Appendix
RefinedC: A Foundational Refinement Type System for C
Based on Separation Logic Programming

Michael Sammler
MPI-SWS

Rodolphe Lepigre
MPI-SWS

Kayvan Memarian
University of Cambridge

Derek Dreyer
MPI-SWS

Robbert Krebbers
Radboud University Nijmegen

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 function.

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" lid2 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 post-condition, and it must be given up when calling sl_unlock as it is required as a precondition.

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<...>.

1The 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.
2Inside RefinedC annotations square brackets [...] delimit quoted Iris propositions.
3The 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.
The main challenge for automating the verification of the `thread_safe_alloc` function using RefineC has to do with the `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 RefineC

Table 1 shows the typing judgments used by RefineC 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 $\vdash \Sigma$ 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 due 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 RefineC 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 $b \triangleright \Sigma \text{block} H$) 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 $\vdash \text{expr}$.
<table>
<thead>
<tr>
<th>class</th>
<th>judgment</th>
<th>description / semantic equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>statements</td>
<td>$\vdash$ CAS $(v, r) \rightarrow { v, r, G(v, r) }$</td>
<td>wp $C$ ${ v, \exists x. v.x \rightarrow r } \rightarrow { v, r, \text{uninit}(n) \rightarrow H(x) }$</td>
</tr>
<tr>
<td></td>
<td>$\vdash$ $((\text{switch}(v, r) \rightarrow { v, r, G(v, r) })$</td>
<td>$\forall v. \exists a. v.a \rightarrow r \rightarrow { v, r, \text{uninit}(n) \rightarrow H(x) }$</td>
</tr>
<tr>
<td></td>
<td>$\vdash$ switch $v : r \rightarrow { v, r, G(v, r) }$</td>
<td>$\forall v. \exists a. v.a \rightarrow r \rightarrow { v, r, \text{uninit}(n) \rightarrow H(x) }$</td>
</tr>
<tr>
<td></td>
<td>$\vdash$ $((\text{assert}(v, r) \rightarrow { v, r, G(v, r) })$</td>
<td>$\forall v. \exists a. v.a \rightarrow r \rightarrow { v, r, \text{uninit}(n) \rightarrow H(x) }$</td>
</tr>
<tr>
<td></td>
<td>$\vdash$ $((\text{annotstmt}(v, r) \rightarrow { v, r, G(v, r) })$</td>
<td>$\forall v. \exists a. v.a \rightarrow r \rightarrow { v, r, \text{uninit}(n) \rightarrow H(x) }$</td>
</tr>
<tr>
<td></td>
<td>$\vdash$ ${ v, r, G(v, r) }$</td>
<td>$\forall v. \exists a. v.a \rightarrow r \rightarrow { v, r, \text{uninit}(n) \rightarrow H(x) }$</td>
</tr>
<tr>
<td>l-expressions</td>
<td>$\vdash$ ${ v, r, G(v, r) }$</td>
<td>$\forall v. \exists a. v.a \rightarrow r \rightarrow { v, r, \text{uninit}(n) \rightarrow H(x) }$</td>
</tr>
<tr>
<td></td>
<td>$\vdash$ ${ v, r, G(v, r) }$</td>
<td>$\forall v. \exists a. v.a \rightarrow r \rightarrow { v, r, \text{uninit}(n) \rightarrow H(x) }$</td>
</tr>
<tr>
<td></td>
<td>$\vdash$ ${ v, r, G(v, r) }$</td>
<td>$\forall v. \exists a. v.a \rightarrow r \rightarrow { v, r, \text{uninit}(n) \rightarrow H(x) }$</td>
</tr>
<tr>
<td></td>
<td>$\vdash$ ${ v, r, G(v, r) }$</td>
<td>$\forall v. \exists a. v.a \rightarrow r \rightarrow { v, r, \text{uninit}(n) \rightarrow H(x) }$</td>
</tr>
<tr>
<td></td>
<td>$\vdash$ ${ v, r, G(v, r) }$</td>
<td>$\forall v. \exists a. v.a \rightarrow r \rightarrow { v, r, \text{uninit}(n) \rightarrow H(x) }$</td>
</tr>
</tbody>
</table>

Table 1. Judgments
\[
\begin{align*}
\text{T-goto-precond} & \quad \exists H. (b \triangleright_{\text{block}} H) \land H \land \text{True} \\
\therefore & \quad \Sigma_{\text{stmt}} \text{ goto } b
\end{align*}
\]

\[
\begin{align*}
\text{T-goto} & \quad \Sigma_{\text{stmt}} C[b] \\
\therefore & \quad \Sigma_{\text{stmt}} \text{ goto } b
\end{align*}
\]

\[
\begin{align*}
\Gamma_{\text{expr}} e_2 \{ v_2, r_2, \exists r, t_1 \downarrow t \downarrow \text{place } K[t_1 : t] \{ t_2, r_3, T. \} \} \text{ write } t_2 \{ t_4, t_1 \downarrow t \downarrow T[t_4 \rightarrow \Sigma_{\text{stmt}} s] \} \\
\therefore & \quad \Sigma_{\text{stmt}} \text{ store}(p_1, e_2); s
\end{align*}
\]

\[
\begin{align*}
\Gamma_{\text{expr}} e \{ v, r. \exists \text{ x. } v \downarrow t(x) \downarrow \ell \downarrow \text{uninit}(n) \downarrow H(x) \} \\
\therefore & \quad \Sigma_{\text{stmt}} \text{ return } e
\end{align*}
\]

\[
\begin{align*}
\Gamma_{\text{expr}} e \{ v, r. \exists \text{ x. } \downarrow t \downarrow \text{switch } v : r \text{ case } s_1 \downarrow \text{ default } s_2 \} \\
\therefore & \quad \Sigma_{\text{stmt}} \text{ switch } e \text{ case } s_1 \downarrow \text{ default } s_2
\end{align*}
\]

\[
\begin{align*}
\Gamma_{\text{expr}} e \{ v, r. \exists \text{ arg } \downarrow \exists H_1. \exists \text{ ret } \downarrow \exists H_2. \text{ v } \downarrow \text{ arg } (\text{ fn (v x. } \text{ arg (x)) ; H_1(x) ) \rightarrow \exists y. \text{ ret (x, y) ; H_2(x, y) } \} \downarrow \exists x. \Gamma_{\text{expr}} e \{ v', r'. \text{ v' } \downarrow \text{ arg (x) } \downarrow H_1(x) \downarrow \exists \text{ vret } . \exists y. \text{ vret (x, y) } \downarrow H_2(x, y) \downarrow \rightarrow \Sigma_{\text{stmt}} s[i \mapsto \text{ vret }] \} \\
\therefore & \quad \Sigma_{\text{stmt}} \text{ let } i = \text{ call } e;r; s
\end{align*}
\]

\[
\begin{align*}
\Gamma_{\text{expr}} e \{ v, r. \downarrow \text{ assert } v : r ; s \} \\
\therefore & \quad \Sigma_{\text{stmt}} \text{ assert } (e); s
\end{align*}
\]

\[
\begin{align*}
\exists r. \ell \downarrow t \downarrow \exists \text{ place } K[t : t] \{ t_2, r_3, T. \} \text{ addr } t_2 \{ t_3, r'_2, \ell \downarrow r \downarrow T[r'_2 \rightarrow \Sigma_{\text{annotstmt}} \text{ annot } x l_2 : r_2 ; s] \} \\
\therefore & \quad \Sigma_{\text{stmt}} \text{ annot } x \& p; s
\end{align*}
\]

\[
\begin{align*}
\Gamma_{\text{expr}} e \{ v, r. v \downarrow t \rightarrow \Sigma_{\text{stmt}} s \} \\
\therefore & \quad \Sigma_{\text{stmt}} e; s
\end{align*}
\]

\[
\begin{align*}
\Gamma_{\text{stmt}} s \\
\therefore & \quad \Sigma_{\text{stmt}} \text{ skip } s
\end{align*}
\]

\textbf{Figure 2. Typing rules for } \Sigma_{\text{stmt}}