Appendix

RefinedC: Automating the Foundational Verification of C Code with Refined Ownership Types

Michael Sammler
MPI-SWS

Rodolphe Lepigre
MPI-SWS

Robbert Krebbers
Radboud University Nijmegen

Kayvan Memarian
University of Cambridge

Derek Dreyer
MPI-SWS

Deepak Garg
MPI-SWS

A Thread-Safe Allocator Using a Spinlock

This appendix goes into more detail on the thread safe implementation of the `alloc` example.

A thread can only call the `alloc` function if it has full ownership of the allocator state. And indeed, `alloc` is clearly subject to data races if used concurrently on the same `struct mem_t`. To make the allocator thread safe, the obvious solution is to protect its global state using a lock, and that is exactly what has been done in the function `thread_safe_alloc` of Figure 1. The allocator state is stored in the global variable `data` (line 7), which is protected by spinlock `lock` (line 3). The `thread_safe_alloc` function then simply acquires and releases the lock using `sl_lock` and `sl_unlock` around the call to `alloc` on `data`.[1]

Global variables. Before introducing the spinlock abstraction, we need to take a detour to explain the handling of global variables in RefinedC. Much like function arguments or `struct` fields, global variable are annotated with a type. This type may (again) depend on logical variables specified with `rc::parameters`, and it is itself specified using `rc::global`. However, global variables are special in the sense that their specification (*i.e.*, their type) in only satisfied once they have been explicitly initialized (*e.g.*, by the `main` function).

As a consequence, when a function relies on some global variable being initialized, this fact must be made explicit in its specification with a precondition using the `rc::requires` annotation. Indeed, `thread_safe_alloc` has such a precondition for both global variables `lock` and `data` on line 11. Here, the separation logic assertions `initialized "lock" lid` and `initialized "data" lid`[2] specify that the variables have been initialized, and they also tie the `lid` parameter of the function to the parameter of the same name in the specification of both global variables. This enforces that the two global variables satisfy their specification for the *same* lock identifier.

Spinlock abstraction. The locking mechanism used in `thread_safe_alloc` is a simple spinlock that was previously verified in RefinedC, and that is used here as a library. The spinlock interface relies on two abstract types `spinlock<...>` and `spinlocked<...>`. The former is the type of a spinlock, and it is parameterized by a `lock_id`, *i.e.*, a unique identifier for a particular spinlock instance. The latter corresponds to the type of a value (whose type is given as third argument) that is protected by the lock identified by the first argument.[3] The main idea for using a lock is that the protected data can only be accessed (*i.e.*, the `spinlocked<...>` type stripped from their type) if a token associated to the lock has been obtained. This token is logically returned by `sl_lock` through a postcondition, and it must be given up when calling `sl_unlock` as it is required as a precondition.

```
1  [[rc::parameters("lid : lock_id")]]
2  [[rc::global("spinlock<lid>")]]
3  struct spinlock lock;
4
5  [[rc::parameters("lid : lock_id")]]
6  [[rc::global("spinlocked<lid, {\"data\"}, mem_t>")]]
7  struct mem_t data;
8
9  [[rc::parameters("lid : lock_id", "n : nat")]]
10 [[rc::args      ("n @ int<size_t>")]]
11 [[rc::requires  ("[initialized \"lock\" lid]",
12                  "[initialized \"data\" lid]")]]
13 [[rc::returns   ("optional<&own<uninit<n>>, null>")]]
14 void* thread_safe_alloc(size_t sz) {
15   sl_lock(&lock);
16   rc_unwrap(data);
17   void* ret = alloc(&data, sz);
18   sl_unlock(rc_wrap(&lock));
19   return ret;
20 }
```

Figure 1. Thread-safe allocation function.

Verification. Before we discuss some details of the verification of `thread_safe_alloc`, it is worth pointing out that its specified return type differs from that of `alloc`. Indeed, due to concurrency, `thread_safe_alloc` cannot give any guarantees about whether it will succeed or not. Hence, the `rc::returns` does not give a refinement on the `optional<...>`.

The main challenge for automating the verification of the `thread_safe_alloc` function using RefinedC has to do with the

[1] The `rc_unwrap` and `rc_wrap` macros expand to RefinedC annotations, and they are no-ops as far as C is concerned. Moreover, the `rc_wrap` macro is only explicitly included for clarity: it is automatically inserted by RefinedC.

[2] Inside RefinedC annotations square brackets `[...]` delimit quoted Iris propositions.

[3] The second argument of `spinlock<...>` is a string that uniquely identifies the object that is being protected. Indeed, with our spinlock abstraction *one* lock can protect, *e.g.*, multiple global variables.

`spinlocked<...>` appearing in the type of the protected data. After the lock has been acquired, the `spinlocked<...>` type constructor must be stripped away before being able to use the protected data. Moreover, it must be reinstated before releasing the lock. Hence the question is: how can the type system decide when to unwrap, and then wrap again, the type of the protected data? This may seem like a simple question to answer, but there are several problems. For instance, there may be several different resources protected by the same lock, and not all of them may need to be unwrapped (or even be unwrappable). Also, the `spinlocked<...>` type may be hidden away behind abstractions. Hence, to keep the system as flexible as possible, it is the responsibility of the programmer to guide the type system using annotations. That is the reason for the use of the `rc_unwrap` and `rc_wrap` macros in the implementation of `thread_safe_alloc`.

B Summary of the judgments of RefinedC

Table 1 shows the typing judgments used by RefinedC and gives their semantic interpretation (except for the judgments for l-expressions, which are described informally). The typing judgment for expressions $\vdash_{\text{EXPR}}$ is defined using the standard weakest precondition provided by Iris [1]. The typing judgment for statements $\vdash^{\Sigma}_{\text{STMT}}$ uses the weakest precondition for Caesium statements $\text{wp}^{C}\ s\ \{\Phi\}$, which is parametrized by the control-flow graph C and derived from the standard Iris weakest precondition. Note that $\vdash^{\Sigma}_{\text{IF}}$ presented in the main paper is slightly simplified to the version presented here and all judgments here are simplified compared to their actual definition in Coq dues to complexities which we could not explain in the paper.

C Typing rules for $\vdash^{\Sigma}_{\text{STMT}}$ and $\vdash_{\text{EXPR}}$

This section presents the fixed typing rules for $\vdash^{\Sigma}_{\text{STMT}}$ (in §C.1) and $\vdash_{\text{EXPR}}$ (in §C.2). The reader interested in the rules for the specialized judgments is referred to the accompanying Coq development.

C.1 Typing rules for $\vdash^{\Sigma}_{\text{STMT}}$

Figure 2 shows the typing rules for $\vdash^{\Sigma}_{\text{STMT}}$. The `goto` b statement requires special treatment by the RefinedC type system as it has two different typing rules depending on whether the loop is annotated with a loop invariant H (represented by the atomic assertion b $\triangleleft^{\Sigma}_{\text{BLOCK}}$ H) or not. Thus, the implementation of the type checker special cases `goto` b to apply the correct rule. This is the only statement for which such a special case is required.

C.2 Typing rules for $\vdash_{\text{EXPR}}$

Figure 3 shows the typing rules for $\vdash_{\text{EXPR}}$.

References

[1] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.* 28 (2018), e20. https://doi.org/10.1017/S0956796818000151

class	judgment	description / semantic equivalent
statements	$\vdash^{(C,\overline{(\ell,n)},\exists x.\, \tau(x);H(x))}_{\text{STMT}}\ s$	$\text{wp}^C\ s\ \left\{v.\exists x.\, v \lhd_{\text{v}} \tau(x) * \overline{\ell \lhd_l \text{uninit}(n)} * H(x)\right\}$
	$\vdash^{\Sigma}_{\text{IF}}\ v : \tau\ \mathtt{then}\ s_1\ \mathtt{else}\ s_2$	$v \lhd_{\text{v}} \tau \mathrel{-\!\!*} \vdash^{\Sigma}_{\text{STMT}}\ \mathtt{if}\ v\ \mathtt{then}\ s_1\ \mathtt{else}\ s_2$
	$\vdash^{\Sigma}_{\text{SWITCH}}\ \mathtt{switch}_{\alpha} v : \tau\ \mathtt{case}\ \overline{s}\ \mathtt{default}\ s'$	$v \lhd_{\text{v}} \tau \mathrel{-\!\!*} \vdash^{\Sigma}_{\text{STMT}}\ \mathtt{switch}_{\alpha}\ v\ \mathtt{case}\ \overline{s}\ \mathtt{default}\ s'$
	$\vdash^{\Sigma}_{\text{ASSERT}}\ v : \tau\ ; s$	$v \lhd_{\text{v}} \tau \mathrel{-\!\!*} \vdash^{\Sigma}_{\text{STMT}}\ \mathtt{assert}(v);\ s$
	$\vdash^{\Sigma}_{\text{ANNOTSTMT}}\ \mathtt{annot}_x\ \ell : \tau; s$	$\ell \lhd_l \tau \mathrel{-\!\!*} \vdash^{\Sigma}_{\text{STMT}}\ s$
r-expressions	$\vdash_{\text{EXPR}}\ e\ \{v, \tau.\ G(v, \tau)\}$	$\text{wp}\ e\ \{v.\exists \tau.\, v \lhd_{\text{v}} \tau * G(v, \tau)\}$
	$\vdash_{\text{BINOP}}\ (v_1 : \tau_1) \odot (v_2 : \tau_2)\ \{v, \tau.\ G(v, \tau)\}$	$v_1 \lhd_{\text{v}} \tau_1 \mathrel{-\!\!*} v_2 \lhd_{\text{v}} \tau_2 \mathrel{-\!\!*} \vdash_{\text{EXPR}}\ v_1 \odot v_2\ \{v, \tau.\ G(v, \tau)\}$
	$\vdash_{\text{UNOP}}\ \odot v : \tau\ \{v', \tau'.\ G(v', \tau')\}$	$v \lhd_{\text{v}} \tau \mathrel{-\!\!*} \vdash_{\text{EXPR}}\ \odot v\ \{v', \tau'.\ G(v', \tau')\}$
	$\vdash_{\text{CAS}}\ \mathtt{CAS}(v_1 : \tau_1, v_2 : \tau_2, v_3 : \tau_3)\ \{v, \tau. G(v, \tau)$	$v_1 \lhd_{\text{v}} \tau_1 \mathrel{-\!\!*} v_2 \lhd_{\text{v}} \tau_2 \mathrel{-\!\!*} v_3 \lhd_{\text{v}} \tau_3 \mathrel{-\!\!*} \vdash_{\text{EXPR}}\ \mathtt{CAS}(v_1, v_2, v_3)\ \{v, \tau.\ G(v, \tau)\}$
	$\vdash_{\text{VAL}}\ v \xrightharpoonup{\tau} G(\tau)$	$\exists \tau.\, v \lhd_{\text{v}} \tau * G(v, \tau)$
	$\vdash_{\text{ANNOTEXPR}}\ \mathtt{annot}_x\ v : \tau\ \{G\}$	$v \lhd_{\text{v}} \tau \mathrel{-\!\!*} G$
l-expressions	$\vdash_{\text{PLACE}}\ K[\ell : \tau]\ \{\ell_2, \tau_2, T.\ G(\ell_2, \tau_2, T)\}$	accessing ℓ with type τ using evaluation context K resulting in ℓ_2 with type τ_2 and ℓ having type T with hole
	$\vdash_{\text{READ}}\ \tau\ \{v_2, \tau', \tau_2.\ G(v_2, \tau', \tau_2)\}$	reading from a location with type τ resulting in v_2 with type τ_2 and the location having type τ'
	$\vdash_{\text{WRITE}}\ \ell_1 : \tau_1 \leftarrow v_2 : \tau_2\ \{\tau_3.\ G(\tau_3)\}$	writing v_2 with type τ_2 to ℓ_1 with type τ_1 resulting in ℓ_1 having type τ_3
	$\vdash_{\text{ADDR}}\ \ell : \tau\ \{\tau_2, \tau'.\ G(\tau_2, \tau')\}$	taking address of ℓ with type τ resulting in τ_2 and ℓ having type τ'
Auxiliary judgments	$A_1 <: A_2\ \{G\}$	$A_1 \mathrel{-\!\!*} A_2 * G$

Table 1. Judgments

T-GOTO-PRECOND
$$\frac{\exists H.\ (\mathsf{b} \lhd^{\Sigma}_{\text{BLOCK}} H) * H * \mathsf{True}}{\vdash^{\Sigma}_{\text{STMT}} \texttt{goto}\ \mathsf{b}}$$

T-GOTO
$$\frac{\vdash^{\Sigma}_{\text{STMT}} C[\mathsf{b}]}{\vdash^{\Sigma}_{\text{STMT}} \texttt{goto}\ \mathsf{b}} \quad \Sigma = (C,\ \overline{(\ell, n)},\ \exists x.\ \tau(x); H(x))$$

T-ASSIGN
$$\frac{\vdash_{\text{EXPR}} e_2\ \{v_2, \tau_2. \exists \tau.\ \ell_1 \lhd_l \tau * \vdash_{\text{PLACE}} K[\ell_1 : \tau]\ \{\ell_3, \tau_3, T.\ \vdash_{\text{WRITE}} \ell_3 : \tau_3 \leftarrow v_2 : \tau_2\ \{\tau_4. \ell_1 \lhd_l T[\tau_4] \mathrel{-\!\!*} \vdash^{\Sigma}_{\text{STMT}} s\}\}\}}{\vdash^{\Sigma}_{\text{STMT}} \texttt{store}(p_1, e_2);\ s} \quad p_1 = K[\ell_1]$$

T-RETURN
$$\frac{\vdash_{\text{EXPR}} e\ \{v, \tau.\ \exists x.\ v \lhd_v \tau(x) * \overline{\ell \lhd_l \mathsf{uninit}(n)} * H(x)\}}{\vdash^{\Sigma}_{\text{STMT}} \texttt{return}\ e} \quad \Sigma = (C,\ \overline{(\ell, n)},\ \exists x.\ \tau(x); H(x))$$

T-IF
$$\frac{\vdash_{\text{EXPR}} e\ \{v, \tau.\ \vdash^{\Sigma}_{\text{IF}} v : \tau\ \texttt{then}\ s_1\ \texttt{else}\ s_2\}}{\vdash^{\Sigma}_{\text{STMT}} \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2}$$

T-SWITCH
$$\frac{\vdash_{\text{EXPR}} e\ \{v, \tau.\ \vdash^{\Sigma}_{\text{SWITCH}} \texttt{switch}_\iota v : \tau\ \texttt{case}\ \overline{s_1}\ \texttt{default}\ s_2\}}{\vdash^{\Sigma}_{\text{STMT}} \texttt{switch}_\iota\ e\ \texttt{case}\ \overline{s_1}\ \texttt{default}\ s_2}$$

T-CALL
$$\frac{\begin{array}{c}\vdash_{\text{EXPR}} e_f\ \{v, \tau.\ \exists \overline{\tau_{\text{arg}}}.\ \exists H_1.\ \exists \tau_{\text{ret}}.\ \exists H_2.\ v \lhd_v (\mathsf{fn}(\forall x.\ \overline{\tau_{\text{arg}}(x)}; H_1(x)) \to \exists y.\ \tau_{\text{ret}}(x, y); H_2(x, y)) * \\ \exists x.\ \overline{\vdash_{\text{EXPR}} e\ \{v', \tau'.\ v' \lhd_v \tau_{\text{arg}}(x)} * H_1(x) \mathrel{-\!\!*} \forall v_{\text{ret}}.\ \forall y.\ v_{\text{ret}} \lhd_v \tau_{\text{ret}}(x, y) \mathrel{-\!\!*} H_2(x, y) \mathrel{-\!\!*} \vdash^{\Sigma}_{\text{STMT}} s[\mathsf{i} \mapsto v_{\text{ret}}]\overline{\}}\}\end{array}}{\vdash^{\Sigma}_{\text{STMT}} \texttt{let}\ \mathsf{i} = \texttt{call}\ e_f(\overline{e});\ s}$$

T-ASSERT
$$\frac{\vdash_{\text{EXPR}} e\ \{v, \tau.\ \vdash^{\Sigma}_{\text{ASSERT}} v : \tau\,; s\}}{\vdash^{\Sigma}_{\text{STMT}} \texttt{assert}(e);\ s}$$

T-ANNOTS
$$\frac{\exists \tau.\ \ell \lhd_l \tau * \ \vdash_{\text{PLACE}} K[\ell : \tau]\ \{\ell_2, \tau_2, T.\ \vdash_{\text{ADDR}} \ell_2 : \tau_2\ \{\tau_3, \tau_2'.\ \ell \lhd_l T[\tau_2'] \mathrel{-\!\!*} \vdash^{\Sigma}_{\text{ANNOTSTMT}} \texttt{annot}_x\ \ell_2 : \tau_2; s\}\}}{\vdash^{\Sigma}_{\text{STMT}} \texttt{annot}_x \&p;\ s} \quad p = K[\ell]$$

T-EXPRS
$$\frac{\vdash_{\text{EXPR}} e\ \{v, \tau.\ v \lhd_v \tau \mathrel{-\!\!*} \vdash^{\Sigma}_{\text{STMT}} s\}}{\vdash^{\Sigma}_{\text{STMT}} e;\ s}$$

T-SKIPS
$$\frac{\vdash^{\Sigma}_{\text{STMT}} s}{\vdash^{\Sigma}_{\text{STMT}} \texttt{skip};\ s}$$

Figure 2. Typing rules for $\vdash^{\Sigma}_{\text{STMT}}$

$$\textsc{T-val}\quad \frac{\vdash_{\text{VAL}} v \xrightarrow{\tau} G(v, \tau)}{\vdash_{\text{EXPR}} v\, \{v', \tau.\, G(v', \tau)\}}$$

$$\textsc{T-unop}\quad \frac{\vdash_{\text{EXPR}} e\, \{v, \tau.\ \vdash_{\text{UNOP}} \odot v : \tau\, \{v_2, \tau_2.\, G(v_2, \tau_2)\}\}}{\vdash_{\text{EXPR}} \odot e\, \{v, \tau.\, G(v, \tau)\}}$$

$$\textsc{T-binop}\quad \frac{\vdash_{\text{EXPR}} e_1\, \{v_1, \tau_1.\ \vdash_{\text{EXPR}} e_2\, \{v_2, \tau_2.\ \vdash_{\text{BINOP}} (v_1 : \tau_1) \odot (v_2 : \tau_2)\, \{v_3, \tau_3.\, G(v_3, \tau_3)\}\}\}}{\vdash_{\text{EXPR}} e_1 \odot e_2\, \{v, \tau.\, G(v, \tau)\}}$$

$$\textsc{T-cas}\quad \frac{\vdash_{\text{EXPR}} e_1\, \{v_1, \tau_1.\ \vdash_{\text{EXPR}} e_2\, \{v_2, \tau_2.\ \vdash_{\text{EXPR}} e_3\, \{v_3, \tau_3.\ \vdash_{\text{CAS}} \texttt{CAS}(v_1 : \tau_1, v_2 : \tau_2, v_3 : \tau_3)\, \{v_4, \tau_4. G(v_4, \tau_4)\}\}\}\}}{\vdash_{\text{EXPR}} \texttt{CAS}(e_1, e_2, e_3)\, \{v, \tau.\, G(v, \tau)\}}$$

$$\textsc{T-skipE}\quad \frac{\vdash_{\text{EXPR}} e\, \{v, \tau.\, G(v, \tau)\}}{\vdash_{\text{EXPR}} \texttt{skip}; e\, \{v, \tau.\, G(v, \tau)\}}$$

$$\textsc{T-use}\quad \frac{\exists \tau.\, \ell \triangleleft_l \tau * \vdash_{\text{PLACE}} K[\ell : \tau]\, \{\ell_2, \tau_2, T.\ \vdash_{\text{READ}} \tau_2\, \{v_3, \tau_2', \tau_3.\, \ell \triangleleft_l T[\tau_2'] \mathrel{-\!\!*} G(v_3, \tau_3)\}\}}{\vdash_{\text{EXPR}} \texttt{use}(p)\, \{v, \tau.\, G(v, \tau)\}}\ p = K[\ell]$$

$$\textsc{T-addr-of}\quad \frac{\exists \tau.\, \ell \triangleleft_l \tau * \vdash_{\text{PLACE}} K[\ell : \tau]\, \{\ell_2, \tau_2, T.\ \vdash_{\text{ADDR}} \ell_2 : \tau_2\, \{\tau_3, \tau_2'.\, \ell \triangleleft_l T[\tau_2'] \mathrel{-\!\!*} G(\ell_2, \&_{\mathsf{own}}(\tau))\}\}}{\vdash_{\text{EXPR}} \&p\, \{v, \tau.\, G(v, \tau)\}}\ p = K[\ell]$$

$$\textsc{T-annotE}\quad \frac{\vdash_{\text{EXPR}} e\, \{v, \tau.\ \vdash_{\text{ANNOTEXPR}} \texttt{annot}_x\, v : \tau\, \{G(v, \tau)\}\}}{\vdash_{\text{EXPR}} \texttt{annot}_x(e)\, \{v, \tau.\, G(v, \tau)\}}$$

Figure 3. Typing rules for $\vdash_{\text{EXPR}}$