Amortised Resource Analysis with Separation Logic

Robert Atkey

LFCS, School of Informatics, University of Edinburgh
bob.atkey@ed.ac.uk

Abstract. Type-based amortised resource analysis following Hofmann and Jost—where resources are associated with individual elements of data structures and doled out to the programmer under a linear typing discipline—have been successful in providing concrete resource bounds for functional programs, with good support for inference. In this work we translate the idea of amortised resource analysis to imperative languages by embedding a logic of resources, based on Bunched Implications, within Separation Logic. The Separation Logic component allows us to assert the presence and shape of mutable data structures on the heap, while the resource component allows us to state the resources associated with each member of the structure.

We present the logic on a small imperative language with procedures and mutable heap, based on Java bytecode. We have formalised the logic within the Coq proof assistant and extracted a certified verification condition generator. We demonstrate the logic on some examples, including proving termination of in-place list reversal on lists with cyclic tails.

1 Introduction

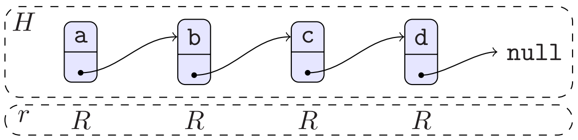
Tarjan, in his paper introducing the concept of amortised complexity analysis [15], noted that the statement and proof of complexity bounds for operations on some data structures can be simplified if we can conceptually think of the data structure as being able to store "credits" that are used up by later operations. By setting aside credit inside a data structure to be used by later operations we amortise the cost of the operation over time. In this paper, we propose a way to merge amortised complexity analysis with Separation Logic [12,14] to formalise some of these arguments.

Separation Logic is built upon a notion of resources and their separation. The assertion $A * B$ holds for a resource if it can be split into two resources that make A true and B true respectively. Resource separation enables local reasoning about mutation of resources; if the program mutates the resource associated with A, then we know that B is still true on its separate resource.

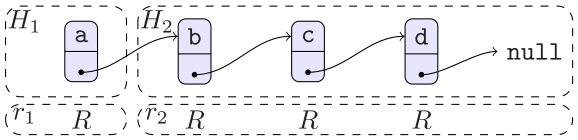
For the purposes of complexity analysis, we want to consider resource consumption as well as resource mutation, e.g. the consumption of time as a program executes. To see how Separation Logic-style reasoning about resources helps in this case, consider the standard inductively defined list predicate from Separation Logic, augmented with an additional proposition R denoting the presence of a consumable resource for every element of the list:

$$\begin{aligned}\mathsf{list}_{\mathsf{R}}(x) \equiv\ & x = \mathsf{null} \wedge \mathsf{emp} \\ & \vee \exists y, z.\ [x \overset{\mathsf{data}}{\mapsto} y] * [x \overset{\mathsf{next}}{\mapsto} z] * R * \mathsf{list}_{\mathsf{R}}(z)\end{aligned}$$

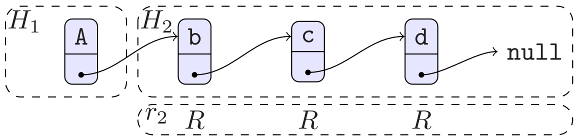
We will introduce the assertion logic properly in Section 4 below. We can represent a heap H and a consumable resource r that satisfy this predicate graphically:



So we have $r, H \models \mathsf{list}_{\mathsf{R}}(x)$, assuming x points to the head of the list. Here $r = R \cdot R \cdot R \cdot R$—we assume that consumable resources form a commutative monoid—and r represents the resource that is available for the program to use in the future. We can split H and r to separate out the head of the list with its associated resource:



This heap and resource satisfy $r_1 \cdot r_2, H_1 \uplus H_2 \models [x \overset{\mathsf{data}}{\mapsto} \mathsf{a}] * [x \overset{\mathsf{next}}{\mapsto} y] * R * \mathsf{list}_{\mathsf{R}}(y)$, where $H_1 \uplus H_2 = H$, $r_1 \cdot r_2 = r$ and we assume that y points to the `b` element. Now that we have separated out the head of the list and its associated consumable resource, we are free to mutate the heap H_1 and consume the resource r_1 without it affecting the tail of the list, so the program can move to a state:



where the head of the list has been mutated to `A` and the associated resource has been consumed; we do not need to do anything special to reason that the tail of the list and its associated consumable resource are unaffected.

The combined assertion about heap and consumable resource describes the current shape and contents of the heap and also the available resource that the program may consume in the future. By ensuring that, for every state in the program's execution, the resource consumed plus the resource available for consumption in the future is less than or equal to a predefined bound, we can ensure that the entire execution is resource bounded. This is the main assertion of soundness for our program logic in Section 3.4.

By intermixing resource assertions with Separation Logic assertions about the shapes of data structures, as we have done with the resource carrying list_R predicate above, we can specify amounts of resource that depend on the shape of data structures in memory. By the definition of list_R, we know that the amount of resource available to the program is proportional to the length of the list, without having to do any arithmetic reasoning about lengths of lists.

The association of resources with parts of a data structure is exactly the banker's approach to amortised complexity analysis proposed by Tarjan [15].

Our original inspiration for this work came from the work of Hofmann and Jost [9] on the automatic heap-space analysis of functional programs. Their analysis associates with every element of a data structure a permission to use a piece of resource (in their case, heap space). This resource is made available to the program when the data structure is decomposed using pattern matching. When constructing part of a data structure, the required resources must be available. A linear type system is used to ensure that data structures carrying resources are not duplicated: this would entail duplication of consumable resource. This scheme was later extended to imperative object-oriented languages [10,11], but still using a type-based analysis.

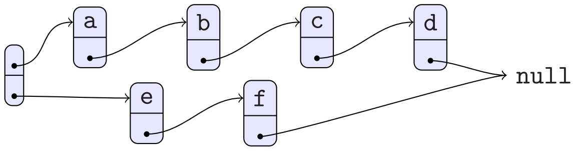
Contributions We summarise the content and contributions of this work:

– In Section 3, we define a program logic that allows mixing of assertions about heap shapes, in the tradition of separation logic, and assertions about future consumable resources. Tying these together allows us to easily state resource properties in terms of the shapes of heap-based data structures, rather than extensional properties such as their size or contents. We have also formalised the soundness proof of our program logic in Coq, along with a verified verification condition generator.
– In Section 5, we define a restricted subset of the assertion logic that allows us to perform effective proof search and inference of resource annotations. A particular feature of the way this is set up is that, given loop invariants that talk only about the the shape of data structures, we can infer necessary resource bounds.
– In Sections 2 and 6, we demonstrate the logic on two small examples, showing how a mixture of amortised resource analysis and Separation Logic can be used to simplify resource-aware specifications, and how it can be used to prove termination in the presence of cyclic structures in the heap.

2 Motivating Example: Functional Queues

Before defining our program logic, we give another example to demonstrate how amortised reasoning is easier than the traditional approach of keeping a global counter for consumed resources as an auxiliary "ghost" variable in the proof. This example is a standard one for introducing amortised complexity analysis, but here we look at the specifications of operations on an imperative data structure taking into account their resource consumption.

We consider functional queues, where a queue is represented by a pair of lists:



The top list represents the head of the queue, while the bottom list represents the tail of the queue in reverse; thus this structure represents the queue $[a, b, c, d, f, e]$. When we enqueue a new element, we add it to the head of the bottom list. To dequeue an element, we remove it from the head of the top list. If the top list is empty, then we reverse the bottom list and change the top pointer to point to it, changing the bottom pointer to point to `null`.

When determining the complexity of these operations, it is obvious that the enqueue operation is constant time, but the dequeue operation either takes constant time if the top list is empty, or time linear in the size of the bottom list, in order to perform the reversal. If we were to account for resource usage by maintaining a global counter then we would have to expose the lengths of the two lists in specification of the enqueue and dequeue instructions. So we would need a predicate $\mathsf{queue}(x, h, t)$ to state that x points to a queue with a head and tail lists of lengths h and t respectively. The operations would have the specifications:

$$\{\mathsf{queue}(x, h, t) \wedge r_c = r_1\}\mathtt{enqueue}\{\mathsf{queue}(x, h, t+1) \wedge r_c = r_1 + R\}$$

$$\{\mathsf{queue}(x, 0, 0) \wedge r_c = r_1\}\mathtt{dequeue}\{\mathsf{queue}(x, 0, 0) \wedge r_c = r_1\}$$

$$\{\mathsf{queue}(x, 0, t+1) \wedge r_c = r_1\}\mathtt{dequeue}\{\mathsf{queue}(x, t, 0) \wedge r_c = r_1 + (1+t)R\}$$

$$\{\mathsf{queue}(x, h+1, t) \wedge r_c = r_1\}\mathtt{dequeue}\{\mathsf{queue}(x, h, t) \wedge r_c = r_1 + R\}$$

where r_c is a ghost variable counting the total amount of resource (time, in this case) consumed by the program, and R is the amount of resource required to perform a single list node manipulation. Note that we have had to give three specifications for `dequeue` for the cases when the queue is empty, when the head list is empty and when the head list has an element. The accounting for the sizes of the internals of the queue data structure is of no interest to clients of this data structure, these specifications will complicate reasoning that must be done by clients in order to use these queues.

Using amortised analysis, this specification can be drastically simplified. We associate a single piece of resource with each element of the tail list so that when we come to reverse the list we have the necessary resource available to reverse each list element. The queue predicate is therefore:

$$\mathsf{queue}(x) \equiv \exists y, z.[x \overset{\mathsf{head}}{\mapsto} y] * [x \overset{\mathsf{tail}}{\mapsto} z] * \mathsf{list}(y) * \mathsf{list}_{\mathsf{R}}(z)$$

where list is the standard Separation Logic list predicate, and list_R is the resource-carrying list predicate given above. The specifications of the operations now become straightforward:

$$\{\mathsf{queue}(x) * R * R\}\texttt{enqueue}\{\mathsf{queue}(x)\} \qquad \{\mathsf{queue}(x) * R\}\texttt{dequeue}\{\mathsf{queue}(x)\}$$

To enqueue an element, we require two resources: one to add the new element to the tail list, and one to "store" in the list so that we may use it for a future reversal operation. To dequeue an element, we require a single resource to remove an element from a list. If a list reversal is required then it is paid for by the resources required by the enqueue operation.

Once we have set the specification of queues to store one element of resource for every node in the tail list, we can use the resource annotation inference procedure presented in Section 5 to generate the resource parts of the specifications for the enqueue and dequeue operations.

3 A Program Logic for Heap and Resources

We define a logic that is capable of asserting facts about both the mutable heap and the consumable resources that a program has access to. Assertions about resources available to a program are intimately connected to the shapes of the data structures that it is manipulating. In this section, we introduce a simple programming language and a resource-aware program logic for it. We define a "shallow" program logic where we treat pre- and post-conditions and program assertions as arbitrary predicates over heaps and consumable resources. In the next section, we will layer on top a "deep" assertion logic where predicates are actually Separation Logic formulae augmented with resource assertions.

3.1 Semantic Domains

Assume an infinite set $\mathbb{A}$ of memory addresses. We model heaps as finite partial maps $\mathbb{H} = (\mathbb{A} \times \mathbb{F}) \rightharpoonup_{\mathit{fin}} \mathbb{V}$, where $\mathbb{F}$ ranges over field names and $\mathbb{V} = \mathbb{A}_\perp + \mathbb{Z}$ represents the values that programs can directly manipulate: possibly null addresses and integers. We write $\mathit{dom}(H)$ for the domain of a heap and $H_1 \# H_2$ for heaps with separate domains; $H_1 \uplus H_2$ denotes union of heaps with disjoint domains.

Consumable resources are represented as elements of an ordered monoid $(\mathbb{R}, \sqsubseteq, \cdot, e)$, where e is the least element. Example consumable resources include $(\mathbb{N}, \leq, +, 0)$ or $(\mathbb{Q}^{\geq 0}, \leq, +, 0)$ for representing a single resource that is consumed (e.g. time or space), or multisets for representing multiple named resources that may be consumed independently. The ordering on consumable resources is used to allow weakening in our assertion logic: we allow the asserter to assert that more resources are required by the program than are actually needed.

To talk about separated combinations of heaps and resources, we make use of a ternary relation on pairs of heaps and consumable resources, as is standard in the semantics of substructural logics [13]:

$$\begin{aligned} Rxyz \Leftrightarrow\ & H_1 \# H_2 \wedge H_1 \uplus H_2 = H_3 \wedge r_1 \cdot r_2 \sqsubseteq r_3 \\ & \text{where } x = (H_1, r_1), y = (H_2, r_2), z = (H_3, r_3) \end{aligned}$$

We extend the order on resources to pairs of heaps and resources by $(H_1, r_1) \sqsubseteq (H_2, r_2)$ iff $H_1 = H_2$ and $r_1 \sqsubseteq r_2$.

3.2 A Little Virtual Machine

The programming language we treat is a simple stack-based virtual machine, similar to Java bytecode without objects or virtual methods, but with mutable heap and procedures. There are two types: int and ref, corresponding to the two kinds of values in $\mathbb{V}$. We assume a set $\mathbb{P}$ of procedure names, where a procedure's name determines its list of argument types and its return type. Programs are organised into a finite set of procedures, indexed by their name and individually consisting of lists of instructions from the following collection:

$$\begin{aligned}\iota ::= {} & \mathsf{iconst}\ z \mid \mathsf{ibinop}\ op \mid \mathsf{pop} \mid \mathsf{load}\ n \mid \mathsf{store}\ n \mid \mathsf{aconst\_null} \\ & \mid \mathsf{binarycmp}\ cmp\ \mathit{offset} \mid \mathsf{unarycmp}\ cmp\ \mathit{offset} \mid \mathsf{ifnull}\ \mathit{offset} \mid \mathsf{goto}\ \mathit{offset} \\ & \mid \mathsf{new}\ desc \mid \mathsf{getfield}\ \mathit{fnm} \mid \mathsf{putfield}\ \mathit{fnm} \mid \mathsf{free}\ desc \mid \mathsf{consume}\ r \\ & \mid \mathsf{return} \mid \mathsf{call}\ pname\end{aligned}$$

These instructions—apart from consume—are standard, so we only briefly explain them. Inside each activation frame, the virtual machine maintains an operand stack and a collection of local variables, both of which contain values from the semantic domain $\mathbb{V}$. Local variables are indexed by natural numbers. The instructions in the first two lines of the list perform the standard operations with the operand stack, local variables and program counter. The third line includes instructions that allocate, free and manipulate structures stored in the heap. The instruction consume r consumes the resource r. The *desc* argument to new and free describe the structure to be created on the heap by the fields and their types. The fourth line has the procedure call and return instructions that manipulate the stack of activation frames.

Individual activation frames are tuples $\langle code, S, L, pc\rangle \in \mathsf{Frm}$ consisting of the list of instructions from the procedure being executed, the operand stack and local variables, and the program counter. The first two lines of instructions that we gave above only operate within a single activation frame, so we give their semantics as a small-step relation between frames: $\xrightarrow{frm} \subseteq \mathsf{Frm} \times \mathsf{Frm}$. This accounts for the bulk of instructions.

The third line of instructions includes those that manipulate the heap and consume resources. Their small-step operational semantics is modelled by a relation $\xrightarrow{mut} \subseteq \mathsf{Frm} \times \mathbb{H} \times \mathsf{Frm} \times \mathbb{H} \times \mathbb{R}$, which relates the before and after activation frames and heaps, and states the consumable resource consumed by this step.

A state of the full virtual machine is a tuple $\langle r, H, fs\rangle \in \mathsf{State}$, where r is the resource consumed to this point, h is the current heap, and fs is a list of activation frames. The small-step operational semantics of the full machine for some program prg is given by a relation $\longrightarrow_{prg} \subseteq \mathsf{State} \times \mathsf{State}$ which incorporates the $\xrightarrow{frm}$ and $\xrightarrow{mut}$ relations and also describes how the call and return instructions manipulate the stack of activation frames.

Finally, we use the predicate $s \downarrow H, r, v$ to indicate when the last activation frame is popped and the machine halts. The H, r and v are the final heap, the consumed resources and the return value respectively.

3.3 Program Logic

We annotate every procedure *pname* in the program with a pre-condition P_{pname} and a post-condition Q_{pname}. Pre-conditions are predicates over $\mathbb{V}^* \times \mathbb{H} \times \mathbb{R}$: lists of arguments to the procedure and the heap and available resource at the start of the procedure's execution. Post-conditions are predicates over $\mathbb{V}^* \times \mathbb{H} \times \mathbb{R} \times \mathbb{V}$: argument lists and the heap, remaining consumable resource and return value. Intermediate assertions in our program logic are predicates over $\mathbb{V}^* \times \mathbb{H} \times \mathbb{R} \times \mathbb{V}^* \times (\mathbb{N} \rightharpoonup \mathbb{V})$: argument lists, the heap, remaining consumable resource and the current operand stack and local variable store.

For our program logic, a proof that a given procedure's implementation *code* matches its specification (P, Q) consists of a map C from instruction offsets in *code* to assertions such that:

1. Every instruction's assertion is suitable for that instruction: for every instruction offset i in *code*, there exists an assertion A such that $C, Q \vdash i{:}code[i] : A$ and $C[i]$ implies A. Figure 1 gives the definition of $C, Q \vdash i{:}\iota : A$ for a selected subset of the instructions ι. The post-condition Q is used for the case of the `return` instruction.
2. The precondition implies the assertion for the first instruction: for all arguments *args*, heaps H and consumable resources r, $P(args, H, r)$ implies $C[0](args, H, r, [], \ulcorner args \urcorner)$, where $[]$ denotes the empty operand stack, and $\ulcorner\cdot\urcorner$ maps lists of values to finite maps from naturals to values in the obvious way.

When condition 1 holds, we write this as $C \vdash code : Q$, indicating that the procedure implementation *code* has a valid proof C for the post-condition Q.

3.4 Soundness

We say that an activation frame is safe if there is a proof for the code being executed in the frame such that the requirements of the next instruction to be executed are satisfied. Formally, a frame $f = \langle code, S, L, pc\rangle$ is safe for arguments *args*, heap H, resource r and post-condition Q, written $safeFrame(f, H, r, args, Q)$ if[1]:

1. There exists a certificate C such that $C \vdash code : Q$;
2. $C[pc]$ exists and $C[pc](args, r, H, S, L)$ holds.

[1] In this definition, and all the later ones in this section, we have omitted necessary assertions about well-typedness of the stack, local variables and the heap because they would only clutter our presentation.

$$C, Q \vdash i{:}\mathsf{consume}\ r_c : \lambda(\mathit{args}, r, H, S, L).\exists r'.r_c \cdot r' \sqsubseteq r \wedge C[i+1](\mathit{args}, r', H, S, L)$$

$$\begin{aligned} C, Q \vdash i{:}\mathsf{ifnull}\ n : \lambda(\mathit{args}, r, H, S, L).\forall a\ S'.S = a :: S' \Rightarrow & \\ & (a \neq \mathtt{null} \Rightarrow C[i+1](\mathit{args}, r, H, S', L)) \wedge \\ & (a = \mathtt{null} \Rightarrow C[n](\mathit{args}, r, H, S', L)) \end{aligned}$$

$$C, Q \vdash i{:}\mathsf{call}\ \mathit{pname} :$$

$$\begin{aligned} \lambda(\mathit{args}, r, H, S, L).\forall \mathit{args}'\ S'.S = \mathit{args}'@S' \Rightarrow & \\ & \exists (H_1, r_1)\ (H_2, r_2). \\ & \quad R(H_1, r_1)(H_2, r_2)(H, r) \wedge \\ & \quad P_{\mathit{pname}}(\mathit{args}', H_1, r_1) \wedge \\ & \quad \forall v\ (H_1', r_1'). \\ & \qquad H_1' \# H_2 \Rightarrow \\ & \qquad Q_{\mathit{pname}}(\mathit{args}', H_1', r_1', v) \Rightarrow \\ & \qquad C[i+1](\mathit{args}', r_1' \cdot r_2, H_1' \uplus H_2, v :: S', L) \end{aligned}$$

Fig. 1. Program Logic Rules (Extract)

Safety of activation frames is preserved by steps in the virtual machine:

Lemma 1 (Intra-frame safety preservation)

1. *If* $\mathit{safeFrame}(f, H, r, \mathit{args}, Q)$ *and* $f \xrightarrow{\mathit{frm}} f'$*, then* $\mathit{safeFrame}(f', H, r, \mathit{args}, Q)$*.*
2. *If* $\mathit{safeFrame}(f, H_1, r, \mathit{args}, Q)$ *and* $H_1 \# H_2$ *and* $H_1 \uplus H_2 = H$ *and* $f, H \xrightarrow{\mathit{mut}} f', H', r_c$*, then there exists* H_1' *and* r' *such that* $H_1' \# H_2$ *and* $H_1' \uplus H_2 = H'$*,* $r_c \cdot r' \sqsubseteq r$ *and* $\mathit{safeFrame}(f', H_1', r', \mathit{args}, Q)$*.*

Remark 1. We pause for a moment to consider the relationship between our program logic and traditional Separation Logic. The second part of the previous lemma effectively states that execution steps for mutating instructions are *local*: for any other piece of heap that is present but not mentioned in its precondition, the execution of a mutating instruction will not affect it. This is usually expressed in Separation Logic by the Frame rule that states if we know that $\{P\}C\{Q\}$ holds, then $\{P * R\}C\{Q * R\}$ holds for any other resource assertion R. We do not have an explicit Frame rule in our program logic; application of the rule is implicit in the rule for the call instruction (so, conflatingly, the Frame rule is applied when we create a new activation frame). We do not have access to the Frame rule in order to modularly reason about the internals of each procedure, e.g. local reasoning about individual loops. This is partially a consequence of the unstructured nature of the bytecode that we are working with. It has not been a hindrance in the small examples that we have verified so far, but may well become so in larger procedures with multiple loops that need invariants. In such a case it may be useful to layer a hierarchical structure, matching the loops or other sub-program structure, on top of the unstructured bytecode that we have considered here in order to apply Frame rules and facilitate local reasoning inside procedures.

We have now handled all the instructions except the call and return instructions that create and destroy activation frames. To state soundness of our program logic for these we need to define what it means for a stack of activation frames to be safe. Intuitively, a stack of activation frames is a bridge between the overall arguments args_{top} and post-condition Q_{top} for the program and the arguments args_{cur} and post-condition Q_{cur} for the current activation frame, with respect to the current heap H and available consumable resources r, such that, when the current activation frame finishes, its calling frame on the top of the stack is safe. We write this as $\mathit{safeStack}(\mathit{fs}, H, r, \mathit{args}_{cur}, Q_{cur}, \mathit{args}_{top}, Q_{top})$.

Accordingly, we say that the empty frame stack is safe when $r = e$, $H = \mathsf{emp}$, $\mathit{args}_{cur} = \mathit{args}_{top}$ and $Q_{cur} = Q_{top}$. A frame stack $\mathit{fs} = \langle \mathit{code}, S, L, \mathit{pc}\rangle :: \mathit{fs}'$ is safe when there exists (H_1, r_1), (H_2, r_2), args, Q and C, A such that:

1. $R(H_1, r_1)(H_2, r_2)(H, r)$;
2. The code is certified: $C, Q \vdash \mathit{code}$;
3. The next instruction has pre-condition A: $C[\mathit{pc}] = A$;
4. When the callee returns, the instruction's pre-condition will be satisfied: for all $v \in \mathbb{V}, H_2', r_2'$ such that $H_2' \# H_1$ and $Q_{cur}(\mathit{args}_{cur}, H_2', r_2', v)$ holds, then $A(\mathit{args}, r_2' \cdot r_1, H_2' \uplus H_1, v :: S, L)$ holds.
5. The rest of the frame stack fs is safe when this activation frame returns: $\mathit{safeStack}(\mathit{fs}, H_2, r_2, \mathit{args}, Q, \mathit{args}_{top}, Q_{top})$.

Note how the *safeStack* predicate divides up the heap and consumable resource between the activation frames on the call stack; each frame hands a piece of its heap and consumable resource off to its callees to use.

Finally, we say that a state $s = \langle r_c, H, \mathit{fs}\rangle$ is safe for arguments args, post-condition Q and maximum resource r_{max}, written $\mathit{safeState}(s, \mathit{args}, Q, r_{max})$, if:

1. there exists an r_{future} such that $r_c \cdot r_{\mathit{future}} \sqsubseteq r_{max}$; and also
2. r_{future} and H split into (H_1, r_1) and (H_2, r_2), i.e. $R(H_1, r_1)(H_2, r_2)(H, r_{\mathit{future}})$;
3. there exists at least one activation frame: $\mathit{fs} = f :: \mathit{fs}'$ and arguments args_{cur} and post-condition Q_{cur}; such that
4. $\mathit{safeFrame}(f, H_1, r_1, \mathit{args}_{cur}, Q_{cur})$; and
5. $\mathit{safeStack}(fs, H_2, r_2, \mathit{args}_{cur}, Q_{cur}, \mathit{args}, Q)$.

The key point in the definition of *safeState* is that the assertions of the program logic talk about the resources that will be consumed in the *future* of the program's execution. Safety for a state says that when we combine the future resource requirements with resources that have been consumed in the past, r_c, then the total is less than the total resources r_{max} that are allowed for the execution.

Theorem 1 (Soundness)

1. *Assume that all the procedures in prg match their specifications. Then if* $\mathit{safeState}(s, \mathit{args}, Q, r_{max})$ *and* $s \longrightarrow_{prg} s'$ *then* $\mathit{safeState}(s', \mathit{args}, Q, r_{max})$.
2. *If* $\mathit{safeState}(s, \mathit{args}, Q, r_{max})$ *and* $s \downarrow H, r, v$, *then there exists an* r' *such that* $Q(\mathit{args}, H, r', v)$ *and* $r \sqsubseteq r_{max}$.

In the halting case in this theorem, the existentially quantified resource r' indicates the resources that the program still had available at the end of its execution. We are also guaranteed that when the program halts, the total resource that it has consumed will be less than the fixed maximum r_{max} that we have set, and moreover, by part 1 of the theorem, this bound has been observed at every step of the computation.

4 Deep Assertion Logic

In the previous section we described a program logic but remained agnostic as to the exact form of the assertions save that they must be predicates over certain domains. The shallow approach to stating makes the statement and soundness proof easier, but inhibits discussion of actual specifications and proofs in the logic. In this section we show that the logic of Bunched Implications, in its Separation Logic guise, can be used as a language for assertions in the program logic.

We defined three different types of assertion in the previous section: procedure pre- and post-conditions and intermediate assertions in methods. These all operate on heaps and consumable resources and the arguments to the current procedure, but differ in whether they talk about return values or the operand stack and local variables. To deal with these differences we assume that we have a set of terms in our logic, ranged over by $t, t_1, t_2, ...$, that at least includes logical variables and a constant *null* for representing the null reference, and also variables for representing the current procedure arguments, the return value and the operand stack and local variables as appropriate.

Formulae are built from at least the following constructors:

$$\begin{aligned}\phi ::= {} & t_1 \bowtie t_2 \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \phi_1 \rightarrow \phi_2 \mid \mathsf{emp} \mid \phi_1 * \phi_2 \mid \phi_1 \mathrel{-\!\!*} \phi_2 \mid \forall x.\phi \mid \exists x.\phi \\ & \mid [t_1 \overset{\mathsf{f}}{\mapsto} t_2] \mid R_r \mid \dots\end{aligned}$$

Where $\bowtie \in \{=, \neq\}$. We can also add inductively defined predicates for lists and list segments as needed. The only non-standard formula with respect to Separation Logic is R_r which represents the presence of some consumable resource r. The semantics of the assertion logic is given in Figure 2 as a relation $\models$ between environments and heap/consumable resource pairs and formulae. We assume a sensible semantics $[\![\cdot]\!]_\eta$ for terms in a given environment.

As a consequence of having an ordering on consumable resources, and our chosen semantics of emp, $*$ and $\mathrel{-\!\!*}$, our logic contains affine Bunched Implications as a sub-logic for reasoning about pure consumable resources.

Proposition 1. *If ϕ is a propositional BI formula with only R_r as atoms, then $r \models_{\mathrm{bi}} \phi$ iff $\eta, (r, h) \models \phi$.*

We have only considered a single separating connective, $\phi_1 * \phi_2$, which states that both the heap and consumable resources must be separated. Evidently, there

$$
\begin{array}{lll}
\eta, x \models \top & \text{iff always} \\
\eta, x \models t_1 \bowtie t_2 & \text{iff } [\![t_1]\!]_\eta \bowtie [\![t_2]\!]_\eta \\
\eta, x \models \mathsf{emp} & \text{iff } x = (h, r) \text{ and } h = \{\} \\
\eta, x \models [t_1 \overset{\mathsf{f}}{\mapsto} t_2] & \text{iff } x = (h, r) \text{ and } h = \{([\![t_1]\!]_\eta, f) \mapsto [\![t_2]\!]_\eta\} \\
\eta, x \models R_{r_i} & \text{iff } x = (h, r) \text{ and } r_i \sqsubseteq r \text{ and } h = \{\} \\
\eta, x \models \phi_1 \wedge \phi_2 & \text{iff } \eta, x \models \phi_1 \text{ and } \eta, x \models \phi_2 \\
\eta, x \models \phi_1 \vee \phi_2 & \text{iff } \eta, x \models \phi_1 \text{ or } \eta, x \models \phi_2 \\
\eta, x \models \phi_1 * \phi_2 & \text{iff exists } y, z. \text{ st. } Ryzx \text{ and } \eta, y \models \phi_1 \text{ and } \eta, z \models \phi_2 \\
\eta, x \models \phi_1 \rightarrow \phi_2 & \text{iff for all } y. \text{ if } x \sqsubseteq y \text{ and } \eta, y \models \phi_1 \text{ then } \eta, y \models \phi_2 \\
\eta, x \models \phi_1 \mathrel{-\!\!*} \phi_2 & \text{iff for all } y, z. \text{ if } Rxyz \text{ and } \eta, y \models \phi_1 \text{ then } \eta, z \models \phi_2 \\
\eta, x \models \forall v. \phi & \text{iff for all } a, \eta[v \mapsto a], x \models \phi \\
\eta, x \models \exists v. \phi & \text{iff exists } a, \eta[v \mapsto a], x \models \phi
\end{array}
$$

Fig. 2. Semantics of assertions

are two other possible combinations that allow sharing of heap or resources. Separation of resources, but sharing of heap:

$$
\begin{array}{ll}
\eta, x \models \phi_1 *_R \phi_2 \text{ iff} & x = (H, r) \text{ and exists } r_1, r_2. \text{ st.} \\
& r_1 \cdot r_2 \sqsubseteq r \\
& \text{and } \eta, (H, r_1) \models \phi_1 \text{ and } \eta, (H, r_2) \models \phi_2
\end{array}
$$

may be useful to specify that we have a single data structure in memory, but two resource views on it. However, we leave such investigation of alternative assertions to future work.

5 Automated Verification

In this section we describe an verification condition generation and proof search procedure for automated proving of programs against programs against specifications in our program logic, as long as procedures have been annotated with loop invariants. The restricted subset of separation logic that we use in this section is similar to the subset used by Berdine et al [3], though instead of performing a forwards analysis of the program, we generate verification conditions by backwards analysis and then attempt to solve them using proof search. As we demonstrate below, the proof search procedure is mainly guided by the structure of the program and the shape of the data structures that it manipulates. The resource annotations that are required can be inferred by linear programming.

5.1 Restricted Assertion Logic

Following Berdine et al, the restricted subset of the assertion logic that we use segregates assertions into pure data, heap and consumable resource sections.

$$
\begin{array}{lll}
\text{Data:} & P & := t_1 = t_2 \mid t_1 \neq t_2 \mid \top \\
\text{Heap:} & X & := [t_1 \overset{\mathsf{f}}{\mapsto} t_2] \mid \mathsf{lseg}(\Theta, t_1, t_2) \mid \mathsf{emp} \\
\text{Resource:} & R & := R_r \mid \top
\end{array}
$$

The terms that we allow in the data and heap assertions are either variables, or the constant null. The list segment predicate that we use here is defined inductively as:

$$\mathsf{lseg}(\Theta, x, y) \equiv (x = y \wedge \mathsf{emp}) \vee (\exists z, z'. \; [x \overset{\mathsf{data}}{\mapsto} z] * [x \overset{\mathsf{next}}{\mapsto} z'] * \Theta * \mathsf{lseg}(\Theta, z', y))$$

We restrict the pre- and post-conditions of each procedure to be of the form $\bigvee_i (\Pi_i \wedge (\Sigma_i * \Theta_i))$ and we use S to range over such formulae. The three components of each disjunct are lists of data, heap and resource assertions, with interpretations as in the following table.

Data:	Π	$:=$	$P_1, ..., P_n$	$(P_1 \wedge ... \wedge P_n)$
Heap:	Σ	$:=$	$X_1, ..., X_n$	$(X_1 * ... * X_n)$
Resource:	Θ	$:=$	$R_1, ..., R_n$	$(R_1 * ... * R_n)$

Note that, due to the presence of resource assertions in the lseg predicate, heap assertions may also describe consumable resources, even if the resource part of a disjunct is empty.

Finally, we have the set of goal formulae that the verification condition generator will produce and the proof search will solve.

$$G := S * G \mid S \mathrel{-\!\!*} G \mid S \mid G_1 \wedge G_2 \mid P \rightarrow G \mid \forall x.G \mid \exists x.G$$

Note that we only allow implications ($\rightarrow$ and $-\!\!*$) to appear in positive positions. This means that we can interpret them in our proof search as adding extra information to the context.

5.2 Verification Condition Generation

Verification condition generation is performed for each procedure individually by computing weakest preconditions for each instruction, working backwards from the last instruction in the method. To resolve loops, we require that the targets of all backwards jumps have been annotated with loop invariants S that are of the same form as the pre- and post-condition formulae from the previous section. We omit the rules that we use for weakest precondition generation since they are very similar to the rules for the shallowly embedded logic in Figure 1. The verification condition generator will always produce a VC for the required entailment of the computed pre-condition of the first instruction and the procedure's pre-condition, plus a VC for each annotated instruction, being the entailment between the annotation and the computed precondition. All VCs will have a formula of the form $\bigvee_i (\Pi_i \wedge (\Sigma_i * \Theta_i))$ as the antecedent and a goal formula as the conclusion.

5.3 Proof Search

The output of the verification condition generation phase is a collection of problems of the form $\Pi|\Sigma|\Theta \vdash G$. We define a proof search procedure by a set of rules shown in Figures 3, 4, 6 and 5. The key idea here is to use the I/O model of proof search as defined for intuitionistic linear logic by Cervesato, Hodas and Pfenning [5], and also the use of heuristic rules for unfolding the inductive list segment predicate.

As well as the main proof search judgement $\Pi|\Sigma|\Theta \vdash G$, we make use of several auxiliary judgements:

$\Pi|\Sigma|\Theta \vdash \Sigma_1\backslash\Sigma_2, \Theta'$	Heap assertion matching
$\Theta \vdash \Theta_1\backslash\Theta_2$	Resource matching
$\Pi \vdash \bot$	Contradiction spotting
$\Pi \vdash \Pi'$	Data entailment

The backslash notation used in these rules follows the I/O model of Cervesato et al, where in the judgement $\Theta \vdash \Theta_1\backslash\Theta_2$, the proof context Θ denotes the facts used as input and Θ_2 denotes the facts that are left over (the output) from proving Θ_1. A similar interpretation is used for the heap assertion matching judgement. We do not define the data entailment or contradiction spotting judgement explicitly here; we intend that these judgements satisfy the basic axioms of equalities and disequalities.

The rules in Figure 3 are the goal driven search rules. There is an individual rule for each possible kind of goal formula. The first two rules are matching rules that match a formula S against the context, altering the context to remove the heap and resource assertions that S requires, as dictated by the semantics of the assertion logic. We must search for a disjunct i that matches the current context. There may be multiple such i, and in this case the search may have to backtrack. When the goal is a formula S, then we check that the left-over heap is empty, in order to detect memory leaks.

$$\frac{\textsf{exists}\ i. \qquad \Pi|\Sigma|\Theta \vdash \Sigma_i\backslash\Sigma', \Theta' \qquad \Pi \vdash \Pi_i \qquad \Theta' \vdash \Theta_i\backslash\Theta'' \qquad \Pi|\Sigma'|\Theta'' \vdash G}{\Pi|\Sigma|\Theta \vdash \bigvee_i (\Pi_i \wedge (\Sigma_i * \Theta_i)) * G}$$

$$\frac{\textsf{exists}\ i. \qquad \Pi|\Sigma|\Theta \vdash \Sigma_i\backslash\textsf{emp}, \Theta' \qquad \Pi \vdash \Pi_i \qquad \Theta' \vdash \Theta_i\backslash\Theta''}{\Pi|\Sigma|\Theta \vdash \bigvee_i (\Pi_i \wedge (\Sigma_i * \Theta_i))}$$

$$\frac{\textsf{forall}\ i. \qquad \Pi, \Pi_i|\Sigma, \Sigma_i|\Theta, \Theta_i \vdash G}{\Pi|\Sigma|\Theta \vdash \bigvee_i (\Pi_i \wedge (\Sigma_i * \Theta_i)) -\!\!* G} \qquad \frac{\Pi, P|\Sigma|\Theta \vdash G}{\Pi|\Sigma|\Theta \vdash P \to G}$$

$$\frac{\Pi|\Sigma|\Theta \vdash G_1 \qquad \Pi|\Sigma|\Theta \vdash G_2}{\Pi|\Sigma|\Theta \vdash G_1 \wedge G_2} \qquad \frac{\Pi|\Sigma|\Theta \vdash G \qquad x \notin \textsf{fv}(\Pi) \cup \textsf{fv}(\Sigma)}{\Pi|\Sigma|\Theta \vdash \forall x.G}$$

$$\frac{\Pi|\Sigma|\Theta \vdash G[t/x]}{\Pi|\Sigma|\Theta \vdash \exists x.G}$$

Fig. 3. Goal Driven Search Rules

Heap Matching Rules:

$$\frac{}{\Pi|\Sigma|\Theta \vdash \mathsf{emp}\backslash\Sigma,\Theta} \qquad \frac{\Pi \vdash t_1 = t_1' \qquad \Pi \vdash t_2 = t_2'}{\Pi|\Sigma,[t_1 \overset{\mathsf{f}}{\mapsto} t_2]|\Theta \vdash [t_1' \overset{\mathsf{f}}{\mapsto} t_2']\backslash\Sigma,\Theta}$$

$$\frac{\Pi|\Sigma|\Theta \vdash \Sigma_1\backslash\Sigma',\Theta' \qquad \Pi|\Sigma'|\Theta' \vdash \Sigma_2\backslash\Sigma'',\Theta''}{\Pi|\Sigma|\Theta \vdash \Sigma_1 * \Sigma_2\backslash\Sigma'',\Theta''} \qquad \frac{\Pi \vdash t_1 = t_2}{\Pi|\Sigma|\Theta \vdash \mathsf{lseg}(\Theta_l,t_1,t_2)\backslash\Sigma,\Theta}$$

$$\frac{\Pi \vdash t_1 = t_1' \qquad \Theta \vdash \Theta_l\backslash\Theta' \qquad \Pi|\Sigma|\Theta' \vdash \mathsf{lseg}(\Theta_l,t_n,t_2)\backslash\Sigma',\Theta''}{\Pi|\Sigma,[t_1 \overset{\mathsf{n}}{\mapsto} t_n],[t_1 \overset{\mathsf{d}}{\mapsto} t_d]|\Theta \vdash \mathsf{lseg}(\Theta_l,t_1',t_2)\backslash\Sigma',\Theta''}$$

$$\frac{\Pi \vdash t_1' = t_1 \qquad \Pi|\Sigma|\Theta \vdash \mathsf{lseg}(\Theta_l,t_2,t_3)\backslash\Sigma',\Theta'}{\Pi|\Sigma,\mathsf{lseg}(\Theta_l,t_1,t_2)|\Theta \vdash \mathsf{lseg}(\Theta_l,t_1',t_3)\backslash\Sigma',\Theta'}$$

Resource Matching Rules:

$$\frac{}{\Theta,R^\rho \vdash R^\rho\backslash\Theta} \qquad \frac{}{\Theta \vdash \top\backslash\Theta} \qquad \frac{\Theta \vdash \Theta_1\backslash\Theta' \qquad \Theta' \vdash \Theta_2\backslash\Theta''}{\Theta \vdash \Theta_1 * \Theta_2\backslash\Theta''}$$

Fig. 4. Matching Rules

The matching rules make use of the heap and resource matching judgements defined in Figure 4. The heap matching judgements take a data, heap and resource context and attempt to match a list of heap assertions against them, returning the left over heap and resource contexts. The first three rules are straightforward: the empty heap assertion is always matchable, points-to relations are looked up in the context directly and pairs of heap assertions are split, threading the contexts through. For the list segment rules, there are three cases. Either the two pointers involved in the list are equal, in which case we are immediately done; or we have a single list cell in the context that matches the start pointer of the predicate we are trying to satisfy, and we have the required resources for an element of this list, so we can reduce the goal by one step; or we have a whole list segment in the context and we can reduce the goal accordingly. The resource matching rules are straightforward.

The final two sets of rules operate on the proof search context . The first set, shown in Figure 5, describe how information flows from the heap part of the context to the data part. If we know that two variables both have a points-to relation involving a field f, then we know that these locations must not be equal. Similarly, if we know that a variable does point to something, then it cannot be null. If any contradictions are found using these rules, then the proof search can terminate immediately for the current goal. This is provided for by the first rule in Figure 5.

The final set of rules performs heuristic unfolding of the inductive lseg predicate. These rules are shown in Figure 6. These rules take information from the data context and use it to unfold lseg predicates that occur in the heap context. The first rule is triggered when the proof search learns that there is a list

$$\frac{\Pi \vdash \bot}{\Pi|\Sigma|\Theta \vdash G}$$

$$\frac{\Sigma = [t_1 \overset{\mathsf{f}}{\mapsto} t], [t_2 \overset{\mathsf{f}}{\mapsto} t'], \Sigma' \qquad \Pi, t_1 \neq t_2|\Sigma|\Theta \vdash G}{\Pi|\Sigma|\Theta \vdash G}$$

$$\frac{\Sigma = [t \overset{\mathsf{f}}{\mapsto} t'], \Sigma' \qquad \Pi, t \neq \mathsf{null}|\Sigma|\Theta \vdash G}{\Pi|\Sigma|\Theta \vdash G}$$

Fig. 5. Contradiction Flushing

$$\frac{\Pi, t_1 = t_2|\Sigma|\Theta \vdash G \qquad \begin{array}{c}\Pi \vdash t_1 \neq \mathsf{null} \\ \Pi|\Sigma, [t_1 \overset{\mathsf{n}}{\mapsto} x], [t_1 \overset{\mathsf{d}}{\mapsto} y], \mathsf{lseg}(R, x, t_2)|\Theta, R \vdash G\end{array}}{\Pi|\Sigma, \mathsf{lseg}(R, t_1, t_2)|\Theta \vdash G}$$

$$\frac{\Pi \vdash t_1 = \mathsf{null} \qquad \Pi, t_2 = \mathsf{null}|\Sigma|\Theta \vdash G}{\Pi|\Sigma, \mathsf{lseg}(R, t_1, t_2)|\Theta \vdash G} \qquad \frac{\Pi \vdash t_1 = t_2 \qquad \Pi|\Sigma|\Theta \vdash G}{\Pi|\Sigma, \mathsf{lseg}(R, t_1, t_2)|\Theta \vdash G}$$

$$\frac{\Pi \vdash t_1 \neq t_2 \qquad \Pi|\Sigma, [t_1 \overset{\mathsf{n}}{\mapsto} x], [t_1 \overset{\mathsf{d}}{\mapsto} y], \mathsf{lseg}(R, x, t_2)|\Theta, R \vdash G}{\Pi|\Sigma, \mathsf{lseg}(R, t_1, t_2)|\Theta \vdash G}$$

Fig. 6. List Unfolding Rules

segment where the head pointer of the list is not equal to null. In this case, two proof search goals are produced, one for the case that the list segment is empty and one for when it isn't. The other rules are similar; taking information from the data context and using it to refine the heap context.

The proof search strategy that we employ works by first saturating the context by repeatedly applying the rules in Figures 5 and 6 to move information from the data context into the heap context and vice versa. This process terminates because there are a finite number of points-to relations and list segment predicates to generate rule applications, and when new predicates are introduced via list segment unfolding they either do not trigger any new inequalities or are over fresh variables about which nothing is yet known. Once the context is fully saturated, the proof search reduces the goal by using the goal-driven search rules and the process begins again.

Theorem 2. *The proof search procedure is sound and terminating.*

5.4 Integration with Linear Programming

A key feature of Hofmann and Jost's system for inference of resource bounds of functional programs [9] is the use of linear programming. In this section, we sketch how to extend the procedure of the previous section with linear constraint generation. Using this technique, as long as the resource bounds are linear, we can simply state our specifications in terms of the shapes of the data structures that the program manipulates and infer the necessary resource annotations.

For simplicity, we assume that we are dealing with resources that are positive rational numbers, so we can replace the resource contexts Θ in the proof search procedure of the previous section with linear expressions over the rationals. The resource matching judgement is altered to take and output linear expressions over rationals, while producing linear constraints over the variables mentioned in the resource expression, and we have the single rule:

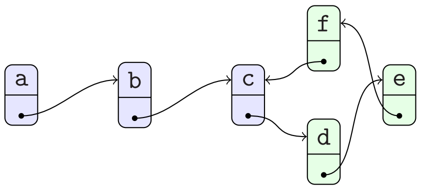
$$e_1 \vdash e_2 \backslash e_1 - e_2, e_2 \leq e_1$$

The proof search judgement is altered to generate a set of constraints over the variables mentioned in the resource expression e: $\Pi|\Sigma|e \vdash G\backslash\mathcal{C}$. The goal driven search rules are then modified to accumulate the generated constraints. The heap matching rules are similarly modified.

Given a collection of verification conditions and a successful proof search over them that has generated a set of linear constraints, we input these into a linear solver, along with the constraint that every variable is positive and an objective function that attempts to minimise variables appearing in the pre-condition.

6 Example: Frying Pan List Reversal

We demonstrate the use of the proof search procedure coupled with linear constraint generation to the standard imperative in-place list reversal algorithm on lists with cyclic tails (also known as "frying pan" lists). This example was used by Brotherston, Bornat and Calcagno [4] to demonstrate the use of cyclic proofs to prove program termination. Here we show how our amortised resource logic can be used to infer bounds on the complexity of this procedure.



The "handle" of the structure consists of the nodes `a`, `b`, `c` and the "pan" consists of the nodes `d`, `e` and `f`. When the in-place list-reversal procedure is run upon a structure of this shape, it will proceed up the handle, reversing it, around the pan, reversing it, and then back down the handle, restoring it to its original order. For the purposes of this example, we assume that it takes one element of resource to handle the reversal of one node. Following Brotherston, Bornat and Calcagno, we can specify a cyclic list in Separation Logic by the following formula, where v_0 points to the head of the list and v_1 points to the join between the handle and the pan.

$$\exists k.\mathsf{lseg}(x_1, v_0, v_1) * [v_1 \overset{\mathsf{next}}{\mapsto} k] * \mathsf{lseg}(x_2, k, v_1) * R^{x_3}$$

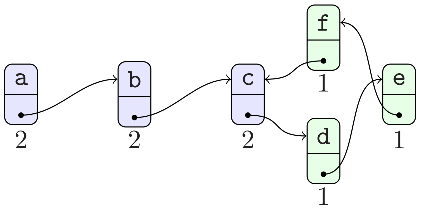
We have annotated the list segments involved with resource annotation variables x_1 and x_2 that we will instantiate using linear programming. The predicate R^{x_3} denotes any extra resource we may require. Similarly, we have annotated the required loop invariant (adapted from Brotherston et al):

$$\begin{aligned}
&(\exists k.\mathsf{lseg}(a_1, l_0, v_1) * \mathsf{lseg}(a_2, l_1, \mathtt{null}) * [v_1 \overset{\mathsf{next}}{\mapsto} k] * \mathsf{lseg}(a_3, k, v_1) * R^{a_4})\\
&\vee (\exists k.\mathsf{lseg}(b_1, k, \mathtt{null}) * [j \overset{\mathsf{next}}{\mapsto} k] * \mathsf{lseg}(b_2, l_0, v_1) * \mathsf{lseg}(b_3, l_1, j) * R^{b_4})\\
&\vee (\exists k.\mathsf{lseg}(c_1, l_0, \mathtt{null}) * \mathsf{lseg}(c_2, l_1, v_1) * [v_1 \overset{\mathsf{next}}{\mapsto} k] * \mathsf{lseg}(c_3, k, v_1) * R^{c_4})
\end{aligned}$$

Each disjunct of the loop invariant corresponds to a different phase of the procedure's progress. Brotherston et al note that it is possible to infer the shape part of this loop invariant using current Separation Logic tools. Here, we are adding the ability to infer resource bounds. Running our tool on this example produces the following instantiation of the variables:

Pre-condition	$x_1 = 2$	$x_2 = 1$	$x_3 = 2$	
Loop invariant, phase 1	$a_1 = 2$	$a_2 = 1$	$a_3 = 1$	$a_4 = 2$
Loop invariant, phase 2	$b_1 = 1$	$b_2 = 1$	$b_3 = 0$	$b_4 = 1$
Loop invariant, phase 3	$c_1 = 1$	$c_2 = 0$	$c_3 = 0$	$c_4 = 0$
Post-condition	$x'_1 = 0$	$x'_2 = 0$	$x'_3 = 0$	

Pictorially, the inference has associated the following amount of resource with each part of the input structure:



Each node of the handle has 2 associated elements of resource, to handle the two passes of the handle that the procedure takes, while the pan has one element of resource for each node. The inferred annotations for the loop invariant track how the resources on each node are consumed by the procedure, gradually all reducing to zero. Since we have added a consume instruction to be executed every time the procedure starts a loop, the resource inference process has also verified the termination of this procedure, and given us a bound on the number of times the loop will execute in terms of the shape of the input.

7 Conclusions

The main limitation of our proof search procedure is that it only supports the statement and inference of bounds that are linear in the size of lists that are mentioned in a procedure's precondition. This is a limitation shared with the work of Hofmann and Jost [9]. We note that this is not a limitation of the program logic that we have presented, only of the automated verification procedure that we have layered on top. We have demonstrated that the use of mixed shape and resource assertions can simplify the complexity of specifications that talk about resources, and this should extend to extensions of the proof search procedure, or to interactive systems based on this program logic. The resource aware program logic of Aspinall et al [2] also uses the same layering: a general program logic for resources (which is proved complete in their case) is used as a base for a specialised logic for reasoning about the output of the Hofmann-Jost system.

A possible direction for future work is to consider different assertion logics and their expressiveness in terms of the magnitude of resources they can express. We conjecture that the deep assertion logic we have presented here, extended with the lseg predicate can express resources linear in the size of the heap. It would be interesting to consider more expressive logics and evaluate them from the point of view of implicit computational complexity; the amount of resource that one can express in an assertion dictates the amount of resource that is available for the future execution of the program.

Other resource inference procedures that are able to deal with non-linear bounds include those of Chin et al [6,7], Albert et al [1] and Gulwani et al [8]. When dealing with heap-based data structures, all of these techniques use a method of attaching size information to assertions about data structures. As we demonstrated in Section 2, this can lead to additional unwanted complexity in specifications. However, all of these techniques deal with numerically bounded loops better than our current prototype automated procedure can, and we are currently investigating how to extend our approach to deal with non-linear and numerically-driven resource bounds.

References

1. Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: Costa: Design and implementation of a cost and termination analyzer for java bytecode. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2007. LNCS, vol. 5382, pp. 113–132. Springer, Heidelberg (2008)
2. Aspinall, D., Beringer, L., Hofmann, M., Loidl, H.-W., Momigliano, A.: A program logic for resources. Theor. Comput. Sci. 389(3), 411–445 (2007)
3. Berdine, J., Calcagno, C., O'Hearn, P.W.: Symbolic execution with separation logic. In: Yi, K. (ed.) APLAS 2005. LNCS, vol. 3780, pp. 52–68. Springer, Heidelberg (2005)
4. Brotherston, J., Bornat, R., Calcagno, C.: Cyclic proofs of program termination in separation logic. In: Necula, G.C., Wadler, P. (eds.) POPL, pp. 101–112. ACM, New York (2008)
5. Cervesato, I., Hodas, J.S., Pfenning, F.: Efficient resource management for linear logic proof search. Theor. Comput. Sci. 232(1-2), 133–163 (2000)
6. Chin, W.-N., Nguyen, H.H., Popeea, C., Qin, S.: Analysing memory resource bounds for low-level programs. In: Jones, R., Blackburn, S.M. (eds.) ISMM, pp. 151–160. ACM, New York (2008)

7. Chin, W.-N., Nguyen, H.H., Qin, S., Rinard, M.C.: Memory usage verification for oo programs. In: Hankin, C., Siveroni, I. (eds.) SAS 2005. LNCS, vol. 3672, pp. 70–86. Springer, Heidelberg (2005)
8. Gulwani, S., Mehra, K.K., Chilimbi, T.M.: Speed: precise and efficient static estimation of program computational complexity. In: Shao, Z., Pierce, B.C. (eds.) POPL, pp. 127–139. ACM, New York (2009)
9. Hofmann, M., Jost, S.: Static prediction of heap space usage for first-order functional programs. In: POPL, pp. 185–197 (2003)
10. Hofmann, M., Jost, S.: Type-based amortised heap-space analysis. In: Sestoft, P. (ed.) ESOP 2006. LNCS, vol. 3924, pp. 22–37. Springer, Heidelberg (2006)
11. Hofmann, M., Rodriguez, D.: Efficient type-checking for amortised heap-space analysis. In: Grädel, E., Kahle, R. (eds.) CSL 2009. LNCS, vol. 5771, pp. 317–331. Springer, Heidelberg (2009)
12. Ishtiaq, S., O'Hearn, P.W.: Bi as an assertion language for mutable data structures. In: Proceedings of the 28th Symposium on Principles of Programming Languages, January 2001, pp. 14–26 (2001)
13. Restall, G.: An Introduction to Substructural Logics. Routledge (2000)
14. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: Proceedings of 17th Annual IEEE Symposium on Logic in Computer Science (2002)
15. Tarjan, R.E.: Amortized computational complexity. SIAM Journal on Algebraic and Discrete Methods 6(2), 306–318 (1985)