Abstracting Allocation

The New `new` Thing

Nick Benton

Microsoft Research
nick@microsoft.com

Abstract. We introduce a Floyd-Hoare-style framework for specification and verification of machine code programs, based on relational parametricity (rather than unary predicates) and using both step-indexing and a novel form of separation structure. This yields compositional, descriptive and extensional reasoning principles for many features of low-level sequential computation: independence, ownership transfer, unstructured control flow, first-class code pointers and address arithmetic. We demonstrate how to specify and verify the implementation of a simple memory manager and, independently, its clients in this style. The work has been fully machine-checked within the Coq proof assistant.

1 Introduction

Most logics and semantics for languages with dynamic allocation treat the allocator, and a notion of what has been allocated at a particular time, as part of their basic structure. For example, marked-store models, and those based on functor categories or FM-cpos, have special treatment of locations baked in, as do operational semantics using partial stores, where programs 'go wrong' when accessing unallocated locations. Even type systems and logics for low-level programs, such as TAL [14], hardwire allocation as a primitive.

For high-level languages such as ML in which allocation is observable but largely abstract (no address arithmetic, order comparison or explicit deallocation), building 'well-behaved' allocation into a model seems reasonable. But even then, we typically obtain base models that are far from fully abstract and have to use a second level of non-trivial relational reasoning to validate even the simplest facts about encapsulation.

For low-level languages, hardwiring allocation is less attractive. Firstly, and most importantly, we want to reason about the low-level code that actually implements the storage manager. Secondly, in languages with address arithmetic, such as the while-language with pointers used in separation logic, one is led to treat allocation as a non-deterministic primitive, which is semantically problematic, especially if one tries to reason about refinement, equivalence or imprecise predicates [23, 13]. Finally, it just doesn't correspond to the fact that 'machine code programs don't go wrong'. The fault-avoiding semantics of separation logic, for example, is prescriptive, rather than descriptive: one can only prove anything

about programs that never step outside their designated footprint, even if they do so in a non-observable way.[1]

We instead start with a completely straightforward operational semantics for an idealized assembly language. There is a single datatype, the natural numbers, though different instructions treat elements of that type as code pointers, heap addresses, integers, etc. The heap is simply a total function from naturals to naturals and the code heap is a total function from naturals to instructions. Computed branches and address arithmetic are perfectly allowable. There is no built-in notion of allocation and no notion of stuckness or 'going wrong': the only observable behaviours are termination and divergence.

Over this simple and permissive model, we aim to develop semantic (defined in terms of observable behaviour) safety properties, and ultimately a program logic, that are rich enough to capture the equational semantics of high-level types as properties of compiled code and also to express and verify the behavioural contracts of the runtime systems, including memory managers, upon which compiled code depends.

Our approach is based on four technical ideas. Firstly, following the interpretation of types as PERs, we work with quantified binary relations rather than the unary predicates more usual in program logics. Program properties are expressed in terms of contextual equivalence, rather than avoidance of some artificial stuck states. Secondly, we use a perping operation, taking relations on states to orthogonal relations on code addresses, to reason about first-class code pointers. Thirdly, we reason modularly about the heap in a style similar to separation logic, but using an explicit notion of the portion of the heap on which a relation depends. Finally, we reason modularly about mutually-recursive program fragments in an assume/guarantee style, using a step-indexing technique similar to that of Appel et al [5, 6, 3] to establish soundness.

In this paper, we concentrate on the specification and verification of an extremely basic memory allocation module, and an example client. Although the code itself may be simple, the specifications and proofs are rather less so, and provide a non-trivial test case for our general framework, as well as constituting a fresh approach to freshness.

Managing the mind-numbing complexity and detail of specifications and proofs for machine code programs, not to mention keeping oneself honest in the face of changing definitions, seems to call for automated assistance. All the definitions and results presented here have been formalized and checked using the Coq proof assistant.

---

[1] For example, `skip` and `[10] := [10]` are, under mild assumptions, observationally equivalent, yet do not satisfy exactly the same set of triples. One might reasonably claim that machine code programs *do* go wrong – by segfaulting – and that this justifies faulting semantics and the use of partial stores. But stuck states in most operational semantics, even for low-level code, do not correspond exactly to the places in which segfaults might really occur, and we'd rather not assume or model anything about an operating system for the moment anyway.

2 The Machine

Our idealized sequential machine model looks like:

$$\begin{array}{rrcl} s \in & \mathbb{S} & \stackrel{def}{=} & \mathbb{N} \to \mathbb{N} \quad \text{states} \\ l, m, n, b \in & \mathbb{N} & & \text{naturals in different roles} \\ p \in Programs & & \stackrel{def}{=} & \mathbb{N} \to Instr \quad \text{programs} \\ \langle p|s|l\rangle \in & Configs & \stackrel{def}{=} & Programs \times \mathbb{S} \times \mathbb{N} \end{array}$$

The instruction set, $Instr$, includes `halt`, direct and indirect stores and loads, some (total) arithmetic and logical operations, and conditional and unconditional branches.[2] The semantics is given by an obvious deterministic transition relation $\langle p|s|l\rangle \to \langle p|s'|l'\rangle$ between configurations. We write $\langle p|s|l\rangle \Downarrow$ if there exists n,l',s' such that $\langle p|s|l\rangle \to^n \langle p|s'|l'\rangle$ with $p(l') = \texttt{halt}$, and $\langle p|s|l\rangle \Uparrow$ if $\langle p|s|l\rangle \to^\omega$.

The major idealizations compared with a real machine are that we have arbitrary-sized natural numbers as a primitive type, rather than fixed-length words, and that we have separated code and data memory (ruling out self-modifying code and dynamic linking for the moment). Note also that we do not even have any registers.

3 Relations, Supports and Indexing

We work with binary relations on the naturals, $\mathbb{N}$, and on the set of states, $\mathbb{S}$, but need some extra structure. Firstly, the reason for using relations is to express specifications in terms of behavioural equivalences between configurations:

$$\langle p|s|l\rangle \Downarrow \iff \langle p'|s'|l'\rangle \Downarrow$$

and the relations on states and naturals we use to establish such equivalences will generally be functions of the programs p and p' (because they will refer to the sets of code pointers that, in p and p', have particular behaviours). Secondly, to reason modularly about mutually recursive program fragments, we need to restrict attention to relations satisfying an admissibility property, which we capture by step-indexing: relations are parameterized by, and antimonotonic in, the number of computation steps available for distinguishing values (showing they're not in the relation). Formally, an *indexed nat relation*, is a function

$$r : Programs \times Programs \to \mathbb{N} \to \mathbb{P}(\mathbb{N} \times \mathbb{N})$$

such that $(r\,(p, p')\,k) \subseteq (r\,(p, p')\,j)$ whenever $j < k$.

For state relations, we also care about what *parts* of the state our relations depend upon. Separation logic does this implicitly, and sometimes indeterminately, via the existential quantification over splittings of the heap in the definition of

[2] The Coq formalization currently uses a shallow embedding of the machine semantics, so the precise instruction set is somewhat fluid.

separating conjunction. Instead, we work with an explicit notion, introduced in [9], of the support of a relation. One might expect this to be a set of locations, but the support is often itself a function of the state (think, for example, of the equivalence of linked lists). However, not all functions $\mathbb{S} \to \mathbb{P}(\mathbb{N})$ make sense as supports: the function itself should not depend on the contents of locations which are not in its result.[3] Formally, we define an *accessibility map* to be a function $A : \mathbb{S} \to \mathbb{P}(\mathbb{N})$ such that

$$\forall s, s'.\ s \sim_{A(s)} s' \implies A(s') = A(s)$$

where, for $L \subseteq \mathbb{N}$ and $s, s' \in \mathbb{S}$, we write $s \sim_L s'$ to mean $\forall l \in L. s(l) = s'(l)$.

Accessibility maps are ordered by $A \subseteq A' \iff \forall s. A(s) \subseteq A'(s)$. Constant functions are accessibility maps, as is the union $A \cup A'$ of two accessibility maps, where $(A \cup A')(s) = A(s) \cup A'(s)$. Despite the name, accessibility maps are about relevance and *not* reachability. Indeed, reachability makes little sense in a model without an inherent notion of pointer.

A *supported indexed state relation* R is a triple $(|R|, A_R, A'_R)$ where

$$|R| : Programs \times Programs \to \mathbb{N} \to \mathbb{P}(\mathbb{S} \times \mathbb{S})$$

satisfies $(|R|\,(p,p')\,k) \subseteq (|R|\,(p,p')\,j)$ for all $(j < k)$, A_R and A'_R are accessibility maps and for all $s_1 \sim_{A_R(s_1)} s_2$ and $s'_1 \sim_{A'_R(s'_1)} s'_2$,

$$(s_1, s'_1) \in |R|\,(p,p')\,k \implies (s_2, s'_2) \in |R|\,(p,p')\,k.$$

We often elide the $|\cdot|$. The constantly total and empty state relations are each supported by any accessibility maps. The separating product of supported indexed relations is given by

$$\begin{aligned} R_1 \otimes R_2 &= (|R_1 \otimes R_2|,\ A_{R_1} \cup A_{R_2},\ A'_{R_1} \cup A'_{R_2}) \quad \text{where} \\ |R_1 \otimes R_2|\,(p,p')\,k &= (|R_1|\,(p,p')\,k) \ \cap\ (|R_2|\,(p,p')\,k) \ \cap \\ &\qquad \{(s,s') \mid A_{R_1}(s) \cap A_{R_2}(s) = \emptyset \wedge A'_{R_1}(s') \cap A'_{R_2}(s') = \emptyset\} \end{aligned}$$

This is associative and commutative with the constantly total relation with empty support, $T_\emptyset$, as unit. The partial order $R_1 \preceq R_2$ on state relations is defined as

$$\forall (s,s') \in |R_1|.\ ((s,s') \in |R_2|) \wedge (A_{R_2}(s) \subseteq A_{R_1}(s)) \wedge (A'_{R_2}(s') \subseteq A'_{R_1}(s'))$$

which has the property that if $R_1 \preceq R_2$ then for any R_I, $|R_1 \otimes R_I| \subseteq |R_2 \otimes R_I|$.

If R is a (supported) indexed state relation, its *perp*, $R^\top$, is an indexed nat relation defined by:

$$\begin{aligned} R^\top\,(p,p')\,k = \{(l,l') \mid \forall j < k. \forall (s,s') \in (R\,(p,p')\,j).\ & \\ (\langle p,s,l\rangle \Downarrow_j \implies \langle p',s',l'\rangle \Downarrow) \ \wedge & \\ (\langle p',s',l'\rangle \Downarrow_j \implies \langle p,s,l\rangle \Downarrow)\} & \end{aligned}$$

[3] In other words, the function should support itself.

where $\Downarrow_j$ means 'converges in fewer than j steps'. Roughly speaking, $R^\top$ relates two labels if jumping to those labels gives equivalent termination behaviour whenever the two initial states are related by R; the indexing lets us deal with cotermination as the limit (intersection) of a sequence of k-step approximants.

If $q \subseteq \mathbb{N} \times \mathbb{N}$, write $\overline{q}$ for the indexed nat relation $\lambda(p, p').\lambda k.q$, and similarly for indexed state relations. If $L \subseteq \mathbb{N}$, A_L is the accessibility map $\lambda s.L$. We write T_L for the supported indexed state relation $(\overline{\mathbb{S} \times \mathbb{S}}, A_L, A_L)$ and write sets of integers $\{m, m+1, \ldots, n\}$ just as mn. If r is an indexed nat relation and $n, n' \in \mathbb{N}$, write $(n, n' \Mapsto r)$ for the supported indexed state relation

$$\lambda(p, p').\lambda k.\, (\{(s, s') \mid (s(n), s'(n')) \in r\,(p, p')\,k\},\ \lambda s.\{n\},\ \lambda s.\{n'\})$$

relating pairs of states that have values related by r stored in locations n and n'. We write the common diagonal case $(n, n \Mapsto r)$ as $(n \mapsto r)$. For $\mathtt{M}$ a program fragment (partial function from naturals to instructions) define

$$\models \mathtt{M} \rhd l : R^\top \ \stackrel{def}{=}\ \forall p, p' \supseteq \mathtt{M}.\forall k.\, (l, l) \in (R^\top\,(p, p')\,k)$$

where the quantification is over all (total) programs extending $\mathtt{M}$. We are only considering a single $\mathtt{M}$, so our basic judgement is that a label is related to itself. More generally, define $\boldsymbol{l_i} : \boldsymbol{R_i^\top} \models \mathtt{M} \rhd l : R^\top$ to mean

$$\forall p, p' \supseteq \mathtt{M}.\forall k.\, (\forall i.(l_i, l_i) \in (R_i^\top\,(p, p')\,k)) \implies ((l, l) \in (R^\top\,(p, p')\,k + 1))$$

i.e. for any programs extending $\mathtt{M}$ and for any k, if the hypotheses on the labels $\boldsymbol{l_i}$ are satisfied to index k, then the conclusion about l holds to index $k+1$.

4 Specification of Allocation

The machine model is very concrete and low-level, so we have to be explicit about details of calling conventions in our specifications. We arbitrarily designate locations 0 - 9 as register-like and, for calling the allocator, will use 0 - 4 for passing arguments, returning results and as workspace. An allocator module is just a code fragment, $\mathtt{M}_a$, which we will specify and verify in just the same way as its clients. There are entry points for initialization, allocation and deallocation.

The code at label `init` sets up the internal data structures of the allocator. It takes a return address in location 0, to which it will jump once initialization is complete. The code at `alloc` expects a return address in location 0 and the size of the requested block in location 1. The address of the new block will be returned in location 0. The code at `dealloc` takes a return address in 0, the size of the block to be freed in 1 and the address of the block to be freed in 2.

After initialization, the allocator owns some storage in which it maintains its internal state, and from which it hands out (transfers ownership of) chunks to clients. The allocator depends upon clients not interfering with, and behaving independently of, both the location and contents of its private state. In particular, clients should be insensitive to the addresses and the initial contents

of chunks returned by calls to `alloc`. In return, the allocator promises not to change or depend upon the contents of store owned by the client. All of these independence, non-interference and ownership conditions can be expressed using supported relations. Furthermore, this can be done extensionally, rather than in terms of which locations are read, written or reachable.

There will be some supported indexed state relation R_a for the private invariant of the allocator. The supports of R_a express what store is owned by the allocator; this has to be a function of the state (rather than just a set of locations), because what is owned varies as blocks are handed out and returned. One should think of $|R_a|$ as expressing what configurations of the store owned by the allocator are valid, and which of those configurations are equivalent.

When `init` is called, the allocator takes ownership of some (infinite) part of the store, which we only specify to be disjoint from locations 0-9. On return, locations 0-4 may have been changed, 5-9 will be preserved, and none of 1-9 will observably have been read. So two calls to `init` yield equivalent behaviour when the return addresses passed in location 0 yield equivalent behaviour whenever the states they're started in are as related as `init` guarantees to make them. How related is that? Well, there are no guarantees on 0-4, we'll preserve any relation involving 5-9 *and* we'll establish R_a on a disjoint portion of the heap. Thus, the specification for initialization is that for *any* nat relations r_5,r_6,...,r_9,

$$\models \mathtt{M}_a \rhd \mathtt{init} : \left(\left(0 \mapsto (R_a \otimes T_{04} \otimes \bigotimes_{i=5}^{9} (i \mapsto r_i))^\top\right) \otimes \bigotimes_{i=5}^{9} (i \mapsto r_i)\right)^\top \tag{1}$$

When `alloc` is called, the client (i.e. the rest of the program) will already have ownership of some disjoint part of the heap and its own invariant thereon, R_c. Calls to `alloc` behave equivalently provided they are passed return continuations that behave the same whenever their start states are related by R_c, R_a *and* in each state location 0 points to a block of memory of the appropriate size and disjoint from R_c and R_a. More formally, the specification for allocation is that for *any* n and for *any* R_c

$$\models \mathtt{M}_a \rhd \mathtt{alloc} : (R_{aparms}(n, R_a, R_a) \otimes T_{24} \otimes R_c \otimes R_a)^\top \tag{2}$$

where

$$R_{aparms}(n, R_a, R_c) = \left(\left(0 \mapsto (R_{aret}(n) \otimes T_{14} \otimes R_c \otimes R_a)^\top\right) \otimes \left(1 \mapsto \overline{\{(n, n)\}}\right)\right)$$

$$\text{and} \quad R_{aret}(n) = \left(\overline{\{(s, s') \mid s(0) > 9 \wedge s'(0) > 9\}},\ A_{aret}(n),\ A_{aret}(n)\right)$$

$$A_{aret}(n) = \lambda s.\{0\} \cup \{s(0), \ldots, s(0) + n - 1\}$$

R_{aret} guarantees that the allocated block will be disjoint from the pseudo-registers, but nothing more; this captures the requirement for clients to behave equivalently whatever block they're returned and whatever its initial contents. A_{aret} includes both location 0, in which the start of the allocated block is returned, and the block itself; the fact that this is tensored with R_a and R_c in the

precondition for the return address allows the client to assume that the block is disjoint from both the (updated) internal datastructures of the allocator and the store previously owned by the client. We can avoid saying anything explicit about preservation of locations 5 to 9 here because they can be incorporated into R_c.

When `dealloc` is called, R_a will hold and the client will have an invariant R_c that it expects to be preserved *and* a disjoint block of store to be returned. The client expresses that it no longer needs the returned block by promising that the return address will behave equivalently provided that just R_c and R_a hold. Formally, for *any* R_c and n,

$$\models \mathtt{M}_a \triangleright \mathtt{dealloc} : \left(\left(0 \mapsto (T_{04} \otimes R_c \otimes R_a)^\top\right) \otimes \left(1 \mapsto \overline{\{(n, n)\}}\right) \otimes T_{34} \otimes R_{fb}(n)\right)^\top \tag{3}$$

where

$$R_{fb}(n) = \left(\overline{\{(s, s') \mid s(2) > 9 \wedge s'(2) > 9\}},\ A_{fb}(n),\ A_{fb}(n)\right)$$
$$A_{fb}(n) = \lambda s.\, \{2\} \cup \{s(2), \ldots, s(2) + n - 1\}$$

Writing the relations on the RHSs of (1), (2) and (3) as $r_{in}(R_a, r_5, \ldots, r_9)$, $r_{al}(R_a, n, R_c)$ and $r_{de}(R_a, n, R_c)$, respectively, the whole specification of an allocator module is therefore

$$\begin{aligned} \exists R_a.\, \models \mathtt{M}_a \triangleright (&\mathtt{init} : \forall r_5, \ldots, r_9.\, r_{in}(R_a, r_5, \ldots, r_9)) \\ &\wedge (\mathtt{alloc} : \forall n. \forall R_c.\, r_{al}(R_a, n, R_c)) \\ &\wedge (\mathtt{dealloc} : \forall n. \forall R_c.\, r_{de}(R_a, n, R_c)) \end{aligned} \tag{4}$$

Note that the existentially-quantified R_a is scoped across the whole module interface: the *same* invariant has to be maintained by the cooperating implementations of all three operations, even though it is abstract from the point of view of clients.

Checking that all the things we have assumed to be accessibility maps and supported relations really *are* is straightforward from the definitions.

5 Verification of Allocation

We now consider verifying the simplest useful allocation module, $\mathtt{M}_a$, shown in Figure 1. Location 10 points to the base of an infinite contiguous chunk of free memory. The allocator owns location 10 and all the locations whose addresses are greater than or equal to the current contents of location 10. Initialization sets the contents of 10 to 11, claiming everything above 10 to be unallocated, and returns. Allocation saves the return address in location 2, copies a pointer to the next currently free location (the start of the chunk to be returned) into 0, bumps location 10 up by the number of locations to be allocated and returns to the saved address. Deallocation is simply a no-op: in this trivial implementation, freed store is actually never reused, though the specification requires that well-behaved clients never rely on that fact.

```
    init : [10] ← 11              // set up free ptr
init + 1 : jmp [0]                // return

   alloc : [2] ← [0]              // save return address
alloc + 1 : [0] ← [10]            // return value = old free ptr
alloc + 2 : [10] ← [10] + [1]     // bump free ptr by n
alloc + 3 : jmp [2]               // return to saved address

 dealloc : jmp [0]                // return (!)
```

Fig. 1. The Simplest Allocator Module, $\mathtt{M}_a$

Theorem 1. *The allocator code in Figure 1 satisfies the specification, (4), of the previous section.* □

For this implementation, the relation, R_a, witnessing the existential in the specification is just

$$R_a \stackrel{def}{=} \left(\overline{\{(s,s') \mid (s(10) > 10) \wedge (s'(10) > 10)\}},\ A_a,\ A_a\right)$$

where A_a is $\lambda s.\{10\} \cup \{m | m \geq s(10)\}$. The only invariant this allocator needs is that the next free location pointer is strictly greater than 10, so memory handed out never overlaps either the pseudo registers 0-9 or the allocator's sole bit of interesting private state, location 10 itself. A_a says what storage is owned by the allocator.

The proof of Theorem 1 is essentially forward relational Hoare-style reasoning, using assumed separation conditions to justify the framing of invariants. In particular, the prerelation for `alloc` lets us assume that the support A_c of R_c is disjoint from both $\{0, \ldots, 4\}$ and A_a in each of the related states (s, s') in which we make the initial calls. Since the code only writes to locations coming from those latter two accessibility maps, we know that they are still related by R_c, even though we do not know anything more about what R_c is. More generally, we have the following reasoning principle:

Lemma 1 (Independent Updates). *For any p, p', k, n, n', v, v', r_{old}, r_{new}, R_{inv}, s, s',*

$$(v, v') \in (r_{new}\,(p, p')\,k) \quad \textit{and} \quad (s, s') \ \in\ ((n, n' \Mapsto r_{old}) \otimes R_{inv})\ (p, p')\,k$$

implies $(s[n \mapsto v],\ s'[n' \mapsto v']) \ \in\ ((n, n' \Mapsto r_{new}) \otimes R_{inv})\ (p, p')\,k.$

Proof. By assumption, the prestates s and s' are related by $|R_{inv}|$, and the supports $A_{inv}(s)$ and $A'_{inv}(s')$ do not include n and n', respectively. Hence, by the self-supporting property of accessibility maps, $A_{inv}(s[n \mapsto v]) = A_{inv}(s)$, and similarly for A'_{inv}. Thus $s[n \mapsto v] \sim_{A_{inv}(s)} s$ and $s'[n' \mapsto v'] \sim_{A'_{inv}(s')} s'$, so the updated states are still related by $|R_{inv}|$ by the saturation property of supported relations, and the supports of the tensored relations in the conclusion are still disjoint. □

We also use the following for reasoning about individual transitions:

Lemma 2 (Single Steps). *For all p, p', k, R_{pre}, l_{pre}, l'_{pre}, if for all $j < k$ and for all $(s_{pre}, s'_{pre}) \in (R_{pre}\,(p,p')\,j)$*

$$\langle p|s_{pre}|l_{pre}\rangle \to \langle p|s_{post}|l_{post}\rangle \quad \textit{and} \quad \langle p'|s'_{pre}|l'_{pre}\rangle \to \langle p'|s'_{post}|l'_{post}\rangle$$

implies there exists an R_{post} such that

$$(s_{post}, s'_{post}) \in (R_{post}\,(p,p')\,j) \quad \textit{and} \quad (l_{post}, l'_{post}) \in (R_{post}\,(p,p')\,j)^\top$$

then $(l_{pre}, l'_{pre}) \in (R_{pre}\,(p,p')\,k)^\top$. □

For straight-line code that just manipulates individual values in fixed locations, the lemmas above, together with simple rules of consequence involving $\preceq$, are basically all one needs. The pattern is that one applies the single step lemma to a goal of the form $(l_{pre}, l'_{pre}) \in (R_{pre}\,(p,p')\,k)^\top$, generating a subgoal of the form 'transition implies exists R_{post} such that post states are related and (l_{post}, l'_{post}) are in $R_{post}^\top$'. One then examines the instructions at l_{pre} and l'_{pre}, which defines the possible post states and values of l_{post} and l'_{post}. One then instantiates R_{post}, yielding one subgoal that the post states (now expressed as functions of the prestates) are related and one about (l_{post}, l'_{post}). In the case that the instruction was an update, one then uses the independent update lemma to discharge the first subgoal, leaving the goal of proving a perp about (l_{post}, l'_{post}), for which the pattern repeats. Along the way, one uses consequence to put relations into the right form for applying lemmas and assumptions.

In interesting cases of ownership transfer, the consequence judgements one has to prove require splitting and recombining relations that have non-trivial supports. This typically involves introducing new existentially quantified logical variables. For example, after the instruction at `alloc+1` we split the state-dependency of the support of R_a by deducing that there exist $b, b' \in \mathbb{N}$, both greater than 10, such that the two intermediate states are related by

$$\left(0 \mapsto \overline{\{b, b'\}}\right) \otimes \left(10 \mapsto \overline{\{b, b'\}}\right) \otimes (\overline{\mathbb{S} \times \mathbb{S}}, A_{old}, A'_{old}) \otimes (\overline{\mathbb{S} \times \mathbb{S}}, A_{new}, A'_{new}) \otimes \cdots$$

where $A_{old}(s) = \{m \mid m \geq b + n\}$, $A'_{old}(s') = \{m \mid m \geq b' + n\}$, $A_{new}(s) = \{m \mid b \leq m < b + n\}$ and $A'_{new}(s') = \{m \mid b' \leq m < b' + n\}$. The first and fourth of these then combine to imply $R_{aret}(n)$, so after the update at `alloc+2` the states are related by

$$R_{aret}(n) \otimes \left(10 \mapsto \overline{\{b + n, b' + n\}}\right) \otimes (\overline{\mathbb{S} \times \mathbb{S}}, A_{old}, A'_{old}) \otimes \cdots$$

the second and third of which then recombine to imply R_a again, eliminating b and b' and establishing the precondition for the return jump at `alloc+3`.

6 Specification and Verification of a Client

We now specify and verify a client of the allocator, using the specification of Section 4. Amongst other things, this shows how we deal modularly with linking,

recursion and adaptation. The client specification is intended as a very simple example of how one might express the semantics of types in a high-level language as relations in our low-level logic, expressing the behavioural contracts of code compiled from phrases of those types. In this case, the high-level language is an (imaginary) pure first-order one that, for the purposes of the example, we compile using heap-allocated activation records.

We concentrate on the meaning of the type $\texttt{nat} \to \texttt{nat}$. From the point of view of the high-level language, the semantics of that type is something like the predomain $\mathbb{N} \to \mathbb{N}_\perp$, or relationally, a PER on some universal domain relating functions that take equal natural number arguments to equal results of type 'natural-or-divergence'. There are many mappings from such a high-level semantics to the low-level, reflecting many different correct compilation schemes. We'll assume values of type `nat` are compiled as the obviously corresponding machine values, so the interpretation $[\![\texttt{nat}]\!]$ is the constantly diagonal relation $\{(n, n) \mid n \in \mathbb{N}\}$.

For functions we choose to pass arguments and return results in location 5, to pass return addresses in 6, to use 7 to point to the activation record, and 0-4 as workspace.[4] Since functions call the allocator, they will also explicitly assume and preserve R_a, as well as some unknown frame R_c for the invariants of the rest of the program. The allocator's invariant is abstract from the point of view of its clients, but they all have to be using the same one, so we parameterize client specs by the allocator's invariant. This leads us to define $[\![\texttt{nat} \to \texttt{nat}]\!](R_a)$ as the following indexed nat relation:

$$\forall R_c. \forall r_7. \left(\begin{array}{l} T_{04} \otimes (5 \mapsto [\![\texttt{nat}]\!]) \otimes (7 \mapsto r_7) \otimes R_c \otimes R_a \otimes \\ \left(6 \mapsto (T_{04} \otimes (5 \mapsto [\![\texttt{nat}]\!]) \otimes R_c \otimes R_a \otimes T_6 \otimes (7 \mapsto r_7))^\top\right) \end{array} \right)^\top$$

which one can see as the usual 'equal arguments to equal results' logical relation, augmented with extra invariants that ensure that the code respects the calling convention, uses the allocator properly and doesn't observably read or write any storage that it shouldn't. Although the high-level type is simple, the corresponding low-level specification is certainly non-trivial.

As a concrete example of something that should meet this spec, we (predictably) take an implementation, $\texttt{M}_f$, of the factorial function, shown in Figure 2. The factorial code is mildly optimized: it calls the allocator to allocate its activation record, but avoids the allocation if no recursive call is needed. After a recursive call, the activation record is deallocated using a tail call: `dealloc` returns directly to the caller of `fact`. The ability to reason about optimized code is a benefit of our extensional approach compared with more type-like methods which assume code of a certain shape.

The result we want about the factorial is that it satisfies the specification corresponding to its type whenever it is linked with code satisfying the specification of an allocator. Opening the existential package, this means that for *any*

[4] This differs from the allocator's calling convention because we need to call the allocator to get some space *before* we can save the parameters to a function call.

```
fact   :  brz [5] (fact+17)          // jump to fact+17 if [5]=0
fact+ 1:  [1] <- 3                   // size of activation record
fact+ 2:  [0] <- (fact+4)            // return address for alloc
fact+ 3:  jmp alloc                  // allocate activation record
fact+ 4:  [[0]] <- [5]               // copy arg to frame[0]
fact+ 5:  [[0]+1] <- [6]             // copy ret addr to frame[1]
fact+ 6:  [[0]+2] <- [7]             // copy old frame ptr to frame[2]
fact+ 7:  [7] <- [0]                 // new frame ptr in 7
fact+ 8:  [5] <- ([5]-1)             // decrement arg
fact+ 9:  [6] <- (fact+11)           // ret addr for recursive call
fact+10:  jmp fact                   // make recursive call
fact+11:  [5] <- ([5]*[[7]])         // return value = (fact (n-1))*n
fact+12:  [0] <- [[7]+1]             // ret addr for dealloc tail call
fact+13:  [2] <- [7]                 // arg for call to dealloc
fact+14:  [7] <- [[7]+2]             // restore old frame ptr
fact+15:  [1] <- 3                   // size of block for dealloc
fact+16:  jmp dealloc                // dealloc frame and tail return
fact+17:  [5] <- 1                   // return value = 1
fact+18:  jmp [6]                    // return
```

Fig. 2. Code for the Factorial Function, $\mathtt{M}_f$

$\mathtt{M}_a$ satisfying (4), there's an R_a such that

$$\models (\mathtt{M}_a \cup \mathtt{M}_f) \rhd (\mathtt{fact} : [\![\mathtt{nat} \to \mathtt{nat}]\!](R_a)) \wedge (\mathtt{alloc} : \forall n.\forall R_c.\, r_{al}(R_a, n, R_c)) \wedge \ldots$$

which is a consequence of the following, quite independent of any particular $\mathtt{M}_a$:

Theorem 2. *For any R_a,*

$$\begin{array}{l}\mathtt{init} : \forall r_5, \ldots, r_9.\, r_{in}(R_a, r_5, \ldots, r_9),\\ \mathtt{alloc} : \forall n.\forall R_c.\, r_{al}(R_a, n, R_c),\\ \mathtt{dealloc} : \forall n.\forall R_c.\, r_{de}(R_a, n, R_c)\end{array} \qquad \models \mathtt{M}_f \rhd \mathtt{fact} : [\![\mathtt{nat} \to \mathtt{nat}]\!](R_a)$$

□

This is another Hoare-style derivation, mostly similar to that of Theorem 1. Proving the calls, including the recursive one, requires the universal quantifications over R_c, n and r_7, occurring in the specifications of `alloc`, `dealloc` and `fact`, to be appropriately instantiated ('adapted'). For example, the instantiation of R_c for the recursive call at label `fact+10` is

$$\begin{array}{l}R'_c \otimes (b, b' \Rrightarrow [\![\mathtt{nat}]\!])\\ \otimes \big(b+1, b'+1 \Rrightarrow ((5 \mapsto [\![\mathtt{nat}]\!]) \otimes T_{04} \otimes R'_c \otimes R_a \otimes T_6 \otimes (7 \mapsto r'_7))^{\top}\big)\\ \otimes (b+2, b'+2 \Rrightarrow r'_7)\end{array}$$

where R'_c and r'_7 were the instantiations of the outer call, and b and b' are logical variables standing for the addresses returned by the previous related calls to the allocator at `fact+3`. This clearly expresses how the recursive call has to preserve

whatever the outer one had to, plus the frame of the outer call, storing the outer call's argument and return address and the outer call's *caller's* frame pointer from location 7.

Recursion is dealt with in the proof of Theorem 2 as one would expect, by adding $\texttt{fact} : [\![\texttt{nat} \to \texttt{nat}]\!](R_a)$ to the context. This is sound thanks to our use of indexing and interpretation of judgements:

Lemma 3 (Recursion). *For any Γ, l, R and $\texttt{M}$, if $\Gamma, l : R^\top \models \texttt{M} \rhd l : R^\top$ then $\Gamma \models \texttt{M} \rhd l : R^\top$.* □

Lemma 3, proved by a simple induction, suffices for first-order examples but only involves statically known labels.[5] We will discuss 'recursion through the store' in detail in future work, but here give a trivial example to indicate that we already have enough structure to deal with it. Consider *independently* verifying the following code fragments, assuming that $\texttt{wantzero} : (1 \mapsto \overline{\{(0,0)\}})^\top$

```
silly   : brz [1] wantzero          knot   : [0] <- silly
silly+1 : [1] <- [1]-1              knot+1 : jmp silly
silly+2 : jmp [0]
```

To show $\texttt{knot} : (1 \mapsto [\![\texttt{nat}]\!])^\top$, there are various choices for the specification assumed for `silly` (and proved of its implementation). An obvious one is that `silly` *expects* to be passed itself in 0, but this may be an overspecification. Alternatively, we can use the *recursive* specification $\mu r.\, ((0 \mapsto \overline{r}) \otimes (1 \mapsto [\![\texttt{nat}]\!]))^\top$, the semantics of which is given by well-founded induction: observe that the meaning of $R^\top$ at index k only depends on R at strictly smaller j. In general, we have a fixpoint equation

$$\mu r.\, (R[r])^\top = \left(\lambda(p,p').\lambda k.\, R\left[\mu r.\, (R[r])^\top\ (p,p')\, k\right]\ (p,p')\, k\right)^\top$$

letting us prove the following two judgements, which combine to give the result we wanted about `knot`:

Theorem 3.

1. $\texttt{wantzero} : \left(1 \mapsto \overline{\{(0,0)\}}\right)^\top \models \texttt{M}_{silly} \rhd \texttt{silly} : \mu r.\, ((0 \mapsto \overline{r}) \otimes (1 \mapsto [\![\texttt{nat}]\!]))^\top$
2. $\texttt{silly} : \mu r.\, ((0 \mapsto \overline{r}) \otimes (1 \mapsto [\![\texttt{nat}]\!]))^\top \models \texttt{M}_{knot} \rhd \texttt{knot} : (1 \mapsto [\![\texttt{nat}]\!])^\top$

□

7 Discussion

As we said in the introduction, this work is part of a larger project on relational parametricity for low-level code, which one might characterize as *realistic realizability*.[6] It should be apparent that we are drawing on a great deal of earlier

[5] This is equivalent to the more symmetric linking rule of our previous work [8].

[6] Modulo the use of unbounded natural numbers, etc. Our computational model is clearly only 'morally' realistic, but it's too nice a slogan not to use...

work on separation logic [21], relational program logics [19, 1, 7, 23], models and reasoning principles for dynamic allocation [18, 20, 9], typed assembly language [14], proof-carrying code [15], PER models of types [2], and so on.

Two projects with similar broad goals to ours are the FLINT project at Yale [11] and the Foundational Proof-Carrying Code project at Princeton [4]. The Yale group started with a purely syntactic approach to types for low-level code, and are now combining first-order Hoare-style reasoning using a semantic consequence relation within a more syntactic framework. This is argued to be simpler than techniques based on sophisticated constructions such as indexing, but the treatment of code pointers in [16] seems no less complex, and possibly less useful, than that of the present work. As the syntactic approach never says what types (or higher-order assertions) are supposed to ensure (what they actually *mean*), it seems more difficult to use it to combine proofs generated from different type systems or compilers, link in hand-written and hand-proved fragments or prove optimizations. The Princeton project takee a semantic approach, which is much closer to ours (as we've said, the step-indexing idea that we use comes from work on FPCC), but is still a fixed type system restricted to talking about a single form of memory safety rather than a general logic. FPCC uses a hardwired and rather limited form of allocation and has no deallocation at all [10].

There are other mechanized proofs of storage managers, including one by Yu et al. [24], and one using separation logic by Marti et al. [12]. These both treat more realistic implementations than we do here, but establish intensional 'internal' correctness properties of the implementations, rather than the more extensional and abstract specification used here. In particular, note that our specification uses no 'model variables' for recording the history of allocations.

Note that we make explicit use of second-order quantification over invariants, such as R_c, in our specifications and proofs (this is rather like row-polymorphism in record calculi). In separation logic, by contrast, the tight interpretation of pre-conditions means that $\{P\}\, C\, \{Q\}$ is semantically equivalent to $\forall I.\{P * I\}\, C\, \{Q * I\}$ so universal quantification over predicates on store outside the footprint of a command can be left implicit, but is still exploitable via the frame rule. Our use of explicit polymorphism is arguably more primitive (especially since procedures and modules require second order quantification anyway), doesn't rule out any programs and is closed under observations. On the other hand, the more modal-style approach of separation logic is simpler for simple programs and its stronger intensional interpretation of separation, whilst being more restrictive, has the significant advantage over ours that it extends smoothly to a concurrent setting.

The proof scripts for the general framework plus the verification of the allocator code and the factorial client currently total about 8,500 lines, which is excessively large. However, this really reflects my own incompetence with Coq, rather than any inherent impracticality of machine-checked proofs in this style. There are dozens of unused lemmas, variations on definitions, cut-and-pasted proofs and downright stupidities that, having learnt more about both Coq and the problem domain, I could now remove. The proofs of actual programs could

easily be made an order of magnitude shorter. We have an eye to using this kind of logic in PCC-style scenarios, for which mechanical checkability is certainly necessary. But working in Coq also caught errors in definitions and proofs. For example, we originally took $|R_{aret}(n)|$ to be simply $\overline{\mathbb{S} \times \mathbb{S}}$. The allocator does satisfy that specification, but a failed proof of a simple client revealed that it has a subtle flaw: if the block size is 1, the allocator can return location 0 itself (in location 0) as the free block.

Theorem 2 is only a semantic type soundness result – it does not say that the code *actually* computes factorials. In fact, only a couple of lines need tweaking to add the functional part of the specification too. We presented a type soundness result because that, rather than more general verification, is the direction of our immediate future plans. Once we have refactored our Coq definitions somewhat, we intend to investigate certified compilation of a small functional language in this style. We will also prove a slightly more interesting allocator which actually has a free list.

Although we have so far only focussed on proving a single program, a significant feature of the relational approach is that it can talk about *equivalence* of low-level code modulo a particular contextual contract. For example, one might hope to prove that all (terminating) allocators meeting our specification are observationally equivalent, or to verify the preservation of equational laws from a high-level language. Previous work on modularity, simulation and refinement in separation logic has run into some technical difficulties associated with the non-deterministic treatment of allocation [23, 13] which we believe are avoided in our approach. We also need to look more seriously at the adjoint perping operation, taking nat relations to nat×state relations [17, 22]. Making all relations be $(\cdot)^{\top\top}$-closed validates more logical principles and may be an alternative to step-indexing.

Thanks to Noah Torp-Smith for helping with an early version of this work, Josh Berdine for many useful discussions about separation, and Georges Gonthier for his invaluable advice and instruction regarding the use of Coq.

References

1. M. Abadi, L. Cardelli, and P.-L. Curien. Formal parametric polymorphism. *Theoretical Computer Science*, 121, 1993.
2. M. Abadi and G. D. Plotkin. A PER model of polymorphism and recursive types. In *Proc. 5th IEEE Symposium on Logic in Computer Science (LICS)*, pages 355–365. IEEE Computer Society Press, June 1990.
3. A. Ahmed. Step-indexed syntactic logical relations for recursive and quantified types. In *Proc. 15th European Symposium on Programming (ESOP)*, 2006.
4. A. Appel. Foundational proof-carrying code. In *Proc. 16th IEEE Symposium on Logic in Computer Science (LICS)*, 2001.
5. A. Appel and A. Felty. A semantic model of types and machine instructions for proof-carrying code. In *Proc. 27th ACM Symposium on Principles of Programming Languages (POPL)*, 2000.

6. A. Appel and D. McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(5), 2001.
7. N. Benton. Simple relational correctness proofs for static analyses and program transformations. In *Proc. 31st ACM Symposium on Principles of Programming Languages (POPL)*, January 2004. Revised version available from `http://research.microsoft.com/~nick/publications.htm`.
8. N. Benton. A typed, compositional logic for a stack-based abstract machine. In *Proc. 3rd Asian Symposium on Programming Languages and Systems (APLAS)*, volume 3780 of *Lecture Notes in Computer Science*, November 2005.
9. N. Benton and B. Leperchey. Relational reasoning in a nominal semantics for storage. In *Proc. 7th International Conference on Typed Lambda Calculi and Applications (TLCA)*, volume 3461 of *Lecture Notes in Computer Science*, 2005.
10. J. Chen, D. Wu, A. W. Appel, and H. Fang. A provably sound TAL for back-end optimization. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2003.
11. N. A. Hamid, Z. Shao, V. Trifonov, S. Monnier, and Z. Ni. A syntactic approach to foundational proof-carrying code. *Journal of Automated Reasoning*, 31(3-4), 2003.
12. N. Marti, R. Affeldt, and A. Yonezawa. Verification of the heap manager of an operating system using separation logic. In *Proc. 3rd Workshop on Semantics, Program Analysis and Computing Environments for Memory Management (SPACE)*, 2006.
13. I. Mijajlovic, N. Torp-Smith, and P. O'Hearn. Refinement and separation contexts. In *Proc. Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, December 2004.
14. G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(3), 1999.
15. G. Necula. Proof-carrying code. In *Proc. 24th ACM Symposium on Principles of Programming Languages (POPL)*, 1997.
16. Z. Ni and Z. Shao. Certified assembly programming with embedded code pointers. In *Proc. 33rd ACM Symposium on Principles of Programming Languages (POPL)*, 2006.
17. A. M. Pitts. Parametric polymorphism and operational equivalence. *Mathematical Structures in Computer Science*, 10, 2000.
18. A. M. Pitts and I. D. B. Stark. Operational reasoning for functions with local state. In *Higher Order Operational Techniques in Semantics*. CUP, 1998.
19. G. D. Plotkin and M. Abadi. A logic for parametric polymorphism. In *Proc. International Conference on Typed Lambda Calculi and Applications (TLCA)*, volume 664 of *Lecture Notes in Computer Science*, 1993.
20. U. S. Reddy and H. Yang. Correctness of data representations involving heap data structures. *Science of Computer Programming*, 50(1–3):129–160, March 2004.
21. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc. 17th IEEE Symposium on Logic in Computer Science (LICS)*, 2002.
22. J. Vouillon and P.-A. Mellies. Semantic types: A fresh look at the ideal model for types. In *Proc. 31st ACM Symposium on Principles of Programming Languages (POPL)*, 2004.
23. H. Yang. Relational separation logic. *Theoretical Computer Science*, 2004. Submitted.
24. D. Yu, N. A. Hamid, and Z. Shao. Building certified libraries for PCC: Dynamic storage allocation. *Science of Computer Programming*, 50, 2004.