Appendix of Quiver: Guided Abductive Inference of Separation Logic Specifications in Coq

SIMON SPIES, MPI-SWS, Germany
LENNARD GÄHER, MPI-SWS, Germany
MICHAEL SAMMLER, MPI-SWS, Germany
DEREK DREYER, MPI-SWS, Germany

Contents

$$
\begin{array}{rrcl}
\text{Types} & A, B & ::= & \mathsf{void} \mid \mathsf{null} \mid \mathsf{num}[\mathit{it}]\, n \mid \mathsf{any}\, n \mid \mathsf{zeros}\, n \mid \mathsf{value}\, n\, v \mid \exists x.\, A\, x \mid A * P \\
 & & \mid & \mathsf{own}\, \ell\, A \mid \mathsf{optional}\, \phi\, A \mid \mathsf{fn}\, T \mid \vec{x} \mathbin{@} \mathrm{P} \mid \mathsf{struct}[s]\, \vec{A} \mid \mathsf{array}[\mathrm{P}]\, xs \mid \cdots \\
\text{Resources} & M, N & ::= & v \triangleleft_v A \mid \ell \triangleleft_l A \mid \mathsf{block}\, \ell\, n \mid \cdots \\
\text{Embedded Goal} & E & ::= & \mathbf{wp}\; e\, \{v, A.\, \Phi\, v\, A\} \mid \mathbf{cast}\; (\mathit{it}_2)_{\mathit{it}_1} (v : A)\{w, B.\, \Phi\, w\, B\} \mid \cdots
\end{array}
$$

Fig. 1. Thorium types.

Type	**Description**
void	void type
null	null
$\mathsf{num}[\mathit{it}]\, n$	integer n of C integer type it
$\mathsf{bool}[\mathit{it}]\, b$	boolean b of C integer type it
$\mathsf{any}\, n$	arbitrary bytes of length n
$\mathsf{zeros}\, n$	zero-bytes of length n
$\mathsf{value}\, n\, v$	the value v, which has size n
$\mathsf{own}\, \ell\, A$	owned pointer ℓ to a value of type A
$\mathsf{optional}\, \phi\, A$	optional pointer, NULL if ϕ is false, A otherwise
$\mathsf{opt}\, v$	the value v, which is guaranteed to be a location
$\exists x.\, A\, x$	type-level existential quantification
$A * P$	type-level separating conjunction
$\mathsf{struct}[s]\, \vec{A}$	type for struct s with fields $\vec{A}$
$\mathsf{array}[\mathrm{P}]\, xs$	type for arrays where the elements are of type P
$\mathsf{slices}\, n\, [\overrightarrow{(i, li, A)}]$	a sequence of segments of potentially different lengths with offsets i, lengths li, and types A
$\mathsf{fn}\, T$	function type with predicate transformer specification T

Fig. 2. Thorium types and their intuitive meaning.

A THE TYPE SYSTEM THORIUM

The type system Thorium is a separating logic based type system for scaling Quiver to the complexities of C. We explain the approach of using refinement types in separation logic (§A.1), and we discuss how they integrate with abductive deductive verification (§A.2).

A.1 Separation Logic with Refinement Types à la RefinedC

Following the footsteps of RefinedC [5], instead of abstract predicates $\mathrm{P}(v, x)$ and points-to assertions $\ell \mapsto v$, the resources in Thorium are *type assignments*. They are of the form $v \triangleleft_v A$ (read "v is an A") and $\ell \triangleleft_l A$ (read "ℓ stores an A"; semantically $(\exists v.\, \ell \mapsto v * v \triangleleft_v A) \vDash \ell \triangleleft_l A$). We discuss the kind of types A Thorium offers and how they guide the proof search.

Types. The types of Thorium are depicted in Fig. 1, and an overview of their meaning is given in Fig. 2. We explain the most important types by recalling the range example from the paper. In C, it would be declared as `typedef struct ran {int s; int e} *range;` and, for the predicate $\mathrm{range}(v, n_s, n_e)$, the analogous Thorium type is defined as:

$$(n_s, n_e) \mathbin{@} \mathsf{range} \equiv_{\mathsf{ty}} \exists \ell.\, \mathsf{own}\, \ell\, (\mathsf{struct}[\mathtt{ran}]\, [A_s; A_e]) * 0 \le n_s \le n_e * \mathsf{block}\, \ell\, sz_{\mathtt{ran}}$$
$$\textit{where}\ \ A_s \triangleq \mathsf{num}[\mathtt{int}]\, n_s,\ A_e \triangleq \mathsf{num}[\mathtt{int}]\, n_e, \textit{and}\ sz_{\mathtt{ran}} \triangleq \mathbf{sizeof}(\mathbf{struct}\ \mathtt{ran})$$

$$
\begin{aligned}
&\mathbf{wp}\ (it_2)_{it_1} e\ \{\Phi\} \ \dashv\ \mathbf{wp}\ e\ \{v, A.\ \mathbf{cast}\ (it_2)_{it_1}(v : A)\{\Phi\}\} && (\textsc{ty-cast-wp})\\
&\mathbf{cast}\ (\texttt{size\_t})_{\texttt{int}}(v : \mathsf{num}[\texttt{int}]\ n)\{\Phi\} \ \dashv\ n \geq 0 * \forall w.\ \Phi\, w\, (\mathsf{num}[\texttt{size\_t}]\ n) && (\textsc{ty-cast-sizet})\\
&\mathbf{cast}\ (it_2)_{it_1}(v : \overline{A})\{\Phi\} \ \dashv\ \exists n.\ (\overline{A} \sqsubseteq \mathsf{num}[it_1]\ n) * \mathbf{cast}\ (it_2)_{it_1}(v : \mathsf{num}[it_1]\ n)\{\Phi\} && (\textsc{ty-cast-def})\\
&\mathbf{wp}\ v\ \{\Phi\} \ \dashv\ v \lhd_v A * \Phi\, v\, A \ \textit{when}\ v \lhd_v A && (\textsc{ty-val})\\
&\mathbf{wp}\ v\ \{\Phi\} \ \dashv\ \exists \overline{A}.\ v \lhd_v \overline{A} * \Phi\, v\, \overline{A} && (\textsc{ty-val-missing})\\
&\mathbf{conv}(v : \mathsf{num}[it]\ n)(x, y, \_.\ \mathsf{num}[x]\ y)\{\Phi\} \ \dashv\ \Phi(\lambda x, y, \_.\ x = it * y = n) && (\textsc{ty-conv-int})\\
&\mathbf{conv}(v : \mathsf{own}\ \ell\ A)(x, y.\ \mathsf{own}\ x\ (B\, x\, y))\{\Phi\} \ \dashv\ \ell \lhd_l A \mathrel{-\!\!*} \Phi(\lambda x.\ x = \ell * \ell \lhd_l B\, x\, y) && (\textsc{ty-conv-own})\\
&\mathbf{call}(v : \mathsf{fn}\ T)(w : B)\{\Phi\} \ \dashv\ w \lhd_v B \mathrel{-\!\!*} \mathbf{apply}(T\, w)\{\Phi\} && (\textsc{ty-call})\\
&\mathbf{ex}(x.\ v \lhd_v Bx * Gx \mid Sx) \ \dashv\ \mathbf{wp}\ v\ \{w, A.\ \mathbf{conv}(w : A)(x.\ Bx)\{S'.\ \mathbf{ex}(x.\ S'x * Gx \mid Sx)\}\} && (\textsc{ex-conv})\\
&\mathbf{ex}(x.\ v\, x \lhd_v B\, x * G\, x \mid S\, x) \ \dashv\ \mathbf{ex}(x.\ G\, x \mid v\, x \lhd_v B\, x * S\, x) && (\textsc{ex-conv-blocked})
\end{aligned}
$$

Fig. 3. A selection of Thorium typing rules (ty-) and existential instantiation rules (ex-).

Types of the form $\vec{x}\,@\,\mathrm{P}$ correspond to user-defined abstract predicates and are defined via a (possibly recursive) equation $\vec{x}\,@\,\mathrm{P} \equiv_{\mathrm{ty}} A$. The type $(n_{\mathrm{s}}, n_{\mathrm{e}})\,@\,\mathrm{range}$ ensures that its values are owned pointers ℓ (via "$\mathsf{own}\ \ell\ A$") to a ran-struct (via "$\mathsf{struct}[s]\ \vec{A}$") with two fields: s containing int-integer n_{s} (via "$\mathsf{num}[it]\ n$"), and e containing the int-integer n_{e}. To hide the location ℓ, we use type-level existential quantification "$\exists x.\ A\, x$" and, to impose the bounds constraint $0 \leq n_{\mathrm{s}} \leq n_{\mathrm{e}}$, we use type-level separating conjunction "$A * P$". Besides the bounds constraint, the type carries an additional constraint: the resource "$\mathsf{block}\ \ell\ n$". It tracks the length of dynamically allocated blocks to ensure that ownership of the entire block is given up when freeing ℓ (see Appendix B for more details).

Typing rules. In Thorium, instead of a standard weakest precondition $\mathbf{wp}\ e\ \{v.\ \Phi\, v\}$, we use *typed weakest preconditions* $\mathbf{wp}\ e\ \{v, A.\ \Phi\, v\, A\}$: their postcondition Φ is about the resulting value v *and, additionally,* its type A.[1] These typed weakest preconditions seamlessly integrate types into goal-directed proof search. It works as follows: Instead of vanilla weakest precondition rules, we use *typing rules* in Thorium, a selection depicted in Fig. 3. There are two kinds of typing rules: (a) structural rules that descend into terms (akin to wp-let in the paper) and (b) type-directed rules that match on types to steer the proof search (akin to wp-assign in the paper).

To explain how they interact, we consider an example, which dereferences a location ℓ_{k} and casts the resulting integer from int to size_t:

$$\ell_{\mathrm{k}} \lhd_l \mathsf{num}[\texttt{int}]\ n_{\mathrm{k}} * [\_] \vdash \mathbf{wp}\ (\texttt{size\_t})_{\texttt{int}}(*\ell_{\mathrm{k}})\ \{\Phi\}$$

First, using a *structural rule* (ty-cast-wp), we descend into the type cast, leaving

$$\ell_{\mathrm{k}} \lhd_l \mathsf{num}[\texttt{int}]\ n_{\mathrm{k}} * [\_] \vdash \mathbf{wp}\ *\ell_{\mathrm{k}}\ \{v, A.\ \mathbf{cast}\ (\texttt{size\_t})_{\texttt{int}}(v : A)\{\Phi\}\}$$

Once it has been determined that the result of $*\ell_{\mathrm{k}}$ is some value v of type $\mathsf{num}[\texttt{int}]\ n_{\mathrm{k}}$, we then encounter the following goal in the postcondition of **wp**

$$\ell_{\mathrm{k}} \lhd_l \mathsf{num}[\texttt{int}]\ n_{\mathrm{k}} * [\_] \vdash \mathbf{cast}\ (\texttt{size\_t})_{\texttt{int}}(v : \mathsf{num}[\texttt{int}]\ n_{\mathrm{k}})\{\Phi\}$$

It uses"$\mathbf{cast}\ (it_2)_{it_1}(v : A)\{\Phi\}$", an auxiliary judgment for C-level type casts, which is overloaded based on the type A. We use a *type-directed rule* to proceed: The rule (ty-cast-sizet) handles the cast

[1]In fact, in the Coq development, we have three such weakest preconditions, one for each syntactic category of Caesium: right expressions, left expressions, and statements. We focus here on the **wp** for right expressions.

from `int` to `size_t` by checking that the `int`-integer n (here n_k) is nonnegative and then calling the postcondition with n as a `size_t`-integer.

A.2 Abductive Deductive Verification with Refinement Types

We discuss how *abductive deductive verification* $\Delta * [R] \vdash \mathbf{wp}\ e\ \{v, A.\ \Phi\, v\, A\}$ interacts with the type system Thorium. We focus on the three most important aspects, including how we deal with *incomplete information* about the context Δ. That is—like RefinedC—we use types to guide the proof search, but—unlike RefinedC—the context Δ is incomplete and, therefore, we may have to infer type assignments as part of the precondition R.

Incomplete information and abstract types. The first aspect is *abstract types* $\overline{A}$, which we introduce to address incomplete information. That is, as discussed in §A.1, the structural rules of Thorium descend into terms. At the leaves, when we encounter a value $\Delta * [R] \vdash \mathbf{wp}\ v\ \{v, A.\ \Phi\, v\, A\}$, we have two options:[2] (a) the value is contained in Δ (TY-VAL) or (b) the type assignment of v should be part of the precondition R. In the latter case, there is an issue: the postcondition Φ demands a type A, but *locally* we do not know yet which type v should have. To solve this issue, we introduce an abstract type $\overline{A}$ (TY-VAL-MISSING), which serves as a placeholder for the type of v. Then, as we continue with the postcondition Φ, we collect constraints on $\overline{A}$. To be precise, we provide "default rules" that impose constraints on $\overline{A}$ when v is *used*—*e.g.*, via the rule TY-CAST-DEF, which requires $\overline{A}$ to be a `num[`it_1`]`-type. During simplification $P \Rrightarrow Q$ (SIMPLIFY in the paper), we use these constraints to instantiate existentials analogous to equalities (in $P \Rrightarrow_{\mathrm{ex}} Q$). Thorium has default rules for unary operators, binary operators, pointer dereference, conditionals, struct field access, pointer arithmetic, *etc.*

Type conversion. The second aspect is type conversion, which integrates types into existential instantiation. That is, by extending the rules of $\mathbf{ex}(x.\ G\, x)$, the *type conversion judgment* $\mathbf{conv}(v : A)(x.\ B\, x)\{\Phi\}$ turns the type assignment $v \lhd_v A$ into $v \lhd_v B\, x$ where "x" is existentially quantified. When we encounter a type assignment in $\mathbf{ex}$, we either (a) trigger type conversion (EX-CONV) or (b) put it on the "blocked-stack" if the value v still depends on x (EX-CONV-BLOCKED). In the first case (EX-CONV), we (1) determine the type of v (using $\mathbf{wp}$), (2) determine a precondition S for the type conversion to succeed (using $\mathbf{conv}$), and (3) return to the existential instantiation (using $\mathbf{ex}$). For example, TY-CONV-INT constrains the C-level integer type x to it (*e.g.*, `int`, `size_t`, …) and the mathematical integer y to n; TY-CONV-OWN constrains the location x to ℓ and, eventually, will lead to type conversion for $\ell \lhd_l B\, x\, y$.

Predicate transformers and joining. The third aspect are predicate transformer specifications. As explained in the paper, our abductive deductive verification judgment infers predicate transformers T. To integrate them into Thorium, we use a type $\mathsf{fn}\ T$ for function pointers and the rule TY-CALL to call them. The rule turns the goal into $\mathbf{apply}(T\, w)\{\Phi\}$, which—by using $\mathbf{ex}$ internally—inherits Argon's support for existential instantiation and Thorium's support for type conversion.

Besides using predicate transformers as function types, there is a second interaction of Thorium and predicate transformers: Thorium provides a heuristic for joining them. That is, when we infer the precondition of a conditional $\mathbf{if}\ \phi\ \mathbf{then}\ G_1\ \mathbf{else}\ G_2$, we first obtain two separate preconditions $R_1(\Phi)$ and $R_2(\Phi)$. Usually, Argon turns them into the precondition $R(\Phi) \triangleq \mathbf{if}\ \phi\ \mathbf{then}\ R_1(\Phi)\ \mathbf{else}\ R_2(\Phi)$ (see ABD-IF in the paper). As an alternative, Thorium provides a heuristic to join them into

[2]Technically, there is a third option: extending the precondition R with the assumption $v = w$ for some $w \lhd_v A$ in Δ and proceeding with $\Phi\, w\, A$. We exclude this option to limit the search space. In doing so, we follow the footsteps of bi-abduction [1, 2], which does not infer "$R \triangleq (\ell = r \land a = b)$" and "$F \triangleq \mathbf{True}$" as a solution for its bi-abduction judgment $\ell \mapsto a * [R] \vdash r \mapsto b * [F]$ for the same reason.

a single precondition $(T \sqcup_\phi T')(\Phi)$, which can be activated on demand (using the annotation "`[[q::join_if]]`" in the implementation). For example, the heuristic would join the assignment $v \triangleleft_v \mathsf{num}[\mathit{it}]\ n_1 \sqcup_\phi v \triangleleft_v \mathsf{num}[\mathit{it}]\ n_2$ into $v \triangleleft_v \mathsf{num}[\mathit{it}]\ (\mathsf{if}\ \phi\ \mathsf{then}\ n_1\ \mathsf{else}\ n_2)$, pushing the conditional further into the precondition. Unfortunately, the heuristic is not always successful due to the considerable expressiveness of Thorium types: As in RefinedC, types can contain nesting (via $\mathsf{own}\ \ell\ A$), existential quantification (via $\exists x.\ A\ x$), separating conjunction (via $A * P$), conditionals (via $\mathsf{optional}\ \phi\ A$), and even recursion (via $x \mathbin{@} \mathrm{P}$), making joining hard. If the heuristic fails, we can still default back to precondition that Argon would otherwise pick, using $(T \sqcup_\phi T')(\Phi) = \textbf{if}\ \phi\ \textbf{then}\ R_1(\Phi)\ \textbf{else}\ R_2(\Phi)$.

$$T_{\texttt{malloc}}(v_{\mathsf{n}})(\Phi) \triangleq$$
$$\exists n.\ (v_{\mathsf{n}} \lhd_v \mathsf{num}[\texttt{size\_t}]\ n) * \begin{pmatrix} \forall w.\ w \lhd_v \mathsf{null} \mathrel{-\!\!*} \Phi\, w \\ \wedge \\ \forall w, \ell.\ w \lhd_v \mathsf{own}\ \ell\ (\mathsf{any}\ n) * \mathsf{block}\ \ell\ n \mathrel{-\!\!*} \Phi\, w \end{pmatrix}$$

$$T_{\texttt{free}}(v_{\mathsf{x}})(\Phi) \triangleq$$
$$\exists \ell, v, n.\ (v_{\mathsf{x}} \lhd_v \mathsf{own}\ \ell\ (\mathsf{value}\ n\ v) * \mathsf{block}\ \ell\ n) * \forall w.\ w \lhd_v \mathsf{void} \mathrel{-\!\!*} \Phi\, w$$

$$T_{\texttt{memcpy}}(v_{\texttt{dst}}, v_{\texttt{src}}, v_{\texttt{sz}})(\Phi) \triangleq$$
$$\exists \ell, r, n, v.\ v_{\texttt{dst}} \lhd_v \mathsf{own}\ \ell\ (\mathsf{any}\ n) * v_{\texttt{dst}} \lhd_v \mathsf{own}\ \ell\ (\mathsf{value}\ n\ v) * v_{\texttt{sz}} \lhd_v \mathsf{num}[\texttt{size\_t}]\ n\ *$$
$$\forall w.\ (\ell \lhd_l \mathsf{value}\ n\ v) * (r \lhd_l \mathsf{value}\ n\ v) * (w \lhd_v \mathsf{value}\ sz_{\texttt{ptr}}\ \ell) \mathrel{-\!\!*} \Phi\, w$$

$$T_{\texttt{memset}}(v_{\texttt{dst}}, v_{\texttt{val}}, v_{\texttt{sz}})(\Phi) \triangleq$$
$$\exists \ell, n, v.\ v_{\texttt{dst}} \lhd_v \mathsf{own}\ \ell\ (\mathsf{any}\ n) * v_{\texttt{val}} \lhd_v \mathsf{num}[\texttt{int}]\ 0 * v_{\texttt{sz}} \lhd_v \mathsf{num}[\texttt{size\_t}]\ n\ *$$
$$\forall w.\ \ell \lhd_l \mathsf{zeros}\ n * w \lhd_v \mathsf{value}\ sz_{\texttt{ptr}}\ \ell \mathrel{-\!\!*} \Phi\, w$$

$$T_{\texttt{realloc}}(v_{\texttt{src}}, v_{\texttt{sz}})(\Phi) \triangleq$$
$$\exists v, n.\ v_{\texttt{src}} \lhd_v \mathsf{opt}\ v * v_{\texttt{sz}} \lhd_v \mathsf{num}[\texttt{size\_t}]\ n\ *$$
$$\textbf{if}\ \neg\neg(v = \mathsf{NULL})\ \textbf{then}\ (\forall w.\ w \lhd_l \mathsf{null} \mathrel{-\!\!*} \Phi\, w) \wedge (\forall r, w.\ w \lhd_v \mathsf{own}\ r\ (\mathsf{any}\ n) * \mathsf{block}\ r\ n \mathrel{-\!\!*} \Phi\, w)$$
$$\textbf{else}\ \exists \ell, m, v'.\ v \lhd_v \mathsf{own}\ \ell\ (\mathsf{value}\ m\ v') * \mathsf{block}\ \ell\ m * m \le n\ *$$
$$\begin{pmatrix} \forall w.\ \ell \lhd_l \mathsf{value}\ m\ v' * \mathsf{block}\ \ell\ m * w \lhd_v \mathsf{null} \mathrel{-\!\!*} \Phi\, w \\ \wedge \\ \forall w, r.\ r \lhd_l \mathsf{slices}\ n\ [(0, m, \mathsf{value}\ m\ v'); (m, n{-}m, \mathsf{any}\ (n{-}m))] * \mathsf{block}\ r\ n * w \lhd_v \mathsf{value}\ sz_{\texttt{ptr}}\ r \mathrel{-\!\!*} \Phi\, w \end{pmatrix}$$

$$T_{\texttt{abort}}()(\Phi) \triangleq \forall w.\ (\mathsf{False} * w \lhd_v \mathsf{void}) \mathrel{-\!\!*} \Phi\, w$$

Fig. 4. Standard Library Specifications, where $sz_{\texttt{ptr}} \triangleq \textbf{sizeof}(\textbf{void*})$.

B STANDARD LIBRARY

For the standard library functions `malloc`, `free`, `memcpy`, `memset`, `realloc`, and `abort`, we assume the specifications depicted in Fig. 4. Several of the memory functions use an (abstract) separation logic resource block $\ell\ n$. It keeps track of the size of an allocated block (pointed to by ℓ) and ensures that, when we free the block with `free`, the ownership of the entire block is given up. We briefly discuss the specifications.

The predicate transformer $T_{\texttt{malloc}}$ takes a `size_t`-integer n and returns either NULL, or it returns a freshly allocated block (pointed to by location ℓ) of size n. The memory in the allocated block is initially arbitrary (any n). To express the two cases in a predicate transformer, we use a conjunction, which ensures that clients of `malloc` have to consider *both* cases (*i.e.*, they have to prove the NULL case *and* the case where a valid pointer is returned).

The predicate transformer $T_{\texttt{free}}$ takes an owned pointer to arbitrary memory ℓ of size n and frees the contents. To ensure that the entire ownership is given up, we also have to supply the block $\ell\ n$ predicate to ensure that n is indeed the size of the allocation block at location ℓ.

The predicate transformer $T_{\texttt{memcpy}}$ takes an owned pointer ℓ for the destination (to n arbitrary bytes), an owned pointer to the source value v (of size n), and the `size_t`-integer n itself. It then copies over the contents such that the locations ℓ and r both store v at the end of the function. (It returns the value ℓ.)

The predicate transformer $T_{\texttt{memset}}$ overwrites the contents of a destination pointer ℓ (of size n) with $v_{\texttt{val}}$. Our specification specializes the value $v_{\texttt{val}}$ to be `0` (of `int` type), since our examples only use this common special case.

The predicate transformer $T_{\texttt{realloc}}$ describes the different modes of using `realloc`. The argument $v_{\texttt{src}}$ is an optional pointer (with value v) and $v_{\texttt{sz}}$ is the desired size after extension. If the argument $v_{\texttt{src}}$ (*i.e.*, v) is `NULL`, then $T_{\texttt{realloc}}$ corresponds to the predicate transformer for `malloc`. That is, the allocation may fail (returning null) or succeed, allocating a fresh pointer of size n. If the argument $v_{\texttt{src}}$ is not `NULL`, then we have to provide the ownership of the corresponding pointer ℓ (storing value v') and the memory block predicate block $\ell\ n$. In this specification, we have to ensure that the desired length n is longer than the current length m of ℓ. In this case, there are two possible outcomes: (1) the allocation fails, returning null and giving back the ownership of ℓ, or (2) the allocation succeeds, consuming the ownership of ℓ and returning a "fresh" location r. This new location r points to n bytes of memory where the first m bytes are the original value v' and the remaining $n - m$ bytes are arbitrary, uninitialized memory. The return value in this case is the "reallocated" location r.

The predicate transformer $T_{\texttt{abort}}$ terminates an execution. We consider early termination safe behavior (*i.e.*, it is not a safety violation). Hence, the postcondition is False (and the return value of type void).

xmalloc

```
void *xmalloc(size_t size) {
    void *ptr = malloc(size);
    if (ptr == NULL) {
        abort();
    }
    return ptr;
}
```

$$T_{\texttt{xmalloc}}(v_{\texttt{size}})(\Phi) \triangleq$$
$$\exists n.\ (v_{\texttt{size}} \lhd_v \texttt{num[size\_t]}\ n) * (\forall w, \ell.\ w \lhd_v \mathsf{own}\ \ell\ (\mathsf{any}\ n) * \mathsf{block}\ \ell\ n \mathbin{-\!\!*} \Phi\ w)$$

xrealloc

```
void *xrealloc(void *ptr, size_t size) {
    void *new_ptr = realloc(ptr, size);
    if (new_ptr == NULL) {
        abort();
    }
    return new_ptr;
}
```

$$T_{\texttt{xrealloc}}(v_{\texttt{ptr}}, v_{\texttt{size}})(\Phi) \triangleq$$
$$\exists v, n.\ v_{\texttt{ptr}} \lhd_v \mathsf{opt}\ v * v_{\texttt{size}} \lhd_v \texttt{num[size\_t]}\ n\ *$$
$$\textbf{if}\ \neg\neg(v = \mathsf{NULL})\ \textbf{then}\ \forall r, w.\ w \lhd_v \mathsf{own}\ r\ (\mathsf{any}\ n) * \mathsf{block}\ r\ n \mathbin{-\!\!*} \Phi\ w$$
$$\textbf{else}\ \exists \ell, v', m.\ v \lhd_v \mathsf{own}\ \ell\ (\mathsf{value}\ m\ v') * \mathsf{block}\ \ell\ m * m \leq n\ *$$
$$\begin{pmatrix}\forall w, r.\ w \lhd_v \mathsf{own}\ r\ (\mathsf{slices}\ n\ [(0, m, \mathsf{value}\ m\ v'); (m, n{-}m, \mathsf{any}\ (n{-}m))]) \mathbin{-\!\!*} \\ \mathsf{block}\ r\ n * v \lhd_v \mathsf{value}\ sz_{\texttt{ptr}}\ \ell \mathbin{-\!\!*} \Phi\ w\end{pmatrix}$$

xmemdup

```
void *xmemdup(const void *ptr, size_t size) {
    void *new_ptr = xmalloc(size);
    memcpy(new_ptr, ptr, size);
    return new_ptr;
}
```

$$T_{\texttt{xmemdup}}(v_{\texttt{ptr}}, v_{\texttt{size}})(\Phi) \triangleq$$
$$\exists n, \ell, v.\ v_{\texttt{size}} \lhd_v \texttt{num[size\_t]}\ n * v_{\texttt{ptr}} \lhd_v \mathsf{own}\ \ell\ (\mathsf{value}\ n\ v)\ *$$
$$(\forall w, r.\ \ell \lhd_l \mathsf{value}\ n\ v * \mathsf{block}\ r\ n * w \lhd_v \mathsf{own}\ r\ (\mathsf{value}\ n\ v) \mathbin{-\!\!*} \Phi\ w)$$

Fig. 5. Implementation and inferred specification of xmalloc, xrealloc, and xmemdup, where $sz_{\texttt{ptr}} \triangleq$ **sizeof**(**void***).

C ALLOCATORS

For the Allocators case study, we consider the functions xmalloc, xzalloc, zalloc, memdup, xrealloc, and xmemdup. Their implementations and inferred specifications can be found in Fig. 5 and Fig. 6. The "x" versions of the functions abort early if a memory allocation fails (*i.e.*, xmalloc, xzalloc, xrealloc, and xmemdup). When we compare the inferred specifications for them with their non-aborting

xzalloc

```c
void *xzalloc(size_t size) {
    void *ptr = xmalloc(size);
    memset(ptr, 0, size);
    return ptr;
}
```

$$T_{\texttt{xzalloc}}(v_{\texttt{size}})(\Phi) \triangleq$$
$$\exists n.\, (v_{\texttt{size}} \lhd_v \mathsf{num}[\texttt{size\_t}]\, n) * (\forall w, \ell.\, w \lhd_v \mathsf{own}\ \ell\ (\mathsf{zeros}\ n) * \mathsf{block}\ \ell\ n \mathrel{-\!\!*} \Phi\ w)$$

zalloc

```c
void *zalloc(size_t size) {
    void *ptr = malloc(size);
    if (ptr == NULL) {
        return NULL;
    }
    memset(ptr, 0, size);
    return ptr;
}
```

$$T_{\texttt{zalloc}}(v_{\texttt{size}})(\Phi) \triangleq$$
$$\exists n.\, (v_{\texttt{size}} \lhd_v \mathsf{num}[\texttt{size\_t}]\, n) * \begin{pmatrix} \forall w.\, w \lhd_v \mathsf{null} \mathrel{-\!\!*} \Phi\ w \\ \wedge \\ \forall w, \ell.\, w \lhd_v \mathsf{own}\ \ell\ (\mathsf{zeros}\ n) * \mathsf{block}\ \ell\ n \mathrel{-\!\!*} \Phi\ w \end{pmatrix}$$

memdup

```c
void *memdup(const void *ptr, size_t size) {
    void *new_ptr = malloc(size);
    if (new_ptr == NULL) {
        return NULL;
    }
    memcpy(new_ptr, ptr, size);
    return new_ptr;
}
```

$$T_{\texttt{memdup}}(v_{\texttt{ptr}}, v_{\texttt{size}})(\Phi) \triangleq$$
$$\exists n.\, (v_{\texttt{size}} \lhd_v \mathsf{num}[\texttt{size\_t}]\, n) *$$
$$\begin{pmatrix} \forall w.\, w \lhd_v \mathsf{null} \mathrel{-\!\!*} \Phi\ w \\ \wedge \\ \exists \ell, v.\, v_{\texttt{ptr}} \lhd_v \mathsf{own}\ \ell\ (\mathsf{value}\ n\ v) * \forall w, r.\, \ell \lhd_l \mathsf{value}\ n\ v * \mathsf{block}\ r\ n * w \lhd_v \mathsf{own}\ r\ (\mathsf{value}\ n\ v) \mathrel{-\!\!*} \Phi\ w \end{pmatrix}$$

Fig. 6. Implementation and inferred specification of xzalloc, zalloc, and memdup.

counterparts (*i.e.*, malloc, zalloc, realloc, and memdup), we can see that Quiver prunes away the branches where NULL would have been returned. The functions zalloc (a variant of malloc that initializes the memory with zeros) and memdup (a function that duplicates a pointer) are implemented using standard library functions.

Type Definition

```
[[q::refined_by(xs : list X)]]
[[q::typedef(list_t := optional (xs≠[]) (∃p,x,xr. own p ... ∗block p (sizeof(struct list))))]]
typedef struct list {
    [[q::field(x @T)]] void *head;
    [[q::field(xr @list_t)]] struct list *tail;
} *list_t;
```

$$xs\,@\,\texttt{list\_t} \equiv_{\text{ty}} \text{optional}\,(xs \neq [\,])\begin{pmatrix}\exists p, x, xr.\ \text{own}\,p\ (\text{struct}[\textbf{struct}\ \texttt{list}]\,[x\,@\,T; xr\,@\,\texttt{list\_t}]) \\ \quad * \text{block}\,p\ (\textbf{sizeof}(\textbf{struct}\ \texttt{list}))\end{pmatrix}$$

init

```
[[q::returns(? @list_t)]]
list_t init () { return NULL; }
```

$$T_{\texttt{init}}()(\Phi) \triangleq \forall w.\ w \triangleleft_v [\,]\,@\,\texttt{list\_t} \mathrel{-\!\!*} \Phi w$$

push

```
[[q::returns(? @list_t)]]
list_t push (list_t p, void *e) {
    struct list *node = xmalloc(sizeof(struct list));
    node->head = e; node->tail = p; return node;
}
```

$$T_{\text{push}}(v_{\text{p}}, v_{\text{e}})(\Phi) \triangleq \exists x, xr.\ (v_{\text{e}} \triangleleft_v x\,@\,T * v_{\text{p}} \triangleleft_v xr\,@\,\texttt{list\_t}) * \forall w.\ w \triangleleft_v (x :: xr)\,@\,\texttt{list\_t} \mathrel{-\!\!*} \Phi\, w$$

Fig. 7. Implementation and inferred specification of the Linked List (Part One).

D LINKED LIST

The implementation of the linked list is depicted in Fig. 7 and Fig. 8. Alongside it are the (Functional) specifications that Quiver infers for the operations and the (recursive) type definition. The are several things to note about the specifications that Quiver infers:

(1) For `init`, Quiver figures out that the list must be [], since only then can the optional in the definition of xs @ list_t be NULL. (An optional optional ϕ A is NULL when the condition ϕ is false and A when ϕ is true.)

(2) For push, Quiver infers the argument types without any sketches for them, knowing that the return value should be a list (*i.e.*, `[[q::returns(? @list_t)]]`). Thus, in this case, the same sketch as for `init` suffices to infer a very different specification—including that the resulting list is $x :: xr$.

(3) For pop, Quiver distinguishes between the case where the list is empty (and hence null is returned) and the nonempty case. In the latter case, we get back ownership of the head element of the list and the list itself (*i.e.*, stored in p) is updated to the tail.

(4) For is_empty the inferred specification returns a boolean whether the list is empty. (Like the one for pop, it could be further simplified by removing a double negation.)

(5) For `reverse`, Quiver infers the pre- and postcondition. The basis of this inference is the loop invariant in the middle of `reverse`: For p to be a _ @ list_t coming into the loop (*i.e.*, to prove the loop invariant), Quiver abducts that p should be a _ @ list_t already in the precondition. Moreover, after the loop, we know that p is null. Hence, zs is empty, and Quiver can infer that the returned list is the reverse of the input list.

pop

```
[[q::parameters(p : loc)]]
[[q::args(own p (? @ list_t))]]
[[q::ensures(p ◁ₗ ? @ list_t)]]
void *pop (list_t *p) {
  if (*p == NULL) { return NULL; }
  struct list *node = *p;
  void *ret = node->head;
  *p = node->tail;
  free(node);
  return ret;
}
```

$$
\begin{aligned}
T_{\texttt{pop}}(v_{\mathrm{p}})(\Phi) \triangleq\ & \exists p, xs.\ v_{\mathrm{p}} \triangleleft_v \text{own}\ p\ (xs \,@\, \text{list\_t}) \ast \\
& \textbf{if}\ \neg\neg(xs = [\,])\ \textbf{then}\ \forall w.\ (p \triangleleft_l [\,] \,@\, \text{list\_t}) \ast w \triangleleft_v \text{null} \mathrel{-\!\!\ast} \Phi w \\
& \textbf{else}\ \forall x, xr, w.\ (p \triangleleft_l xr \,@\, \text{list\_t}) \ast (w \triangleleft_v x \,@\, T) \ast xs = x :: xr \mathrel{-\!\!\ast} \Phi w
\end{aligned}
$$

is_empty

```
[[q::parameters(p : loc, xs : list ℤ)]]
[[q::args(own p (xs @ list_t))]]
[[q::ensures(p ◁ₗ xs @ list_t)]]
bool is_empty (list_t *l) {
    return *l == NULL;
}
```

$$
\begin{aligned}
T_{\texttt{is\_empty}}(v_{\mathrm{l}}, v_{\mathrm{e}})(\Phi) \triangleq\ & \exists p, xs.\ (v_{\mathrm{l}} \triangleleft_v \text{own}\ p\ (xs \,@\, \text{list\_t})) \ast \\
& \forall w.\ p \triangleleft_l xs \,@\, \text{list\_t} \ast w \triangleleft_v \text{bool[uchar]}\ \neg\neg(xs = [\,]) \mathrel{-\!\!\ast} \Phi\, w
\end{aligned}
$$

reverse

```
[[q::parameters(xs : list ℤ)]]
list_t reverse (list_t p) {
    list_t w, t;
    w = NULL;

    [[q::constraints(∃ys, zs. w ◁ₗ ys @ list_t ∗ p ◁ₗ zs @ list_t ∗ xs = rev ys ++ zs)]]
    while (p != NULL) {
        t = p->tail; p->tail = w; w = p; p = t;
    }
    return w;
}
```

$$
T_{\texttt{reverse}}(v_{\mathrm{p}})(\Phi) \triangleq \exists xs.\ v_{\mathrm{p}} \triangleleft_v xs \,@\, \text{list\_t} \ast \forall w.\ w \triangleleft_v (\text{rev}\ xs) \,@\, \text{list\_t} \mathrel{-\!\!\ast} \Phi\, w
$$

Fig. 8. Implementation and inferred specification of the Linked List (Part Two).

$$T_{\mathsf{mkvec}}(v_{\mathsf{n}})(\Phi) \triangleq$$
$$\exists n.\ (v_{\mathsf{n}} \lhd_v \mathtt{num[int]}\ n * n \geq 0) * (\forall w.\ w \lhd_v 0^n \,@\, \mathsf{vec\_t} \mathrel{-\!\!*} \Phi\, w)$$

$$T_{\mathsf{get\_unsafe}}(v_{\mathsf{vec}}, v_{\mathsf{i}}, v_{\mathsf{x}})(\Phi) \triangleq$$
$$\exists xs, i, \ell.\ (0 \leq i < \mathsf{len}\ xs * v_{\mathsf{vec}} \lhd_v xs \,@\, \mathsf{vec\_t} * v_{\mathsf{i}} \lhd_v \mathtt{num[int]}\ i * v_{\mathsf{x}} \lhd_v \mathsf{own}\ \ell\ (\mathsf{any}\ sz_{\mathtt{int}})) *$$
$$\forall w.\ \ell \lhd_l \mathtt{num[int]}\ (xs[i]) * v_{\mathsf{vec}} \lhd_v xs \,@\, \mathsf{vec\_t} * w \lhd_v \mathsf{void} \mathrel{-\!\!*} \Phi\, w$$

$$T_{\mathsf{set\_unsafe}}(v_{\mathsf{vec}}, v_{\mathsf{i}}, v_{\mathsf{x}})(\Phi) \triangleq$$
$$\exists xs, \ell, i, n.\ (0 \leq i < \mathsf{len}\ xs * v_{\mathsf{vec}} \lhd_v xs \,@\, \mathsf{vec\_t} * v_{\mathsf{i}} \lhd_v \mathtt{num[int]}\ i * v_{\mathsf{x}} \lhd_v \mathsf{own}\ \ell\ (\mathtt{num[int]}\ n)) *$$
$$\forall w.\ \ell \lhd_l \mathtt{num[int]}\ n * v_{\mathsf{vec}} \lhd_v (xs[i := n]) \,@\, \mathsf{vec\_t} * w \lhd_v \mathsf{void} \mathrel{-\!\!*} \Phi\, w$$

$$T_{\mathsf{get\_checked}}(v_{\mathsf{vec}}, v_{\mathsf{i}})(\Phi) \triangleq$$
$$\exists xs, i.\ (v_{\mathsf{vec}} \lhd_v xs \,@\, \mathsf{vec\_t} * v_{\mathsf{i}} \lhd_v \mathtt{num[int]}\ i) *$$
$$\textbf{if}\ (i < 0 \lor i \geq \mathsf{len}\ xs)\ \textbf{then}\ \forall w.\ v_{\mathsf{vec}} \lhd_v xs \,@\, \mathsf{vec\_t} * w \lhd_v \mathtt{num[int]}\ (-1) \mathrel{-\!\!*} \Phi\, w$$
$$\textbf{else}\ \forall w.\ v_{\mathsf{vec}} \lhd_v xs \,@\, \mathsf{vec\_t} * w \lhd_v \mathtt{num[int]}\ (xs[i]) \mathrel{-\!\!*} \Phi\, w$$

$$T_{\mathsf{set\_checked}}(v_{\mathsf{vec}}, v_{\mathsf{i}})(\Phi) \triangleq$$
$$\exists xs, i.\ (v_{\mathsf{vec}} \lhd_v xs \,@\, \mathsf{vec\_t} * v_{\mathsf{i}} \lhd_v \mathtt{num[int]}\ i) *$$
$$\textbf{if}\ (i < 0 \lor i \geq \mathsf{len}\ xs)\ \textbf{then}\ \forall w.\ v_{\mathsf{vec}} \lhd_v xs \,@\, \mathsf{vec\_t} * w \lhd_v \mathsf{void} \mathrel{-\!\!*} \Phi\, w$$
$$\textbf{else}\ \exists n.\ v_{\mathsf{val}} \lhd_v \mathtt{num[int]}\ n * \forall w.\ v_{\mathsf{vec}} \lhd_v (xs[i := n]) \,@\, \mathsf{vec\_t} * w \lhd_v \mathsf{void} \mathrel{-\!\!*} \Phi\, w$$

Fig. 9. Vector Specifications, Part One

E VECTOR

For the Vector case study, we consider the operations `mkvec`, `get_unsafe`, `set_unsafe`, `vec_grow`, `get_checked`, `set_checked`, `swap`, and `vec_free`. We give the inferred vector specifications in Fig. 9 and Fig. 10. Their implementation is depicted in Fig. 11 and Fig. 12.

In the accompanying Coq development, we additionally consider the following two variations of the vector operations:

(1) `grow_no_branch`, which uses a sketch to eliminate the branching that otherwise arises in the specification of `grow` (compare $T_{\mathtt{grow}}$).
(2) `get_checked_joined` and `set_checked_joined`, which are versions of `get_checked` and `set_checked` that use Thorium's heuristic to join the branches in the implementation.

$$
\begin{aligned}
&T_{\mathsf{swap}}(v_{\mathsf{vec}}, v_{\mathsf{i}}, v_{\mathsf{j}})(\Phi) \triangleq \\
&\quad \exists xs, j.\ (v_{\mathsf{vec}} \triangleleft_v xs\,@\,\mathsf{vec\_t} * v_{\mathsf{j}} \triangleleft_v \mathtt{num[int]}\ j) * \\
&\quad \textbf{if } (j < 0 \lor j \geq \mathsf{len}\ xs) \textbf{ then } \forall w.\ v_{\mathsf{vec}} \triangleleft_v xs\,@\,\mathsf{vec\_t} * w \triangleleft_v \mathsf{void} \mathrel{-\!\!*} \Phi\, w \\
&\quad \textbf{else } \exists i.\ v_{\mathsf{i}} \triangleleft_v \mathtt{num[int]}\ i * \textbf{if } (i < 0 \lor i \geq \mathsf{len}\ xs) \textbf{ then } \forall w.\ v_{\mathsf{vec}} \triangleleft_v xs\,@\,\mathsf{vec\_t} * w \triangleleft_v \mathsf{void} \mathrel{-\!\!*} \Phi\, w \\
&\qquad\qquad\qquad\qquad\qquad\qquad \textbf{else } \forall w.\ v_{\mathsf{vec}} \triangleleft_v xs[i := xs[j], j := xs[i]]\,@\,\mathsf{vec\_t} * w \triangleleft_v \mathsf{void} \mathrel{-\!\!*} \Phi\, w
\end{aligned}
$$

$$
\begin{aligned}
&T_{\mathsf{grow}}(v_{\mathsf{vec}}, v_{\mathsf{new}})(\Phi) \triangleq \\
&\quad \exists xs, n.\ (v_{\mathsf{vec}} \triangleleft_v xs\,@\,\mathsf{vec\_t} * v_{\mathsf{new}} \triangleleft_v \mathtt{num[int]}\ n) * \\
&\quad \textbf{if } n \leq \mathsf{len}\ xs \textbf{ then } \forall w.\ v_{\mathsf{vec}} \triangleleft_v xs\,@\,\mathsf{vec\_t} * w \triangleleft_v \mathtt{num[int]}\ (\mathsf{len}\ xs) \mathrel{-\!\!*} \Phi\, w \\
&\quad \textbf{else } \forall w.\ v_{\mathsf{vec}} \triangleleft_v (xs \mathbin{+\!\!+} 0^{n - \mathsf{len}\ xs})\,@\,\mathsf{vec\_t} * w \triangleleft_v \mathtt{num[int]}\ n \mathrel{-\!\!*} \Phi\, w
\end{aligned}
$$

$$
T_{\mathsf{free}}(v_{\mathsf{vec}})(\Phi) \triangleq \exists xs.\ v_{\mathsf{vec}} \triangleleft_v xs\,@\,\mathsf{vec\_t} * \forall w, \ell.\ v_{\mathsf{vec}} \triangleleft_v \mathsf{value}\ sz_{\mathsf{ptr}}\ \ell * w \triangleleft_v \mathsf{void} \mathrel{-\!\!*} \Phi\, w
$$

$$
\begin{aligned}
&T_{\mathsf{grow\_no\_branch}}(v_{\mathsf{vec}}, v_{\mathsf{new}})(\Phi) \triangleq \\
&\quad \exists xs, n.\ (v_{\mathsf{vec}} \triangleleft_v xs\,@\,\mathsf{vec\_t} * v_{\mathsf{new}} \triangleleft_v \mathtt{num[int]}\ n * n > \mathsf{len}\ xs) * \\
&\quad \forall w.\ v_{\mathsf{vec}} \triangleleft_v (xs \mathbin{+\!\!+} 0^{n - \mathsf{len}\ xs})\,@\,\mathsf{vec\_t} * w \triangleleft_v \mathtt{num[int]}\ n \mathrel{-\!\!*} \Phi\, w
\end{aligned}
$$

$$
\begin{aligned}
&T_{\mathsf{get\_checked\_joined}}(v_{\mathsf{vec}}, v_{\mathsf{i}})(\Phi) \triangleq \\
&\quad \exists xs, i.\ (v_{\mathsf{vec}} \triangleleft_v xs\,@\,\mathsf{vec\_t} * v_{\mathsf{i}} \triangleleft_v \mathtt{num[int]}\ i) * \\
&\quad \forall w.\ v_{\mathsf{vec}} \triangleleft_v xs\,@\,\mathsf{vec\_t} * w \triangleleft_v \mathtt{num[int]}\ (\mathsf{if}\ (i < 0 \lor i \geq \mathsf{len}\ xs)\ \mathsf{then}\ -1\ \mathsf{else}\ xs[i]) \mathrel{-\!\!*} \Phi\, w
\end{aligned}
$$

$$
\begin{aligned}
&T_{\mathsf{set\_checked\_joined}}(v_{\mathsf{vec}}, v_{\mathsf{i}}, v_{\mathsf{v}})(\Phi) \triangleq \\
&\quad \exists xs, i, n.\ v_{\mathsf{vec}} \triangleleft_v xs\,@\,\mathsf{vec\_t} * v_{\mathsf{i}} \triangleleft_v \mathtt{num[int]}\ i * v_{\mathsf{v}} \triangleleft_v \mathtt{num[int]}\ n * \\
&\quad \forall w.\ v_{\mathsf{vec}} \triangleleft_v (\mathsf{if}\ (i < 0 \lor i \geq \mathsf{len}\ xs)\ \mathsf{then}\ xs\ \mathsf{else}\ xs[i := n])\,@\,\mathsf{vec\_t} * w \triangleleft_v \mathsf{void} \mathrel{-\!\!*} \Phi\, w
\end{aligned}
$$

Fig. 10. Vector Specifications, Part Two, where $sz_{\mathsf{ptr}} \triangleq$ **`sizeof`**(**`void*`**)

```
[[q::refined_by(xs : list Z)]]
[[q::typedef(vec_t := ∃p. own p ... ∗ block p (sizeof(struct vector)))]]
typedef struct vector {
  [[q::field(∃q. own q (array[num[int]] xs) ∗ block q (sizeof(int) · len xs))]] int *data;
  [[q::field(num[int] (len xs))]] int len;
} *vec_t;

vec_t mkvec(int n) {
  size_t s=sizeof(int)*(size_t)n;
  vec_t vec=xmalloc(sizeof(*vec));
  vec->data=xzalloc(s);
  vec->len=n;
  [[q::type(? @ vec_t)]] return vec;
}

[[q::requires(ˆvec ◁v ? @ vec_t)]]
[[q::ensures(ˆvec ◁v ? @ vec_t)]]
void get_unsafe(vec_t vec, int i, int *x) {
  *x = vec->data[i];
}

[[q::requires(ˆvec ◁v ? @ vec_t)]]
[[q::ensures(ˆvec ◁v ? @ vec_t)]]
void set_unsafe(vec_t vec, int i, int *x) {
  vec->data[i] = *x;
}

[[q::requires(ˆvec ◁v ? @ vec_t)]]
[[q::ensures(ˆvec ◁v ? @ vec_t)]]
int get_checked(vec_t vec, int i){
  assert (vec->len >= 0);
  if (i < 0 || i >= vec->len) {
    return -1;
  }
  return vec->data[i];
}

[[q::requires(ˆvec ◁v ? @ vec_t)]]
[[q::ensures(ˆvec ◁v ? @ vec_t)]]
void set_checked(vec_t vec, int i, int v) {
  assert (vec->len >= 0);
  if (i < 0 || i >= vec->len) {
    return;
  }
  vec->data[i] = v;
}
```

Fig. 11. Vector Implementation, Part One

```c
[[q::requires(^vec ◁_v ? @ vec_t)]]
[[q::ensures(^vec ◁_v ? @ vec_t)]]
int vec_grow(vec_t vec, int new_size) {
  if (vec == NULL) {
    return 0;
  }
  if (new_size <= vec->len) {
    return vec->len;
  }
  int *buf = xmalloc(sizeof(int) * new_size);
  memcpy(buf, vec->data, sizeof(int) * vec->len);
  free(vec->data);
  vec->data = buf;
  memset(&(vec->data[vec->len]), 0, sizeof(int) * (new_size - vec->len));
  vec->len = new_size;
  return vec->len;
}

[[q::requires(^vec ◁_v ? @ vec_t)]]
[[q::ensures(^vec ◁_v ? @ vec_t)]]
void swap(vec_t vec, int i, int j) {
  if (j < 0 || j >= vec->len){
    return;
  }
  if (i >= vec->len || i < 0) {
    return;
  }
  int tmp = vec->data[i];
  vec->data[i] = vec->data[j];
  vec->data[j] = tmp;
}

[[q::requires(^vec ◁_v ? @ vec_t)]]
void vec_free(vec_t vec) {
  free(vec->data);
  free(vec);
}
```

Fig. 12. Vector Implementation, Part Two

F MEMCACHED BIPBUFFER

In the Bipbuffer case study from the open-source library memcached [3], we infer specifications for the following functions: `bipbuf_size`, `bipbuf_size_corrected` (not counted in the paper), `bipbuf_sizeof`, `bipbuf_unused`, `bipbuf_used`, `bipbuf_init`, `bipbuf_new`, `bipbuf_free`, `bipbuf_is_empty`, `__check_for_switch_to_b`, `bipbuf_request`, `bipbuf_push`, `bipbuf_offer`, `bipbuf_peek`, `bipbuf_peek_all`, `bipbuf_poll`.

The source code and the inferred specifications of these functions can be found in the accompanying Coq development. To give the reader an impression of the Bipbuffer, we have included the functions `bipbuf_size`, `bipbuf_size_corrected`, and `bipbuf_peek_all` (in Fig. 13).

For the specification of the Bipbuffer, we use the following type definition:

```
[[q::refined_by(len : ℤ, a_start: ℤ, a_end: ℤ, b_end: ℤ, b_inuse: ℤ)]]
[[q::typedef(bipbuf_t := slices (sz_buf + len) [(0, sz_buf, ...); (sz_buf, len, any len)])]]
[[q::constraints(0 ≤ b_end ∧ b_end ≤ a_start ∧ a_start ≤ a_end ∧ a_end ≤ len)]]
typedef struct buf
{
    [[q::field(num[unsigned long int] len)]] unsigned long int size;
    /* region A */
    [[q::field(num[unsigned int] a_start)]] unsigned int a_start;
    [[q::field(num[unsigned int] a_end)]] unsigned int a_end;
    /* region B */
    [[q::field(num[unsigned int] b_end)]] unsigned int b_end;
    /* is B inuse? */
    [[q::field(num[int] b_inuse)]] int b_inuse;
    unsigned char data[];

} bipbuf_t;
```

where $sz_{\text{buf}} \triangleq$ **`sizeof`**`(`**`struct`** `buf)`. The Bipbuffer consists of a size field `size`, a region A with fields `a_start` and `a_end`, a region B with field `b_end`, and a flag whether region B is in use, `b_inuse`. Furthermore, the bipbuffer has a `data` field, which stores the contents of the buffer. In our type _ @ buf_t, we track the fields `size`, `a_start`, `a_end`, `b_end`, and `b_inuse`. Moreover, we track the length of the bytes in the `data` field (but not their contents) with the type any `len` in the definition; it represents n arbitrary bytes that are appended after the **`struct`** `buf` itself. Finally, we maintain a data type invariant that the buffer fields are ordered in the sense that

$$0 \leq \texttt{b\_end} \wedge \texttt{b\_end} \leq \texttt{a\_start} \wedge \texttt{a\_start} \leq \texttt{a\_end} \wedge \texttt{a\_end} \leq \texttt{len}$$

Inference The functions `bipbuf_size` and `bipbuf_size_corrected` are accessor functions for the `size` field (*i.e.*, they return the value of the `size` field). The function `bipbuf_size` is part of the original implementation of the Bipbuffer. It contains the type mismatch mentioned in the paper. This type mismatch is revealed in the inferred predicate transformer $T_{\texttt{bipbuf\_size}}$ (in Fig. 14) through the precondition `len ∈ int` (*i.e.*, the length field should store a valid `int`-integer). If we change the return type to **`unsigned long int`**, as is the type of `size`, the constraint disappears (see $T_{\texttt{bipbuf\_size\_corrected}}$).

We additionally showcase the inferred specification of `bipbuf_peek_all`, one of the other buffer functions. What is interesting about it is that for `bipbuf_peek_all` it suffices to give only a postcondition sketch (see Fig. 13). The reason is that `bipbuf_peek_all` calls an auxiliary function, `bipbuf_is_empty`, in the beginning whose specification constraints $v_{\texttt{me}}$ to be Bipbuffer. Thus, Quiver takes this constraint into account and infers a _ @ buf_t precondition for `bipbuf_peek_all`.

```
[[q::args(own ? (? @ bipbuf_t))]]
[[q::ensures(^me ◁ᵥ own ? (? @ bipbuf_t))]]
int bipbuf_size(const bipbuf_t* me)
{
    return me->size;
}

[[q::args(own ? (? @ bipbuf_t))]]
[[q::ensures(^me ◁ᵥ own ? (? @ bipbuf_t))]]
unsigned long int bipbuf_size_corrected(const bipbuf_t* me)
{
    return me->size;
}

[[q::ensures(^me ◁ᵥ own ? (? @ bipbuf_t))]]
unsigned char *bipbuf_peek_all(const bipbuf_t* me, unsigned int *size)
{
    if (bipbuf_is_empty(me))
        return NULL;

    *size = me->a_end - me->a_start;
    return (unsigned char*)__data(me) + me->a_start;
}
```

Fig. 13. Implementation of bipbuf_size, bipbuf_size_corrected, and bipbuf_peek_all, where $sz_{\texttt{buf}} \triangleq$ **sizeof**(**struct** buf).

$$
\begin{aligned}
&\mathcal{T}_{\texttt{bipbuf\_size}}(v_{\texttt{me}})(\Phi) \triangleq \\
&\quad \exists \ell, \texttt{len}, \texttt{a\_start}, \texttt{a\_end}, \texttt{b\_end}, \texttt{b\_inuse}. \\
&\quad \texttt{len} \in \texttt{int} * v_{\texttt{me}} \lhd_v \text{own } \ell\ (\texttt{len}, \texttt{a\_start}, \texttt{a\_end}, \texttt{b\_end}, \texttt{b\_inuse}) \mathbin{@} \text{bipbuf\_t} * \\
&\quad \forall w.\ v_{\texttt{me}} \lhd_v \text{own } \ell\ (\texttt{len}, \texttt{a\_start}, \texttt{a\_end}, \texttt{b\_end}, \texttt{b\_inuse}) \mathbin{@} \text{bipbuf\_t} \mathrel{-\!\!*} \\
&\qquad w \lhd_v \texttt{num[int]}\ \texttt{len} \mathrel{-\!\!*} \Phi\, w \\
&\mathcal{T}_{\texttt{bipbuf\_size\_corrected}}(v_{\texttt{me}})(\Phi) \triangleq \\
&\quad \exists \ell, \texttt{len}, \texttt{a\_start}, \texttt{a\_end}, \texttt{b\_end}, \texttt{b\_inuse}. \\
&\quad v_{\texttt{me}} \lhd_v \text{own } \ell\ (\texttt{len}, \texttt{a\_start}, \texttt{a\_end}, \texttt{b\_end}, \texttt{b\_inuse}) \mathbin{@} \text{bipbuf\_t} * \\
&\quad \forall w.\ v_{\texttt{me}} \lhd_v \text{own } \ell\ (\texttt{len}, \texttt{a\_start}, \texttt{a\_end}, \texttt{b\_end}, \texttt{b\_inuse}) \mathbin{@} \text{bipbuf\_t} \mathrel{-\!\!*} \\
&\qquad w \lhd_v \texttt{num[unsigned long int]}\ \texttt{len} \mathrel{-\!\!*} \Phi\, w
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{T}_{\texttt{bipbuf\_peek\_all}}(v_{\texttt{me}})(\Phi) \triangleq \\
&\quad \exists \ell, \texttt{len}, \texttt{a\_start}, \texttt{a\_end}, \texttt{b\_end}, \texttt{b\_inuse}. \\
&\quad v_{\texttt{me}} \lhd_v \text{own } \ell\ (\texttt{len}, \texttt{a\_start}, \texttt{a\_end}, \texttt{b\_end}, \texttt{b\_inuse}) \mathbin{@} \text{bipbuf\_t} * \\
&\quad \textbf{if } \texttt{a\_start} = \texttt{a\_end} \textbf{ then} \\
&\qquad \forall w.\ v_{\texttt{me}} \lhd_v \text{own } \ell\ (\texttt{len}, \texttt{a\_start}, \texttt{a\_end}, \texttt{b\_end}, \texttt{b\_inuse}) \mathbin{@} \text{bipbuf\_t} * w \lhd_v \text{null} \mathrel{-\!\!*} \Phi w \\
&\quad \textbf{else} \\
&\qquad \exists \ell.\ v_{\texttt{size}} \lhd_v \text{own } \ell\ (\text{any } sz_{\texttt{uint}}) * \\
&\qquad \forall w.\ \ell \lhd_l \texttt{num[unsigned int]}\ (\texttt{a\_end} - \texttt{a\_start}) \mathrel{-\!\!*} \\
&\qquad\quad w \lhd_v \text{value } sz_{\texttt{ptr}}\ (\ell + sz_{\texttt{uchar}} * sz_{\texttt{buf}} + sz_{\texttt{uchar}} * \texttt{a\_start}) \mathrel{-\!\!*} \\
&\qquad\quad v_{\texttt{me}} \lhd_v \text{own } \ell\ (\texttt{len}, \texttt{a\_start}, \texttt{a\_end}, \texttt{b\_end}, \texttt{b\_inuse}) \mathbin{@} \text{bipbuf\_t} \mathrel{-\!\!*} \Phi w
\end{aligned}
$$

Fig. 14. Inferred predicate transformers for bipbuf_size, bipbuf_size_corrected, and bipbuf_peek_all, where we abbreviate $sz_{\texttt{uchar}} \triangleq$ **sizeof**(**unsigned char**), $sz_{\texttt{uint}} \triangleq$ **sizeof**(**unsigned int**), $sz_{\texttt{buf}} \triangleq$ **sizeof**(**struct buf**), and $sz_{\texttt{ptr}} \triangleq$ **sizeof**(**void***).

G OPENSSL BUFFER

For the OpenSSL Buffer [4] case study, we verify the functions `BUF_MEM_new_ex`, `BUF_MEM_new`, `BUF_MEM_free`, `sec_alloc_realloc`, `BUF_MEM_grow`, and `BUF_MEM_grow_clean` (for both versions). The source code and the inferred specifications of these functions can be found in the accompanying Coq development. To give the reader an impression of the Buffer, we have included the function `BUF_mem_grow_clean` and its helper function `sec_alloc_realloc` (in Fig. 15).

The implementation contains the sketches that we provide in the *length* version for both functions. For the length specification of the Buffer, we use the following type definition:

```
 1 [[q::refined_by(len : ℤ, cap : ℤ, sec : 𝔹)]]
 2 [[q::exists(b : loc)]]
 3 [[q::typedef(buf_t := own b ...)]]
 4 [[q::constraints(block b (sizeof(struct struct_buf_mem_st)))]]
 5 [[q::constraints(len ≤ cap)]]
 6 struct buf_mem_st {
 7     [[q::field(num[size_t] len)]]
 8     size_t length;            // current number of bytes
 9     [[q::field(optional (cap > 0) (∃q. own q (any cap) ∗ block q cap))]]
10     char *data;
11     [[q::field(num[size_t] cap)]]
12     size_t max;               // size of buffer
13     [[q::field(bool[unsigned long] sec)]]
14     unsigned long flags;
15 };
```

The type (len, cap, sec) @ buf_t is an owned pointer (with a block predicate) to the **struct** `buf_mem_st` tracking the buffer contents. Concretely, the struct has a length field `length` tracking the length of the buffer, a `cap` field tracking the capacity of the buffer (*i.e.*, the size of the allocated memory), a `data` field tracking the contents of the buffer, and a `flags` field, which controls how the buffer is reallocated (explained below). The field `data` is an optional, owned pointer to *cap* bytes. We are not tracking the contents of the pointer and, hence, it can store arbitrary bytes (for which we use any *cap*). (When the buffer is empty, the pointer is `NULL`.) As a data type invariant, this buffer ensures that the length *len* does not exceed the capacity *cap* (Line 5).

For the function `BUF_mem_grow_clean` (from Fig. 15), the inferred specification can be found in Fig. 16. Its precondition is that $v_{\mathtt{str}}$ is a buffer $(len, cap, \mathrm{true})$ @ buf_t where the secure flag is set to true, and that $v_{\mathtt{len}}$ is an integer $v_{\mathtt{len}} \triangleleft_v \mathtt{num[size\_t]}\ new$. It has four possible postconditions:

(1) If (a) the current length exceeds the new size or (b) the current capacity is still sufficient for the new length, the function updates the length to *new* (and giving back its ownership) and returns the length *new* as the new length (Line 24 and Line 29).
(2) If the new length *new* exceeds `LIMIT_BEFORE_EXPANSION` (*i.e.*, 1610612732), the buffer is left unchanged and the length 0 is returned (Line 34).
(3) If the new length is below `LIMIT_BEFORE_EXPANSION`, the reallocation can fail, leading to the return value 0 and an unchanged buffer.
(4) Alternatively, the reallocation can succeed, leading to a buffer with length *len* and capacity $(new + 3)/3 \cdot 4$. The return value in this case is *new*.

There are several things to note about this inference:

```
[[q::args(? @ buf_t, _)]]
[[q::join_conj]]
static char *sec_alloc_realloc(BUF_MEM *str, size_t len) {
    char *ret;
    ret = OPENSSL_secure_malloc(len);
    if (str->data != NULL) {
        if (ret != NULL) {
            memcpy(ret, str->data, str->length);
            OPENSSL_secure_clear_free(str->data, str->length);
            str->data = NULL;
            [[q::assert(ret ◁_l own ? (any ?))]];
        }
    }
    return ret; }
[[q::args((?, ?, true) @ buf_t, _)]]
[[q::ensures(^str ◁_v ? @ buf_t)]]
[[q::join_if]]
size_t BUF_MEM_grow_clean(BUF_MEM *str, size_t len) {
    char *ret; size_t n;
    if (str->length >= len) {
        if (str->data != NULL)
            memset(&str->data[len], 0, str->length - len);
        str->length = len;
        return len;
    }
    if (str->max >= len) {
        memset(&str->data[str->length], 0, len - str->length);
        str->length = len;
        return len;
    }
    /* This limit is sufficient to ensure (len+3)/3*4 < 2**31 */
    if (len > LIMIT_BEFORE_EXPANSION) {
        ERR_raise(ERR_LIB_BUF, ERR_R_PASSED_INVALID_ARGUMENT);
        return 0;
    }
    n = (len + 3) / 3 * 4;
    if ((str->flags & BUF_MEM_FLAG_SECURE))
        ret = sec_alloc_realloc(str, n);
    else
        ret = OPENSSL_clear_realloc(str->data, str->max, n);
    if (ret == NULL) {
        len = 0;
    } else {
        str->data = ret;
        str->max = n;
        memset(&str->data[str->length], 0, len - str->length);
        str->length = len;
    }
    return len;
}
```

Fig. 15. BUF_MEM_grow_clean from the OpenSSL Buffer

Inferred Specification

$$
\begin{array}{l}
T_{\texttt{BUF\_MEM\_grow\_clean}}(v_{\texttt{str}}, v_{\texttt{len}})(\Phi) \triangleq \exists \mathit{len}, \mathit{cap}, \mathit{new}.\ v_{\texttt{str}} \lhd_v (\mathit{len}, \mathit{cap}, \mathsf{true}) \mathbin{@} \mathsf{buf\_t} * v_{\texttt{len}} \lhd_v \texttt{num[size\_t]}\ \mathit{new} * \\
\quad \textbf{if } (\mathit{len} \geq \mathit{new}) \lor (\mathit{cap} \geq \mathit{new}) \textbf{ then } \forall w.\ v_{\texttt{str}} \lhd_v (\mathit{new}, \mathit{cap}, \mathsf{true}) \mathbin{@} \mathsf{buf\_t} * w \lhd_v \texttt{num[size\_t]}\ \mathit{new} \mathrel{-\!\!*} \Phi\, w \\
\quad \textbf{else if } \mathit{new} > 1610612732 \textbf{ then } \forall w.\ v_{\texttt{str}} \lhd_v (\mathit{len}, \mathit{cap}, \mathsf{true}) \mathbin{@} \mathsf{buf\_t} * w \lhd_v \texttt{num[size\_t]}\ 0 \mathrel{-\!\!*} \Phi\, w \\
\quad \textbf{else} \left(\begin{array}{l}
\forall \ell, r, w.\ \ell \lhd_v \mathsf{value}\ \mathit{sz}_{\texttt{ptr}}\ r * v_{\texttt{str}} \lhd_v (\mathit{len}, \mathit{cap}, \mathsf{true}) \mathbin{@} \mathsf{buf\_t} * w \lhd_v \texttt{num[size\_t]}\ 0 \mathrel{-\!\!*} \Phi\, w \\
\qquad\qquad\qquad\qquad\qquad \land \\
\forall w.\ \ell \lhd_v \mathsf{value}\ \mathit{sz}_{\texttt{ptr}}\ r * v_{\texttt{str}} \lhd_v (\mathit{new}, (\mathit{new}+3)/3 \cdot 4, \mathsf{true}) \mathbin{@} \mathsf{buf\_t} * w \lhd_v \texttt{num[size\_t]}\ \mathit{new} \mathrel{-\!\!*} \Phi\, w
\end{array} \right)
\end{array}
$$

Fig. 16. Inferred specification for BUF_MEM_grow_clean from the OpenSSL Buffer, where we abbreviate $\mathit{sz}_{\texttt{ptr}} \triangleq$ **sizeof**(**void***).

(1) As mentioned in the paper, the reallocation operation does an overflow check, namely `len > LIMIT_BEFORE_EXPANSION`. Quiver figures out that if the overflow check passes, meaning `len <= LIMIT_BEFORE_EXPANSION`, then the subsequent expansion `n = (len + 3) / 3 * 4` (Line 36) does not overflow.
(2) As we can see by comparing the sketches for BUF_MEM_grow_clean and the inferred specification, Quiver figures out the various postconditions *and* what the values of the buffer fields are in each of the cases. Moreover, it infers a type for the argument $v_{\texttt{len}}$, for which we provide no sketch.
(3) As we can see by comparing the specification and the implementation, Quiver provides abstraction over the implementation. More specifically, the inferred specification describes how the buffer changes, but without dropping to the level of pointer manipulations.
(4) For the inference of BUF_MEM_grow_clean, Quiver reuses the specification that it infers for sec_alloc_realloc. The inferred specification of sec_alloc_realloc can be found in the accompanying Coq development.
(5) The inferred specification reduces the amount of branching in the function (and also the branching inherited from sec_alloc_realloc). The first two conditionals are turned into a single case in the resulting specification. Moreover, we provide as an outline that the flag `flag` should be true. Quiver takes this into account, adds it as a precondition that the flag should be true, and prunes away the branch that uses OPENSSL_clear_realloc.

(The annotation "**[[q::join_if]]**" activates the Thorium heuristic for joining conditionals, and the annotation "**[[q::join_conj]]**" on sec_alloc_realloc distributes conjunctions over conditionals.)

It is important to note that we do not verify the memory management of OpenSSL and, in particular, not the secure memory management that is behind OPENSSL_secure_malloc. Instead, in this case study, we assume the OpenSSL functions OPENSSL_malloc, OPENSSL_zalloc, OPENSSL_memdup, OPENSSL_free, and OPENSSL_realloc have the corresponding standard library specifications from Fig. 4 (or the inferred specification from Fig. 5). For the nonstandard functions OPENSSL_clear_realloc and OPENSSL_clear_free, we provide implementations copied from OpenSSL for which we infer the specification (based on the others). (We do not count them as part of the OpenSSL Buffer case study.) For the function OPENSSL_secure_malloc, we assume its specification corresponds to the standard library function malloc, and for the function OPENSSL_secure_clear_free, we use the specification inferred for OPENSSL_clear_free.

H BINARY SEARCH

For the Binary Search case study, we infer the specification of the binary search implementation:

```
[[q::parameters(xs: list ℤ, k: ℤ)]]
[[q::args(xs @ sorted_array, num[size_t] (len xs), num[int] k)]]
size_t bin_search(int *a, size_t n, int x) {
    size_t l = 0;
    size_t r = n;

    [[q::constraints(∃(ab : ℤ). l ◁ₗ num[size_t] a ∗ r ◁ₗ num[size_t] b)]]
                          ∗ vₐ ◁ᵥ xs @ sorted_array ∗ a ≤ b ∗ b ≤ len xs
                          ∗ in_between xs k a b
    while (l < r) {
        size_t m = l + (r - l) / 2;
        if (a[m] < x) {
            l = m + 1;
        } else {
            r = m;
        }
    }
    return l;
}
```

where we define the type of sorted arrays as

$$xs \,@\, \text{sorted\_array} \equiv_{\text{ty}} \exists p.\ \text{own}\ p\ (\texttt{array[num[int]]}\ xs) * \text{block}\ \ell\ (\text{len}\ xs \cdot sz_{\texttt{int}}) * \text{sorted}\ xs$$

and the predicate for the loop invariant, `in_between`, as

$$\begin{aligned}\texttt{in\_between}\ xs\ x\ l\ r \triangleq\ & (\forall a i.\ i < l \rightarrow xs[i] = \mathsf{Some}\ a \rightarrow a < x)\\ & \land (\forall a i.\ r \leq i \rightarrow xs[i] = \mathsf{Some}\ a \rightarrow x \leq a)\end{aligned}$$

Inference. The predicate transformer specification that Quiver infers is:

$$\begin{aligned}T_{\texttt{bin\_search}}(v_{\mathsf{a}}, v_{\mathsf{n}}, v_{\mathsf{x}}) \triangleq\ & \exists xs, k.\ v_{\mathsf{a}} \triangleleft_v xs \,@\, \text{sorted\_array} * v_{\mathsf{n}} \triangleleft_v \texttt{num[size\_t]}\ (\text{len}\ xs) * v_{\mathsf{x}} \triangleleft_v \texttt{num[int]}\ k\ *\\ & \quad \forall m, w.\ 0 \leq x \leq \text{len}\ x * \texttt{in\_between}\ xs\ k\ m\ m \twoheadrightarrow\\ & \qquad v_{\mathsf{a}} \triangleleft_v xs \,@\, \text{sorted\_array} * w \triangleleft_v \texttt{num[size\_t]}\ m \twoheadrightarrow \Phi w\end{aligned}$$

Note that the sketch for the binary search implementation does not detail a postcondition. It only sketches the loop invariant and the precondition. Nevertheless, based on the invariant and knowing that after the loop $\neg(\texttt{l} < \texttt{r})$, Quiver infers the postcondition, namely that the resulting index m is in between all elements smaller than k and those larger or equal to k.

I HASHMAP

For the Hashmap case study, we verify the functions `fsm_realloc_if_necessary`, `compute_min_count`, `fsm_remove`, `fsm_get`, `fsm_insert`, `fsm_init`, `fsm_probe`, and `fsm_slot_for_key`. The source code and the inferred specifications of these functions can be found in the accompanying Coq development. To give the reader an impression of the Hashmap, we have included the functions `fsm_init` and `fsm_realloc_if_necessary` (in Fig. 18).

For the specification of the Hashmap, we use the following type definition:

```
[[q::typedef(x := item_ref)]]
[[q::exists(tag: ℤ, key: ℤ, val: ℤ)]]
[[q::constraints(build_item tag key val = Some x)]]
struct item {
  [[q::field(num[size_t] tag)]] size_t tag;
  [[q::field(num[size_t] key)]] size_t key;
  [[q::field(num[size_t] val)]] size_t value;
};

[[q::typedef(mp := gmap ℤ ℤ, items : list item_ref, count : ℤ)]]
[[q::constraints(fsm_invariant mp items count)]]
struct fixed_size_map {
  [[q::field(∃p. own p (array[λx.x @ item] items) ∗ block p (sizeof(struct item) ∗ len items))]]
  struct item (* items)[];
  [[q::field(num[size_t] count)]]
  size_t count;
  [[q::field(num[size_t] (len items))]]
  size_t length;
};
```

The Hashmap tracks finite, mathematical maps "`gmap ℤ ℤ`" from integers to integers (finite Coq maps). The type definition consists of two parts: a type of map items **struct** `item` and the actual map **struct** `fixed_size_map`. The items consist of a `tag`, `key`, and `value`. The tag can be `Empty` for an empty entry, `Entry` for an entry that contains a key `key` and value `val`, and `Tombstone` for an entry that has been deleted with key `key`. The type (`tag`, `key`, `value`) @ item tracks `tag`, `key`, and `value`, and ensures that the tag matches the current state of the item. The map struct itself consists of a pointer `items` for the items stored in the map, a field `count` counting the number of empty items, and a field `length` counting the total number of items. The type (`mp`, `items`, `count`) @ fixed_size_map keeps track of the mathematical map `mp`, the items as a mathematical list `items`, and the count `count`. To ensure that the physical representation in terms of items corresponds to the mathematical map `mp`, the type maintains an invariant `fsm_invariant mp items count`, which can be found in the accompanying Coq development.

Inference. For the functions `init` and `fsm_realloc_if_necessary`, the resulting specifications from Quiver's inference are depicted in Fig. 17. We highlight three points:

(1) The functional correctness reasoning involved to verify the Hashmap implementation is considerable. The specifications track mathematical maps and lists of custom inductive types (*i.e.*, `item_ref`), and describe the effect of the individual operations on them. For example, `init` returns an empty map and `realloc` resizes the underlying memory by allocating a new map and inserting the old elements, keeping the *contents* of the map unchanged. The remaining map functions, `fsm_insert`, `fsm_remove`, `fsm_get`, `fsm_probe`, which can be found

Init Specification

$$
\begin{aligned}
T_{\texttt{init}}(v_{\texttt{m}}, v_{\texttt{len}})(\Phi) \triangleq\ & \exists (\mathsf{q} : \mathsf{loc}), (\texttt{count1} : \mathbb{Z}).\\
& v_{\texttt{m}} \triangleleft_v \mathsf{own}\ \mathsf{q}\ (\mathsf{any}\ (\texttt{sizeof}(\texttt{struct fixed\_size\_map}))) * v_{\texttt{len}} \triangleleft_v \texttt{num[size\_t]}\ \texttt{count1}\\
& * (\texttt{sizeof}(\texttt{struct item}) \cdot \texttt{count1} \in \texttt{size\_t}) * (1 < \texttt{count1})\\
& * \forall w : \mathsf{val}.\ w \triangleleft_v \mathsf{void} * v_{\texttt{m}} \triangleleft_v \mathsf{own}\ \mathsf{q}\ ((\emptyset, \mathsf{Empty}^{\texttt{count1}}, \texttt{count1}) \mathbin{@} \mathsf{fixed\_size\_map}) \mathrel{-\!\!*} \Phi w
\end{aligned}
$$

Realloc Specification

$$
\begin{aligned}
T_{\texttt{realloc}}(v_{\texttt{m}})(\Phi) \triangleq\ & \exists (\texttt{ptr} : \mathsf{loc}), (\texttt{mp} : \mathsf{gmap}\ \mathbb{Z}\ \mathbb{Z}), (\texttt{items1} : \texttt{list item\_ref}), (\texttt{count1} : \mathbb{Z}).\\
& v_{\texttt{m}} \triangleleft_v \mathsf{own}\ \texttt{ptr}\ ((\texttt{mp, items1, count1}) \mathbin{@} \mathsf{fixed\_size\_map})\\
& * \forall (\texttt{items2} : \texttt{list item\_ref}), (\texttt{count2} : \mathbb{Z}), (w : \mathsf{val}).\\
& \quad \texttt{count1} \le \texttt{count2} * 1 < \texttt{count2} * \texttt{count2} \le \mathsf{max\_int}\ \texttt{size\_t} \mathrel{-\!\!*}\\
& \quad \mathsf{len}\ \texttt{items1} \le \mathsf{len}\ \texttt{items2} * w \triangleleft_v \mathsf{void} \mathrel{-\!\!*}\\
& \quad \texttt{ptr} \triangleleft_l (\texttt{mp, items2, count2}) \mathbin{@} \mathsf{fixed\_size\_map} \mathrel{-\!\!*} \Phi w
\end{aligned}
$$

Fig. 17. Mutable Map Specifications

in the accompanying Coq development, track the effect that these operations have on the mathematical map `mp`. Moreover, we implicitly prove that the operations maintain the internal data structure invariant `fsm_invariant mp items count`.

(2) For the function `init`, Quiver infers the type of the argument `len`, additional side conditions, and the resulting list of entries (*i.e.*, $\texttt{Empty}^{\texttt{count1}}$) that is stored in the map.

(3) The function `fsm_realloc_if_necessary` is mutually recursive with the function `fsm_insert`. Quiver does not support inferring specifications of functions with mutual recursion. To verify this example, we manually cut the dependencies: For `fsm_insert` we assume the reallocation specification in Fig. 17. We use it to infer the specification of `fsm_insert`. (Effectively, we verify the specification of `fsm_insert`, since the sketch we provide for it is precise.) Then, using this inferred specification of `fsm_insert`, we infer a specification for the function `fsm_realloc_if_necessary` (again effectively precise). The resulting specification that we infer for `fsm_realloc_if_necessary`, depicted in Fig. 17, coincides with the one we have assumed for `fsm_insert`.

Differences to the RefinedC implementation. Compared to the implementation verified by RefinedC [5], there are three noteworthy differences:

(1) *Struct* vs. *Union.* The RefinedC implementation uses a **`union`** for the items. Quiver currently does not have support for unions and, thus, we use a single struct to express the three cases of items.

(2) *Integer Values.* The RefinedC implementation stores pointers as values in the map. Here, we consider integer values stored in the map. Consequently, our map tracks a mathematical map $\mathsf{gmap}\ \mathbb{Z}\ \mathbb{Z}$ from integers to integers, instead of a map from integers to types.

(3) *Initialization.* The RefinedC implementation initalizes a map through a loop. Here, we use `xzalloc`, requiring us to prove that the allocated sequence of zero bytes corresponds to a valid map.

```
[[q::parameters(p : loc)]]
[[q::args(own p (any (sizeof(struct fixed_size_map))), _)]]
[[q::ensures(^m ◁ᵥ own p ((∅, ? , ?) @ fixed_size_map))]]
void fsm_init(struct fixed_size_map *m, size_t len) {
  void *storage = xzalloc(sizeof(struct item) * len);
  m->length = len;
  m->items = storage;
  m->count = len;
}


[[q::parameters(ptr: loc, mp: gmap ℤ ℤ, items1: list item_ref, count1: ℤ)]]
[[q::args(own ptr ((mp, items1, count1) @ fixed_size_map))]]
[[q::exists(items2: list item_ref, count2: ℤ)]]
[[q::returns(void)]]
[[q::ensures(ptr ◁ₗ (mp, items2, count2) @ fixed_size_map)]]
[[q::ensures(count1 ≤ count2, 1 < count2 ≤ max_int size_t, len items1 ≤ len items2)]]
[[q::exact_post]]
void fsm_realloc_if_necessary(struct fixed_size_map *m) {
  if(compute_min_count(m->length) <= m->count) {
    return;
  }
  if(m->length < SIZE_MAX / 2 / sizeof(struct item) - 16) {} else { abort(); }

  struct fixed_size_map m2;
  size_t new_len = m->length * 2;
  size_t i;

  fsm_init(&m2, new_len);
  [[q::constraints(∃(idx : ℕ), (items2 : list item_ref), (count2 : ℤ). i ◁ₗ num[size_t] idx     )]]
                   ∗ m ◁ₗ value sz_ptr ptr
                   ∗ ptr ◁ₗ (mp, items1, count1) @ fixed_size_map
                   ∗ m2 ◁ₗ (fsm_copy_entries items1 idx, items2, count2) @ fixed_size_map
                   ∗ count1 + len items1 − idx < count2 ∗ 0 < count1
                   ∗ idx ≤ len items1 ∗ len items1 · 2 ≤ len items2
  for(i = 0; i < m->length; i += 1) {
    struct item *item = &(*m->items)[i];

    if(item->tag == ITEM_ENTRY) {
      fsm_insert(&m2, item->key, item->value);
    }
  }
  free(m->items);
  m->length = m2.length;
  m->count = m2.count;
  m->items = m2.items;
}
```

Fig. 18. fsm_realloc_if_necessary and fsm_init

REFERENCES

[1] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. 2009. Compositional shape analysis by means of bi-abduction. In *POPL*. ACM, 289–300. https://doi.org/10.1145/1480881.1480917

[2] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. 2011. Compositional Shape Analysis by Means of Bi-Abduction. *J. ACM* 58, 6 (2011), 26:1–26:66. https://doi.org/10.1145/2049697.2049700

[3] memcached. 2023. memcached. https://www.memcached.org/.

[4] OpenSSL. 2023. OpenSSL. https://www.openssl.org.

[5] Michael Sammler, Rodolphe Lepigre, Robbert Krebbers, Kayvan Memarian, Derek Dreyer, and Deepak Garg. 2021. RefinedC: automating the foundational verification of C code with refined ownership types. In *PLDI*. ACM, 158–174. https://doi.org/10.1145/3453483.3454036