Short Presentation: Abstracting Allocation: The New `new` Thing

Nick Benton
Microsoft Research
nick@microsoft.com

Noah Torp-Smith
The IT University of Copenhagen
noah@itu.dk

ABSTRACT

We sketch some work in progress on using binary relations for the modular specification and verification of low-level code for memory allocators and their clients.

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, functor categories and 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, hardwire allocation as a primitive.

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

For low-level languages, hardwiring allocation seems less attractive. Firstly, and most importantly, we would like to be able to reason about the low-level code that actually implements the storage manager. Secondly, in languages in which one can perform address arithmetic, such as the while-language with pointers used in separation logic, one is led to treat allocation as a non-deterministic operation, which seems to involve unpleasant semantic complexities, especially if one tries to reason about equivalences [6, 5]. Finally, it just doesn't correspond to the reality that 'machine code programs don't go wrong'.

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 '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, we work with (quantified) binary relations rather than unary predicates. Program properties are expressed in terms of equivalence of observable behaviour, rather than avoidance of some artificial stuck states. Secondly, we use 'perping' operations, taking relations on states to relations on code addresses (and vice-versa) 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 to establish soundness. The hope is that this basic framework can provide a unified treatment of many type systems, static analyses and program transformations. Here, we merely present some preliminary ideas on how one might use it to specify and verify the implementation and clients of a very basic memory allocator. This is elementary, and very much work in progress, but we believe it is a fresh approach to freshness.

SPACE 2006. Charleston, South Carolina.

2. THE MACHINE

Our idealized sequential machine model looks like:

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

The instruction set, Instr, includes `halt`, direct and indirect stores and loads, some (total) arithmetic and logical operations, and conditional and unconditional branches. 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 $\langle p|s|l\rangle \to^\omega$. Omitting registers was a misguided attempt at simplification; we intend to add them. The other major idealizations are the use of unbounded natural numbers and the separation of code and data memory.

3. RELATIONS

We work with binary relations on the state, $R \subseteq \mathbb{S}\times\mathbb{S}$ and an explicit notion [4] of the 'support' of a relation, i.e. which locations it depends upon. These supports are in general themselves functions of the state, but should not depend on locations not in their results. More formally:

- If $L \subseteq \mathbb{N}$ and $s, s' \in \mathbb{S}$ then we write $s \sim_L s'$ to mean $\forall l \in L. s(l) = s'(l)$.
- If $A : \mathbb{S} \to \mathbb{P}(\mathbb{N})$ then A is an *accessibility map* if for all s, s', $s \sim_{A(s)} s' \implies A(s') = A(s)$.
- A *supported state relation* R is a pair $(|R|, A_R)$ where $|R| \subseteq \mathbb{S}\times\mathbb{S}$, A_R is an accessibility map and for all $s_1 \sim_{A_R(s_1)} s_1'$, $s_2 \sim_{A_R(s_2)} s_2'$, if $(s_1, s_2) \in |R|$ then $(s_1', s_2') \in |R|$. We often elide the $|\cdot|$.

Note that accessibility maps do not necessarily correspond to sets of 'reachable' locations. The separating product of supported relations is given by $R\otimes R' = (|R\otimes R'|, \lambda s. A_R(s) \cup A_{R'}(s))$ where

$$|R\otimes R'| = |R| \cap |R'| \cap \{(s_1, s_2) \mid A_R(s_i) \cap A_{R'}(s_i) = \emptyset, i = 1, 2\}$$

If R is a (supported) state relation, and p a program, we define $R^\top(p) \subseteq \mathbb{N}\times\mathbb{N}$ by

$$R^\top(p) \stackrel{def}{=} \{(l, l') \mid \forall(s, s') \in R. \langle p|s|l\rangle \Downarrow \iff \langle p|s'|l'\rangle \Downarrow\}$$

In other words, $R^\top$ relates two labels if jumping to those labels gives equivalent termination behaviour whenever the two initial states are related by R.

4. SPECIFICATION OF ALLOCATION

An allocator is just a piece of sequential machine code, which we can specify and verify using the same relational methods that we will use for its clients. There are entry points for initialization, allocation and deallocation, though we only discuss the first two here.

We arbitrarily choose to use locations 0-4 for arguments and results, and designate locations 5-9 as being callee-saves. 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 label `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.

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, we do so extensionally, rather than in terms of which locations are read, written or reachable.

There will be some supported relation (R_a, A_a) for the private invariant of the allocator. A_a captures the store owned by the allocator; this has to be a function of the store because it varies. R_a says which internal states are valid and 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 for any program p extending the allocator module, and for any supported relation (R_{pres}, A_{pres}) where A_{pres} is the constant accessibility map $\lambda s.\{5, \ldots, 9\}$,

$$(\texttt{init}, \texttt{init}) \in (R_{iret} \otimes R_{pres})^\top(p) \qquad (1)$$

where R_{iret} is the supported relation

$$(\{(s, s') \mid (s(0), s'(0)) \in (T_{04} \otimes R_{pres} \otimes R_a)^\top(p)\}, \lambda s.\{0\})$$

and T_{04} is $(\mathbb{S}\times\mathbb{S}, \lambda s.\{0, \ldots, 4\})$.

When `alloc` is called, the client 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, for any p extending the allocator, for any R_c, for any n

$$(\texttt{alloc}, \texttt{alloc}) \in (R_{aparms}(n) \otimes R_c \otimes R_a)^\top(p) \qquad (2)$$

where $A_{aparms}(n) = \lambda s.\{0, \ldots, 4\}$, $|R_{aparms}(n)|$ is

$$\{(s, s') \mid \begin{array}{l}(s(0), s'(0)) \in (R_{aret}(n) \otimes R_c \otimes R_a)^\top(p)\\ \wedge\ s(1) = n \wedge s'(1) = n\}\end{array}$$

and $R_{aret}(n) = (\mathbb{S}\times\mathbb{S}, \lambda s.\{0\} \cup \{s(0), \ldots, s(0) + n - 1\})$.

$|R_{aret}(n)|$ being simply $\mathbb{S}\times\mathbb{S}$ is what requires the return addresses to behave equivalently whatever chunk of store they're given. Abbreviating the notation somewhat, the specification of the allocator module might be written as

$$\begin{array}{rl}\Gamma_a \stackrel{def}{=} & \exists R_a.(\texttt{init} : \forall R_{pres} : A_{pres}.(R_{iret} \otimes R_{pres})^\top \wedge \\ & \qquad \texttt{alloc} : \forall R_c.\forall n.(R_{aparms}(n) \otimes R_c \otimes R_a)^\top)\end{array}$$

Note that R_a is scoped across both labels.

5. VERIFICATION OF ALLOCATION

We now consider verifying the simplest (useful) allocation module, defined by the following code fragment $\texttt{M}_a$:

```
   init :   [10] ← 11
 init + 1 : jmp [0]
  alloc :   [2] ← [0]
alloc + 1 : [0] ← [10]
alloc + 2 : [10] ← [10] + [1]
alloc + 3 : jmp [2]
```

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.

The witness for the implementation $\mathtt{M}_a$ is just

$$|R_a| \overset{def}{=} \{(s,s') \mid (s(10) > 10) \wedge (s'(10) > 10)\}$$

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 state, location 10 itself.

We then show that for any program p extending $\mathtt{M}_a$, the properties (1) and (2) actually hold. This is essentially relational Hoare-style verification, using separation assumptions. In particular, the prerelation for `alloc` lets us assume that the support A_c of R_c is disjoint from both A_{aparms} and A_a in each of the related states (s,s') in which we make the initial call. Since the code only writes to locations coming from those latter two accessibility maps, we know that A_c is unchanged in the two return states, and that they are still related by R_c, even though we do not know anything more about R_c or A_c.

We can also verify client programs under the assumption that the allocator satisfies its specification. If `C` is a client fragment and Γ is a relational specification on labels occuring in `C`, then what we will need to show is that for any p extending `C`, if p satisfies Γ_a then p satisfies Γ. Then if `C` is disjoint from $\mathtt{M}_a$, we'll be able to deduce that any extension of `C` linked with $\mathtt{M}_a$ satisfies both Γ and Γ_a. Client specifications have to mention R_a, even though they treat it abstractly, so we prove that for *every* R_a, if p extends `C` and satisfies (1) and (2), then p satisfies Γ. In judgemental form, this would look like

$$R_a;\, \mathtt{init} : \forall R_{pres}.(\ldots)^{\top}, \mathtt{alloc} : \forall R_c.(\ldots)^{\top} \vdash \mathtt{C} \rhd \Gamma$$

where R_a is universally quantified over the whole judgement.

6. LINKING, RECURSION AND INDEXING

For proving properties of particular small program fragments, one can work directly in the semantics. This deals with mutual recursion on a case-by-case basis and involves working explicitly with the inductive definition of termination every time. We need more generic reasoning principles, such as a general linking rule [3]:

$$\frac{\Theta;\Gamma_2 \vdash M_1 \rhd \Gamma_1 \qquad \Theta;\Gamma_1 \vdash M_2 \rhd \Gamma_2}{\Theta; - \vdash M_1, M_2 \rhd \Gamma_1 \wedge \Gamma_2}$$

Such rules require the Γs to be suitably admissible, which we ensure by expressing each of them as the limit of a sequence of finite approximations defined in terms of counting steps in the semantics [2]. The k-step approximation of 'equivalent behaviour' is 'indistinguishable when tested for up to k steps', and 'equivalent behaviour' proper is then the intersection of all its finite approximants. More formally, we index all our relations by naturals and define

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

where $\Downarrow_j$ means 'converges in j steps'. We then expect to be able to justify rules like that above, provided that we strengthen the semantics of the individual judgements we prove so that $\Gamma' \vdash M : \Gamma$ means that for all p extending M, for all k, if p satisfies $\Gamma'(k,p)$ then p satisfies $\Gamma(k+1,p)$. This is usually clear, since the code in M will take at least one step before relying on any of the assumptions in Γ'.

7. DISCUSSION

This is part of a grander programme on low-level semantics for high-level types that one might (not entirely accurately) call 'realistic realizability'. It should be apparent that this builds on a great deal of earlier work on separation logic, relational Hoare logic, models of dynamic allocation, typed assembly language, proof-carrying code, PER models of types, and so on. Space precludes even beginning to give proper references here, but we should mention the Princeton FPCC project [1], which has a very similar vision and from which we took (amongst other things) the step-indexing idea. The distinctive features of our (much less developed) approach are the use of potentially very expressive, extensional, binary relations, rather than unary predicates built from a particular notion of memory safety, and our approach to modular reasoning about the heap.

We are working in the Coq theorem prover to manage changing definitions and the detail of proofs about particular low-level programs. We have no concrete program logic yet, but envisage turning semantic definitions and lemmas formalized in Coq into syntax and inference rules for a more application-specific logic in due course.

Our next steps are to do more complex examples of allocators and clients, develop the metatheory of indexing and perping and to take steps towards a more convenient logic. We then plan to look at more generic correctness proofs for compilation schemes from a range of high-level languages into low-level code, and at applications to program transformation. We have so far concentrated on relational properties of a single program, and some parts of the framework described here (e.g. working with a single program and supporting relations by a single accessibility map) and of our particular specifications (e.g. requiring the allocated block size to be the same on both sides in the allocator spec) are not quite general enough to deal with interesting transformations.

Finally, we thank Georges Gonthier for his invaluable advice and instruction regarding the use of Coq.

8. REFERENCES

[1] A. Appel. Foundational proof-carrying code. In *Symposium on Logic in Computer Science*, 2001.

[2] A. Appel and D. McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM TOPLAS*, 23(5), 2001.

[3] N. Benton. A typed, compositional logic for a stack-based abstract machine. In *Proc. APLAS 2005*, LNCS 3780.

[4] N. Benton and B. Leperchey. Relational reasoning in a nominal semantics for storage. In *Proc. TLCA 2005*, LNCS 3461.

[5] N. Torp-Smith. *Advances in Separation Logic.* PhD thesis, IT University of Copenhagen, 2005.

[6] H. Yang. Relational separation logic. Submitted to TCS, 2004.