrete model such as the pointer model, disjointness has a clear meaning. Pleasingly, it turns out that disjointness may be defined in an abstract, behavioural manner for any memory algebra. This abstract notion, when applied to pointer models of datatypes, yields the expected form of separation, and when applied to other, e.g., functional models, also gives very natural results.

The disjointness predicate has the form $\gamma \perp_m \delta$, where γ and δ are tuples of of objects valid in m. Formally, we add the following new formula:

$$\phi ::= ... \mid \alpha \perp_m \alpha$$

We define the interpretation of disjointness coinductively using non-interference: two objects may be treated as disjoint if manipulations on one of them cannot affect the other, and vice versa. In particular, the disjointness of two objects is relative to the operations one may use on them.

Definition 4. *Let $\mathcal{F}$ be a set of memory operations from a memory signature.* Disjointness with respect to $\mathcal{F}$ *is the greatest memory-indexed family of symmetric relations on valid tuples of objects and such that if $\gamma \perp_m \delta$, then:*

- *if v is an object of abstract sort, then $\gamma + (v) \perp_m \delta$,*

– *if γ is of length n, $F : \{1, \ldots, N\} \to \{1, \ldots, n\}$ is any function and $f \in \mathcal{F}$ is of the appropriate type, and if $A[f](m, F(\gamma)) = (m', \gamma')$, then:*
(i) δ is valid in m', (ii) $(m', \delta) \sim (m, \delta)$, and (iii) $(\gamma|_{m'} + \gamma')\bot_{m'}\delta$.

Definition 4 imposes the *frame condition* that manipulation of objects cannot affect other objects which are disjoint in the same memory. However, because the disjointness notion is relativised to a particular memory, it is possible for an operation to destructively combine two objects from one memory, and, in the new memory, for those objects to be no longer disjoint, or to become invalid. An example of the second case is the familiar destructive append, which invalidates the first object; an example of the first case is $\texttt{smash\_tails}(l_1, l_2)$ which coalesces two disjoint objects l_1 and l_2 so that they share the longest possible suffix.

Disjointness is anti-monotone with respect to the set of operations $\mathcal{F}$, and for $\mathcal{F} = \emptyset$ all valid tuples are disjoint in all memories. When it is left implicit, we take all memory operations to be in $\mathcal{F}$. In the pointer model of lists, disjointness w.r.t. all the operations is just real disjointness of two lists in memory. In particular, any list is disjoint with the nil list 0. In the algebraic model of lists, meanwhile, the disjointness relation is the total relation — we even have $l\bot_m l$ for any list l. All this is not surprising. But if, in the pointer model, we consider disjointness w.r.t. operations other than `delete`, then we also get the total relation. This is true even when the "real" lists overlap in memory — the overlap cannot have consequence and so is ignored. One could claim that a model deserves the name "functional" just in case its disjointness relation is total.

At the other end of the spectrum, suppose we add $\texttt{zero\_memory} : \mu \to \mu$ to our signature, and implement it in the pointer model as a constant function which takes any memory m to a memory in which all addresses are on the free list. If we consider disjointness w.r.t. the set $\mathcal{F} = \{\texttt{zero\_memory}\}$, then only pairs of sequences of nil lists and booleans are disjoint. We don't have $0\bot_m l$, where 0 is the empty list and l is non-empty, since zeroing memory on the left side invalidates l. If we now set $\mathcal{F}$ to contain all of the operations, then even $()\bot_m()$ no longer holds, since it is possible to first produce a non-empty list and then invalidate it, as suggested above.

6 A Specification of Pointers

Now we specify a datatype of pointers. This is a purely "imperative" datatype, with no functional aspects. In the next section we show how to use it to implement datatypes such as lists. The signature has a sort `pointer` and operations:

$\texttt{0} : \mu \to \mu \times \texttt{pointer}$
$\texttt{new} : \mu \to \mu \times \texttt{pointer}$
$\texttt{is0} : \mu \times \texttt{pointer} \to \mu \times \texttt{bool}$
$\texttt{dispose} : \mu \times \texttt{pointer} \to \mu$
$\texttt{set}_1, \texttt{set}_2 : \mu \times \texttt{pointer} \times \texttt{pointer} \to \mu$
$\texttt{setbool}_1, \texttt{setbool}_2 : \mu \times \texttt{pointer} \times \texttt{bool} \to \mu$
$\texttt{val}_1, \texttt{val}_2 : \mu \times \texttt{pointer} \to \mu \times \texttt{pointer}$
$\texttt{valbool}_1, \texttt{valbool}_2 : \mu \times \texttt{pointer} \to \mu \times \texttt{bool}$

The operation `0` returns a special, constant pointer, and `is0` tests identity of this pointer. In any pointer two values may be stored and retrieved, each being either another pointer or a boolean. Pointers may be created and disposed of.

The axioms specifying pointers are as follows:

$$\forall m \cdot \texttt{new}(m) \neq \bot \ \wedge \ (\texttt{new}(m) \to (m', \_) \implies m \lesssim m') \tag{10}$$

$$\forall m \forall p \in m \cdot \texttt{val}_i(m, p) \neq \bot \ \wedge \ (\texttt{val}_i(m, p) \to (m', \_) \implies m \lesssim m') \tag{11}$$

$$\forall m_0 \cdot \texttt{0}(m_0) \to (m, p) \implies \texttt{is0}(m, p) \to (\_, b) \implies b = \texttt{t} \tag{12}$$

$$\forall m_0 \cdot \texttt{new}(m_0) \to (m, p) \implies \texttt{is0}(m, p) \to (\_, b) \implies b = \texttt{f} \tag{13}$$

These are similar to ones given earlier for list operations. An axiom analogous to (10) is required for `0` and ones analogous to (11) for `is0` and for `valbool`.

$$\forall m_0 \cdot \texttt{new}(m_0) \to (m, p) \implies \forall q \in m \cdot \texttt{set}_i(m, p, q) \neq \bot \wedge (\texttt{set}_i(m, p, q) \to m' \implies m_0 \lesssim m') \tag{14}$$

$$\forall m_0 \cdot \texttt{new}(m_0) \to (m, p) \implies \texttt{dispose}(m, p) \neq \bot \wedge (\texttt{dispose}(m, p) \to m' \implies m_0 \lesssim m') \tag{15}$$

$$\forall m_1 \cdot \texttt{new}(m_1) \to (m_2, p) \implies \forall q \in m_2 \cdot \texttt{set}_i(m_2, p, q) \to m_3 \implies \texttt{val}_i(m_3, p) \to (m_4, q') \implies (m_3, q) \lesssim (m_4, q') \tag{16}$$

Axiom (14) says that storing in a pointer does not modify objects that existed before the pointer was created; a similar axiom is needed for `setbool`. Axiom (15) says that disposing a pointer does not affect objects that existed before its creation. Axiom (16) and a similar axiom for the boolean case guarantee that storing an object and then loading it gives the same (or a behaviourally equivalent) object back. Behavioural equivalence for pointers is crucial: it means that an implementation may keep internal information (e.g., needed for garbage collection) in pointers, as long as the outside world cannot access it.

We can define a model for pointers by altering the model of lists as follows:

- memories must keep both addresses 0 and 1 on the free list (they are used to denote booleans), and to keep 2 off the free list, initialised to $(0, 0)$ (it will be used as the `0` pointer),
- the set of valid pointers in m is the greatest set of addresses not on the free list and such that if a is valid and $m(a) = (a_1, a_2)$, then $a_i = 0$, $a_i = 1$ or a_i is valid in m (for $i = 1, 2$),
- `0` returns the address 2; `is0` returns true called on 2, false otherwise,
- `new` allocates a new pointer and sets its fields to 0,
- $\texttt{val}_i$ and $\texttt{set}_i$ just return and set the appropriate fields under the given address; the boolean versions do the same, with 0 as false and 1 as true,
- `dispose` adds the given address to the free list.

One could also imagine a "lazy" model, in which `dispose` would defer its work, adding addresses to the free list later on when other operations are called.

7 Implementations

So far we haven't said how memory specifications may be *implemented*, that is, how one can define memory algebras. We now show how this can be done, and show how the correctness of implementations may be proved. We do not consider how to construct implementations "ex nihilo", but rather how to implement one datatype making use of other datatypes. This approach is reasonable, because one may assume that simple datatypes are given as built-in.

Let Δ be a memory signature containing the abstract sort `bool`. *Programs* over Δ are expressions P_λ of the following form:

$$\begin{array}{l} P_\lambda ::= \lambda(\alpha) \cdot P \\ P \ ::= P; P \mid x \to x \mid f(\alpha) \to (\alpha) \mid \text{if } e \text{ then } P \mid \text{return}(\alpha) \mid \text{self}(\alpha) \end{array}$$

Here, α denotes tuples of variables, implicitly typed by sorts from Δ, e denotes boolean expressions built using abstract operations and variables of abstract sorts, and f denotes memory operations from Δ. No variable is allowed to appear on the right hand side of the binding "$\to$" more than once. The instruction "self(α)" recursively calls the program being run, while "return(α)" terminates the computation (all recursive calls). It is required that the last instruction in any program is either a self, or a return. Type-soundness is also enforced, i.e., if a memory operation f is invoked with arguments being variables of sorts $s_1, \ldots, s_n$ and results being variables of sorts $t_1, \ldots, t_k$, then we have $f : \mu \times (s_1 \times \cdots \times s_n) \to \mu \times (t_1 \times \cdots \times t_k)$ in Δ. A program has *type* $s_1 \times \cdots \times s_n \to t_1 \times \cdots \times t_k$ if it binds, under the λ, variables of sorts $s_1, \ldots, s_n$, if it passes variables of such types via self, and if it invokes return with variables of type $t_1, \ldots, t_k$.

An *implementation* of a memory signature Σ by Δ is a map I taking any memory sort in Σ to a memory sort in Δ, and any memory operation $f : \mu \times (s_1 \times \cdots \times s_n) \to \mu \times (t_1 \times \cdots \times t_k)$ to a program over the signature Δ of type $I(s_1) \times \cdots \times I(s_n) \to I(t_1) \times \cdots \times I(t_k)$.

For example, suppose we want to implement lists, as defined in Sect. 3, using pointers, as defined in the previous section. We can define this by:

$$\begin{array}{ll} \texttt{list} & := \texttt{pointer} \\ \texttt{nil} & := \lambda() \cdot \texttt{0} \to p;\ \text{return}(p) \\ \texttt{isnil} & := \lambda(p) \cdot \texttt{is0}(p) \to b;\ \text{return}(b) \\ \texttt{cons} & := \lambda(b, p) \cdot \texttt{new} \to p_1; \\ & \quad \texttt{setbool}_1(p_1, b) \to (); \\ & \quad \texttt{set}_2(p_1, p) \to (); \\ & \quad \text{return}(p_1) \end{array} \qquad \begin{array}{ll} \texttt{hd} & := \lambda(p) \cdot \texttt{valbool}_1(p) \to b;\ \text{return}(b) \\ \texttt{tl} & := \lambda(p) \cdot \texttt{val}_2(p) \to p';\ \text{return}(p') \\ \texttt{delete} & := \lambda(p) \cdot \texttt{is0}(p) \to b; \\ & \quad \text{if } b \text{ then return}(); \\ & \quad \texttt{val}_2(p) \to p'; \\ & \quad \texttt{dispose}(p) \to (); \\ & \quad \text{self}(p') \end{array}$$

We define the semantics of programs in the obvious way, with infinitely looping programs causing non-definedeness, $\perp$. The semantics of programs induces a semantics of implementations: for any implementation $I : \Sigma \to \Delta$ and any memory algebra B over Δ, the semantics of Δ-programs gives us a memory algebra $B|_I$ over Σ.

Proving that implementations are correct. An implementation of lists by pointers is *correct*, if assuming that the pointer axioms of Sect. 5 hold, then so do the list axioms of Sect. 3. This will guarantee that if B satisfies the pointer axioms, then $B|_I$ satisfies the list axioms.

For any implementation $I : \Sigma \to \Delta$ one can define a set $\mathrm{Sen}(I)$ of formulas over an extended signature $\Delta \cup \Sigma$ which define the operations in Σ. For example, the definition of `nil` leads to the two formulae:

$$\forall m \cdot \mathtt{nil}(m) \to (m', p) \implies \mathtt{0}(m) \neq \bot \wedge (\mathtt{0}(m) \to (m'', p') \implies (m', p) \lesssim (m'', p'))$$

$$\forall m \cdot \mathtt{0}(m) \to (m', p) \implies \mathtt{nil}(m) \neq \bot \wedge (\mathtt{nil}(m) \to (m'', p') \implies (m', p) \lesssim (m'', p'))$$

In a similar manner, definitions of other operations may be generated. Armed with these formulas and the axioms defining pointers, we may now attempt to prove the axioms defining lists. Consider, for example, axiom (6). Thanks to the definition of `cons` and `hd` in $\mathrm{Sen}(I)$, this is equivalent to:

$$\forall m \forall b \forall p \in m \cdot \mathtt{new}(m) \to (m_1, p') \implies \mathtt{setbool}_1(m_1, p', b) \to m_2 \implies \mathtt{set}_2(m_2, p', p) \to m' \implies \mathtt{valbool}_1(m', p') \neq \bot \wedge (\mathtt{valbool}_1(m', p') \to (m'', b') \implies (m', b) \lesssim (m'', b'))$$

This is indeed a consequence of the pointer axioms. Next consider axiom (8). It is equivalent to the following two formulas, corresponding to two branches of the if in `delete`'s definition:

$$\forall m_0 \cdot \mathtt{0}(m_0) \to (m, p) \implies \mathtt{is0}(m, p) \neq \bot \wedge (\mathtt{is0}(m, p) \to (m', b) \implies b = \mathtt{t} \implies m_0 \lesssim m') \quad (17)$$

$$\forall m_0 \cdot \mathtt{0}(m_0) \to (m, p) \implies \mathtt{is0}(m, p) \neq \bot \wedge (\mathtt{is0}(m, p) \to (m_1, b) \implies b = \mathtt{f} \implies \mathtt{val}_2(m_1, p) \to (m_2, p') \implies \mathtt{dispose}(m_2, p) \to m_3 \implies \mathtt{delete}(m_3, p') \to m' \implies m_0 \lesssim m') \quad (18)$$

In this simple case, because the second branch of the if always holds, we can prove the second formula even without again using the definition of `delete`, which may be found in $\mathrm{Sen}(I)$. But in general, we may repeatedly use the definitions in $\mathrm{Sen}(I)$ — this is, for example, necessary when proving that axiom (9) holds.

It can be shown that the method presented above is indeed sound. The notation $\Phi \models_\Sigma \varphi$ below means that any Σ-algebra satisfying all the formulas from the set Φ satisfies φ as well. The notation $\Phi|_I \models_\Sigma \varphi$, where $I : \Sigma \to \Delta$ is an implementation, Φ is a set of Δ-formulas and φ is a Σ-formula means that for any Δ-algebra B satisfying all the formulas from Φ, the Σ-algebra $B|_I$ satisfies the formula φ. We say that I *is an identity* on a subsignature Σ_0 of Σ if it is an identity on sorts and if, for any symbol f in Σ_0, we have $I(f) = \lambda(\alpha) \cdot f(\alpha) \to (\beta); \mathrm{return}(\beta)$.

Theorem 1 *If $I : \Sigma \to \Delta$ is an implementation which is an identity on $\Sigma \cap \Delta$, Φ is a set of Δ-formulas and φ is a Σ-formula, and if $\Phi \cup \mathrm{Sen}(I) \models_{\Sigma\cup\Delta} \varphi$ then $\Phi|_I \models_\Sigma \varphi$.*

Proof (sketch). Assume B satisfies $B \models_\Delta \Phi$. Let B' be the union (amalgamation) of B and $B|_I$, i.e., an algebra over $\Sigma\cup\Delta$. This union may be formed, since B and $B|_I$ coincide on $\Sigma\cap\Delta$. All observations in B' are also observations in B, because operations in $B|_I$ are defined in terms of operations of B. Therefore B and B' satisfy the same Δ-formulas; in particular, $B' \models_{\Sigma\cup\Delta} \Phi$. By construction of $\mathrm{Sen}(I)$ we also have $B' \models_{\Sigma\cup\Delta} \mathrm{Sen}(I)$. By assumption we then have $B' \models_{\Sigma\cup\Delta} \varphi$. It can be shown by induction on φ that this implies $B' \models_\Sigma \varphi$, since there are no more observations over Σ than over $\Sigma \cup \Delta$. □

This theorem provides a sound method of proving implementations correct. In general, this method is not complete. This is because the relations $\sim$ and $\bot_m$ over the signature Σ are coarser than the same relations over $\Sigma \cup \Delta$, where more operations exist. Consider, e.g., sets implemented by lists, with repetitions allowed in the representations. Then two lists differing only in the number of repetitions will be considered equivalent over Σ (i.e., with respect to the set operations). But over $\Sigma \cup \Delta$ (i.e., with respect to the list operations) they are not equivalent any more.

A second source of incompleteness is non-termination: the defining sentences in $\mathrm{Sen}(I)$ don't force non-terminating memory operations to actually return $\bot$. To circumvent this problem, we consider, for a set of Δ-formulas Φ, only implementations $I : \Sigma \to \Delta$ that are *total* w.r.t. Φ, that is, implementations such that in $B|_I$ memory operations are total for any algebra B s.t. $B \models_\Delta \Phi$.

Save for the above phenomena, the presented proof method is complete. In other words, for total implementations, completeness is guaranteed if all observations in $\Sigma \cup \Delta$ may be conducted in Δ as well:

Theorem 2 *If $I : \Sigma \to \Delta$ is an implementation, Σ contains Δ and I is an identity on Δ, then for any set Φ of Δ-formulas such that I is total w.r.t. Φ, and for any Σ-formula φ we have $\Phi \cup \mathrm{Sen}(I) \models_\Sigma \varphi$ iff $\Phi|_I \models_\Sigma \varphi$.*

Proof. By the previous theorem and since I is an identity on $\Sigma \cap \Delta = \Delta$, only the "if" direction needs to be shown. Assume $B \models_\Sigma \Phi \cup \mathrm{Sen}(I)$ and let B_0 be the restriction of B to Δ. Then $B_0|_I \models_\Sigma \Phi$, because $B_0|_I$ and B coincide on Δ and all operations in Σ are defined by I in terms of operations from Δ, so no new observations exist in $B_0|_I$. Thus $B_0|_I \models_\Sigma \varphi$. We also have $B_0 \models_\Delta \Phi$, since $B \models_\Sigma \Phi$ and B_0 is a restriction of B. Since I is total w.r.t. Φ, this implies that in $B_0|_I$ all operations are total. At the same time, B and $B_0|_I$ coincide on Δ and $B \models_\Sigma \mathrm{Sen}(I)$. But there can be only one total algebra which coincides on Δ with B and satisfies $\mathrm{Sen}(I)$, and so $B = B_0|_I$. Hence, $B \models_\Sigma \varphi$, as required. □

Another way to ensure completeness is to allow various versions of the relations $\sim$ and $\bot_m$ to appear in axioms. If we tag these relations by the appropriate legal sets of observations (in the form of programs), and if by $I(\varphi)$ we

denote the formula with appropriate tagging, then we would get the equivalence $\Phi \cup \mathrm{Sen}(I) \models_{\Sigma\cup\Delta} I(\varphi)$ *iff* $\Phi|_I \models_\Sigma \varphi$. The obvious downside of using $I(\varphi)$ is that we have to deal with a more complex logic. But consider a logic obtained by allowing the relations $\sim$ and $\bot_m$ to appear as *premises* in implications in the formulas: in this case the proposed proof method ceases to be sound, because the new observations work both ways, harming both completeness and soundness. If we use the translation $I(\varphi)$, soundness and completeness are preserved.

8 Conclusions

We introduced an algebraic scheme for specifying and proving correct pointer programs using their observable behaviour. The mechanism is based on first-order and behavioural principles, and so could be adopted within existing algebraic specification frameworks. Further investigations are warranted, including the study of a suitable proof system and extensions of the language. One extension would be the aforementioned generalised equivalence and separation relations; another would be a more expressive logic where coinductive predicates such as disjointness are definable directly.

While we aimed at being more abstract than existing work based on concrete memory models, it is clear that we could be more abstract still. Recasting our work in a higher-order setting would allow us to make comparisons with related work in programming language semantics based on monads and coalgebras (e.g. [7, 8]), as would studying model-theoretic foundations. It is natural to want to specify parts of our models to be generated inductively and require an initial subalgebra interpretation, while the observational relations have coinductive characterisations related to final coalgebras.

Finally, we would like to revisit our starting point of specifying interaction between resources and their consumption in general. Although our focus was on models of memory in this paper, there is nothing special about "memories" in our approach that forces them to have this interpretation. Getting closer to real machines, we might add stack operations to our memory signature. To describe space usage and consider limited memories, we can add a `size` function on memories together with an *out-of-memory* exception. Imagining a different interpretation entirely, we could conceive of the sort μ to denote a database of statistical data in tables; computations on this data may interfere when data sources are combined but not independent.

Related work. As mentioned at the start, there is much current activity in studying pointer programs, their logics and correctness proofs, as well as more abstract notions of resource. Space precludes a survey; we mention only a few connections. First, program logics designed for managing aliasing (e.g., Reynolds' Separation Logic [1], Honda et al's process algebra inspired approach [2]) aim to simplify proofs of pointer manipulating programs, particularly for better modularity. Our notion of machine generalises the concrete model considered by Separation Logic, but our low-level language of assertions differs, in particular making the global heap explicit. Nonetheless our equivalence notion allows both strong assertions

like $m \lesssim m'$ which amount to *global* frame conditions, as well as *local* equivalences such as $(m, a) \sim (m, a')$ which, conceptually, are restricted to reachable heap portions.

Elsewhere, other authors have found equivalence relations like ours useful. For example, Calcagno and O'Hearn [9] pointed out a need for an observational approach in Separation Logic in the presence of garbage collection, because otherwise assertions in the logic can distinguish programs which ought to be considered identical. Benton [10, 5] studies correctness proofs for program analyses and transformations using a relational Hoare logic, noting that in general desirable program equivalences are context-sensitive.

In the algebraic domain, perhaps surprisingly, nobody seems to have begun from the same simple definitions as we gave in Sect. 2. But there are certainly a number of rich mechanisms for treating state in dynamic systems, for example, Hidden Algebra [11] and SB-CASL [12], as well as work on notions of behavioural equivalence between algebras and proof mechanisms (see e.g., [13]). We hope that one of our contributions in this work is to open a way to bring together this strand of work in algebraic specification with the recent work in program logics.

Acknowledgements. This work was supported by the British Council; DA was also supported by the EC project MOBIUS (IST-15905), PH by the EC project SENSORIA (IST-16004). We're grateful for feedback from referees and colleagues.

References

1. Reynolds, J.: Separation logic: A logic for shared mutable data structures. In: LICS 2002. (2002) 55–74
2. Honda, K., Yoshida, N., Berger, M.: An observationally complete program logic for imperative higher-order frame rules. In: LICS'05. (2005) 270–279
3. Aspinall, D., Beringer, L., Hofmann, M., Loidl, H.W., Momigliano, A.: A program logic for resources. Theoretical Computer Science (2007) Accepted.
4. Petersen, L., Harper, R., Crary, K., Pfenning, F.: A type theory for memory allocation and data layout. In: POPL'03. (2003) 172–184
5. Benton, N., Kennedy, A., Hofmann, M., Beringer, L.: Reading, writing and relations. In: APLAS'06. Volume 4279 of LNCS. (2006) 114–130
6. Pym, D., O'Hearn, P., Yang, H.: Possible worlds and resources: The semantics of BI. Theoretical Computer Science **315**(1) (2004) 257–305
7. Jacobs, B., Poll, E.: Coalgebras and monads in the semantics of Java. TCS **291**(3) (2003) 329–349
8. Schröder, L., Mossakowski, T.: Monad-independent dynamic logic in HasCasl. J. Log. Comput. **14**(4) (2004) 571–619
9. Calcagno, C., O'Hearn, P., Bornat, R.: Program logic and equivalence in the presence of garbage collection. TCS **298**(3) (2003) 557–581
10. Benton, N.: Simple relational correctness proofs for static analyses and program transformations. In: POPL'04. (2004) 14–25
11. Goguen, J., Malcolm, G.: A hidden agenda. TCS **245**(1) (2000) 55–101
12. Baumeister, H., Zamulin, A.: State-based extension of CASL. In: IFM 2000. Volume 1945 of LNCS. (2000) 3–24
13. Hennicker, R., Bidoit, M.: Observational logic. In: AMAST'98. Volume 1548 of LNCS. (1999) 263–277