Explicit Stabilisation for Modular Rely-Guarantee Reasoning

John Wickerson, Mike Dodds and Matthew Parkinson

University of Cambridge Computer Laboratory

Abstract. We propose a new formalisation of stability for Rely-Guarantee, in which an assertion's stability is encoded into its syntactic form. This allows two advances in modular reasoning. Firstly, it enables Rely-Guarantee, for the first time, to verify concurrent libraries independently of their clients' environments. Secondly, in a sequential setting, it allows a module's internal interference to be hidden while verifying its clients. We demonstrate our approach by verifying, using RGSep, the Version 7 Unix memory manager, uncovering a twenty-year-old bug in the process.

1 Introduction

Reasoning about concurrent programs is hard because commands from different threads are interleaved non-deterministically. With many threads and many commands per thread, naïve reasoning soon succumbs to a combinatorial explosion. The Rely-Guarantee (RG) method [14] restores tractability through abstraction. In addition to the pre and postconditions inherited from Hoare logic [12], a command is specified by two relations between states: a *rely* R that specifies all the state transitions (or 'actions') the environment can cause, and a *guarantee* G that specifies all the actions of the command itself. (The environment is the set of concurrently-running threads.) The method conservatively assumes that between consecutive commands in a thread, any number of actions in R may occur. The truth of an assertion that holds after one command must be preserved by this 'interference', so that it may be safely assumed by the next command. Such an assertion is deemed *stable under* R.

Stability is traditionally enforced through side-conditions on proof rules. We propose (Sect. 3) a new formalisation in which stability is recorded within the syntactic form of the assertion itself. Just as 'explicit substitution' [1] added substitution to the syntax of λ-calculus terms, our work adds stabilisation to the syntax of RG assertions. We propose two new constructs: $\lfloor p \rfloor_R$ to denote the weakest assertion that is both stronger than p and stable under R, and dually, $\lceil p \rceil_R$ to denote the strongest stable assertion that is weaker than p.

The main benefit is in modularity, two forms of which we tease apart and tackle separately: verifying concurrent libraries independently of clients, and verifying clients of a (sequential) module independently of its implementation.

Verifying libraries independently of clients. RG is a compositional method: an entire program's proof depends only upon the proofs of its constituent commands.

A.D. Gordon (Ed.): ESOP 2010, LNCS 6012, pp. 611–630, 2010.

Yet it is not modular: a command's proof cannot necessarily be re-used when the command features in a different program, because proofs are environment-specific. Thus, RG cannot verify libraries that are invoked in several different environments. Our solution (Sect. 4) has the library record stability requirements using $\lfloor\ \rfloor_R$ and $\lceil\ \rceil_R$, but leave the specification parametric in R. Each client then instantiates R appropriately and performs the stabilisation.

Verifying clients independently of module implementations. In Sect. 5, we bring explicit stabilisation to an RG-style logic that reasons about heap-manipulating programs: RGSep [20]. Because it divides the heap into both thread-local and shared regions, RGSep's notion of stability is more complex than that of ordinary RG; in particular, while only the shared heap is susceptible to interference, we shall see that the local heap can still affect stability arguments. Originally conceived for concurrency, RGSep is apt for verifying sequential modules too. Such a verification must consider every action by which a client can mutate the module's part of the heap. Our extension of explicit stabilisation to RGSep permits an InfoHiding rule that allows this so-called 'internal interference' to be hidden while verifying clients. We demonstrate (Sect. 6) this approach by verifying – for the first time – the Version 7 Unix memory manager. In doing so, we uncover a bug that has lain dormant since 1979.

We begin with a short introduction to the RG proof system, followed by a brief account of the failure of traditional RG to provide a modular specification for even one of the most trivial library functions: increment.

2 Background: Rely-Guarantee reasoning

RG specifications are of the form $R, G \vdash \{p\}\, C\, \{q\}$, where R and G are relations between states. Following [17], G shall be reflexive. This specification expresses that when C begins execution in a state satisfying the precondition p, in an environment whose interference is limited to the actions in the rely R, then any state transitions performed by C are within its guarantee G, and moreover, if the execution terminates, the final state satisfies the postcondition q.

Figure 1 presents a selection of the RG proof rules, which concern commands of the following simple parallel language:

$$C ::= \texttt{skip} \mid C\,;C \mid C \,\|\, C \mid C + C \mid C^+ \mid c$$

The $+$ operator chooses one of its operands to execute, while C^+ executes C at least once.[1] We consider only partial correctness, so these non-deterministic constructs for choice and looping suffice for encoding `if` and `while` commands. The language is parameterised on the set of basic commands c, which are relations that model atomic state transformations. We shall assume c includes `assert`

[1] Interestingly, a variant of the Loop rule for reasoning about C^* commands would require a stability check on p, in case C^* should behave like `skip`. Our language uses C^+ so as to sidestep this check.

$$\textsc{Weaken}\ \frac{R', G' \vdash \{p'\}\, C\, \{q'\} \quad p \Rightarrow p' \quad q' \Rightarrow q \quad R \subseteq R' \quad G' \subseteq G}{R, G \vdash \{p\}\, C\, \{q\}} \qquad \textsc{Par}\ \frac{R \cup G_2, G_1 \vdash \{p_1\}\, C_1\, \{q_1\} \qquad R \cup G_1, G_2 \vdash \{p_2\}\, C_2\, \{q_2\}}{R, G_1 \cup G_2 \vdash \{p_1 \wedge p_2\}\, C_1 \parallel C_2\, \{q_1 \wedge q_2\}}$$

$$\textsc{Basic}\ \frac{\vdash \{p\}\, c\, \{q\} \quad \overleftarrow{p} \cap c \subseteq G \quad p\ \mathsf{stab}\ R \quad q\ \mathsf{stab}\ R}{R, G \vdash \{p\}\, c\, \{q\}} \qquad \textsc{Skip}\ \frac{p\ \mathsf{stab}\ R}{R, G \vdash \{p\}\ \mathtt{skip}\ \{p\}} \qquad \textsc{Loop}\ \frac{R, G \vdash \{p\}\, C\, \{p\}}{R, G \vdash \{p\}\, C^+\, \{p\}}$$

Fig. 1: Selected RG proof rules (with stability checks)

and `assume` commands and variable assignment. See [21] for the complete set of proof rules and the formal semantics of our language.

The Basic rule requires that c meets the sequential specification $\{p\}\, c\, \{q\}$, and that any action it performs is within its guarantee. It uses the notation $\overleftarrow{p} \stackrel{\text{def}}{=} \{(\sigma, \sigma') \mid \sigma \models p\}$. The pre and postconditions of the two 'ground' commands, c and `skip`, are required to be stable. Since the other commands are built inductively from these, their rules can assume any inherited assertions to be stable (or else derived from stable assertions by the Weaken rule). Stability checks are notated as follows:

Definition 1 (Stability). $p\ \mathsf{stab}\ R \stackrel{\text{def}}{=} \forall \sigma, \sigma'.\, \sigma \models p \ \wedge\ R(\sigma, \sigma') \implies \sigma' \models p.$

The Par rule marks the epitome of RG reasoning. When reasoning about commands composed in parallel, the rely of each command is extended to include the guarantee of the other. The composed command $C_1 \parallel C_2$ guarantees actions in either of its components' guarantees, and establishes both components' postconditions upon completion.

2.1 The problem with verifying libraries

Consider a library function `f()` that atomically increments a shared variable `x`. Its two clients, `g()` and `h()`, invoke `f()` in an empty environment and an environment that may increase `x`, respectively. Call this latter environment $R_{\mathtt{x+}}$. The guarantee $G_{\mathtt{x+}}$ additionally dictates that no variable other than `x` changes.

Definition 2.
$$\begin{aligned}
\mathtt{f()} &\stackrel{\text{def}}{=} \mathtt{x{+}{+}} \\
\mathtt{g()} &\stackrel{\text{def}}{=} \mathtt{assume(x{=}3)\,;\ f()\,;\ assert(x{=}4)} \\
\mathtt{h()} &\stackrel{\text{def}}{=} \mathtt{assume(x{=}5)\,;\ (f()} \parallel \mathtt{f())\,;\ assert(x{\geq}6)} \\
R_{\mathtt{x+}} &\stackrel{\text{def}}{=} \{(\sigma, \sigma') \mid \sigma(\mathtt{x}) \leq \sigma'(\mathtt{x})\} \\
G_{\mathtt{x+}} &\stackrel{\text{def}}{=} \{(\sigma, \sigma') \mid \sigma(\mathtt{x}) \leq \sigma'(\mathtt{x}) \ \wedge\ \forall v \neq \mathtt{x}.\, \sigma(v) = \sigma'(v)\}
\end{aligned}$$

Now, the proofs of `g()` and `h()` hinge, respectively, upon deriving the following two specifications for `f()`:

$$\emptyset, G_{\mathtt{x+}} \vdash \{\mathtt{x}{=}X\}\, \mathtt{f()}\, \{\mathtt{x}{=}X{+}1\} \qquad R_{\mathtt{x+}}, G_{\mathtt{x+}} \vdash \{\mathtt{x}{\geq}X\}\, \mathtt{f()}\, \{\mathtt{x}{\geq}X{+}1\}$$

Both hold, yet no single 'most general' specification can derive them both. The first has the stronger postcondition but the smaller rely; the second is vice versa. This troublesome tradeoff can be blamed on stability: the larger the rely, the tougher the stability requirement, and thus, the weaker the postcondition.

In Sect. 4, we shall present a single specification for `f()` from which both of the above can be derived. Parameterised on an arbitrary rely R, it simply states that the postcondition needs weakening from $\mathtt{x}{=}X{+}1$ just enough to become stable under R. Upon instantiating R to $R_{\mathtt{x}+}$, to verify `h()`, the postcondition becomes $\mathtt{x}{\geq}X{+}1$. And when R is $\emptyset$, for `g()`'s proof, no weakening is required.

3 Explicit Stabilisation

This section describes our formalisation of stability and applies it to the RG proof rules. The remaining sections develop two alternate proof systems: one (Sect. 4) that can specify libraries independently of clients, and another (Sects. 5 and 6) that lets a module hide from clients its internal interference.

We propose two new syntactic constructs: $\lfloor p \rfloor_R$ for the weakest assertion that is stronger than p and stable under R, and $\lceil p \rceil_R$ for the strongest assertion that is weaker than p and stable under R. That is, $\lfloor p \rfloor_R = \bigvee\{q \mid q \Rightarrow p \wedge q \mathsf{\ stab\ } R\}$ and $\lceil p \rceil_R = \bigwedge\{q \mid q \Leftarrow p \wedge q \mathsf{\ stab\ } R\}$.

Definition 3 (Semantics of $\lfloor p \rfloor_R$ and $\lceil p \rceil_R$). *The required properties are realised uniquely by the following constructions:*

$$\sigma \models \lfloor p \rfloor_R \stackrel{\text{def}}{\iff} \forall \sigma'.\, (\sigma, \sigma') \in R^* \implies \sigma' \models p$$
$$\sigma \models \lceil p \rceil_R \stackrel{\text{def}}{\iff} \exists \sigma'.\, (\sigma', \sigma) \in R^* \wedge \sigma' \models p.$$

Figure 2 presents the intuition behind our new operators. The nodes represent states; those that are filled satisfy some assertion p. The edges depict transitions of an arbitrary rely R. The states in $\lfloor p \rfloor_R$ are those from which any reachable state satisfies p. The states in $\lceil p \rceil_R$ are those reachable from a state in p.

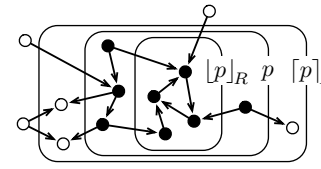

Fig. 2

Our operators can also be defined using Dijkstra's predicate transformer semantics [6]: $\lfloor p \rfloor_R$ is the weakest precondition of R^* given postcondition p, while $\lceil p \rceil_R$ is the strongest postcondition of R^* given precondition p.

Example. We stabilise $\mathtt{x}{=}0$ and $\mathtt{x}{\neq}0$ under $R_{\mathtt{x}+}$ (see Definition 2) like so:

$$\lfloor \mathtt{x}{=}0 \rfloor_{R_{\mathtt{x}+}} \Leftrightarrow \mathsf{false} \qquad \lceil \mathtt{x}{=}0 \rceil_{R_{\mathtt{x}+}} \Leftrightarrow \mathtt{x}{\geq}0 \qquad \lfloor \mathtt{x}{\neq}0 \rfloor_{R_{\mathtt{x}+}} \Leftrightarrow \mathtt{x}{>}0 \qquad \lceil \mathtt{x}{\neq}0 \rceil_{R_{\mathtt{x}+}} \Leftrightarrow \mathsf{true}$$

3.1 Properties of explicit stabilisation

Both $\lfloor\ \rfloor$ and $\lceil\ \rceil$ are monotonic with respect to $\Rightarrow$. They are related via the equivalence $\lfloor \neg p \rfloor_R \Leftrightarrow \neg \lceil p \rceil_{R^{-1}}$. Each has no effect on an already-stable operand,

or when R is empty. Both true and false are stable, and conjunction and disjunction both preserve stability. The distributivity properties of $\lfloor\ \rfloor$ and $\lceil\ \rceil$ over $\wedge$ and $\vee$ are analogous to those of $\forall$ and $\exists$ respectively:

$$\begin{array}{lll} \lfloor p \wedge q \rfloor_R \;\Leftrightarrow\; \lfloor p \rfloor_R \wedge \lfloor q \rfloor_R & & \lfloor p \vee q \rfloor_R \;\Leftarrow\; \lfloor p \rfloor_R \vee \lfloor q \rfloor_R \\ \lceil p \wedge q \rceil_R \;\Rightarrow\; \lceil p \rceil_R \wedge \lceil q \rceil_R & & \lceil p \vee q \rceil_R \;\Leftrightarrow\; \lceil p \rceil_R \vee \lceil q \rceil_R \end{array}$$

Several properties mirror those of the floor and ceiling functions in arithmetic, from which our syntax is borrowed. If $R \subseteq R'$, we have:

$$\begin{array}{l} \lfloor \lfloor p \rfloor_R \rfloor_{R'} \;\Leftrightarrow\; \lfloor \lfloor p \rfloor_{R'} \rfloor_R \;\Leftrightarrow\; \lceil \lfloor p \rfloor_{R'} \rceil_R \;\Leftrightarrow\; \lfloor p \rfloor_{R'} \\ \lceil \lceil p \rceil_R \rceil_{R'} \;\Leftrightarrow\; \lceil \lceil p \rceil_{R'} \rceil_R \;\Leftrightarrow\; \lfloor \lceil p \rceil_{R'} \rfloor_R \;\Leftrightarrow\; \lceil p \rceil_{R'} \end{array}$$

Finally, the following property reminds us of the trade-off mentioned in Sect. 2.1: that as the rely becomes more permissive, stability becomes harder to show:

$$R \subseteq R' \text{ implies } \lfloor p \rfloor_R \Leftarrow \lfloor p \rfloor_{R'} \text{ and } \lceil p \rceil_R \Rightarrow \lceil p \rceil_{R'}$$

3.2 Application to RG proof rules

We now describe how the RG proof rules (Fig. 1) can be adapted to use explicit stabilisation rather than side-conditions.

Figure 3 displays the replacements for the Basic and Skip rules; the others remain unchanged. The Basic-S rule first derives p and q by considering c sequentially; that is, without concern for stability. A concurrent specification is obtained by strengthening p and weakening q until they are both stable. The Skip-S axiom is justified by considering the execution of `skip` from an initial state satisfying p. This state also satisfies $\lceil p \rceil_R$, and the final state must too, since `skip` does nothing. The following backward-reasoning alternative is interderivable: $R, G \vdash \{\lfloor p \rfloor_R\}\ \mathtt{skip}\ \{p\}$.

$$\textsc{Basic-S}\quad \frac{\vdash \{p\}\, c\, \{q\} \qquad \overleftarrow{p} \cap c \subseteq G}{R, G \vdash \{\lfloor p \rfloor_R\}\, c\, \{\lceil q \rceil_R\}}$$

$$\textsc{Skip-S}\quad \frac{}{R, G \vdash \{p\}\ \mathtt{skip}\ \{\lceil p \rceil_R\}}$$

Fig. 3: New RG proof rules (with stabilised assertions)

The new rules are at least as powerful as the originals, which can be obtained by restoring the stability checks and then removing the redundant stabilisations.

3.3 Aside: Simplification of complex RG proof rules

We now highlight the elegance of explicit stabilisation by showing how it can simplify and generalise complex RG proof rules that rely subtly upon stability.

Coleman [5] proposes the following rule for reasoning about one-armed conditional statements whose test conditions are evaluated non-atomically in the presence of interference.

$$\frac{\mathsf{StableExpr}(e_s, R) \qquad R, G \vdash \{p \wedge e_s\}\, C\, \{q\} \qquad \mathsf{SingleUnstableVar}(e_u, R) \\ \forall \sigma, \sigma'.\, \sigma \models p \wedge (\sigma, \sigma') \in R^* \wedge \sigma' \models \neg(e_s \wedge e_u) \Longrightarrow \sigma' \models q \qquad \{\neg e_u, p, q\}\ \mathsf{stab}\ R}{R, G \vdash \{p\}\ \mathtt{if}\ \ e_u \wedge e_s\ \ \mathtt{then}\ \ C\, \{q\}}$$

Tests are pure, and comprise an unstable conjunct e_u and a 'stable' conjunct e_s that contains no variables that R can change (first premise). Crucially, only e_s can be assumed still to hold by C (second premise). By requiring e_u to involve only a single read of an unstable variable (third premise), we can treat it as a predicate of a single state – the state in which the read occurs – despite not knowing which state that is. Should the test fail, the postcondition must be met without evaluating C (fourth premise). That premise requires R to preserve the falsity of e_u (fifth premise) so as to ensure that the obligation to fulfil q cannot be bypassed by having the test evaluate to `false` but later become logically `true`.

Now consider the following alternative rule, which uses explicit stabilisation.

$$\frac{\mathsf{SingleUnstableVar}(e,R) \qquad \{p,q\}\ \mathsf{stab}\ R \qquad \begin{array}{l} R,G \vdash \{p \wedge \lceil e \rceil_R\}\, C_1\, \{q\} \\ R,G \vdash \{p \wedge \lceil \neg e \rceil_R\}\, C_2\, \{q\} \end{array}}{R,G \vdash \{p\}\ \texttt{if}\ e\ \texttt{then}\ C_1\ \texttt{else}\ C_2\, \{q\}}$$

Essentially, the execution of C_1 begins in a state that is reachable (by a sequence of environment actions) from one in which e evaluated to `true`. Similarly, $\lceil \neg e \rceil_R$ describes a state reached from one where e did not hold. Stability checks on p and q remain only for compatibility with the rest of Coleman's system.

Thanks to explicit stabilisation, the new rule has fewer and simpler premises, plus it extends naturally to two-armed conditionals. Moreover, e need not be split into stable and unstable conjuncts, for our rule handles arbitrary test conditions.

4 Verifying concurrent library code

Equipped with a notation for stabilising assertions, we revisit the challenge we set in Sect. 2.1: to verify concurrent library code using RG reasoning.

Recall our library function `f()` and its clients `g()` and `h()` from Definition 2. Using explicit stabilisation, we can derive the following specification, which is parametric in R (although its instantiation will be restricted, as described shortly).

$$R, G_{\mathtt{x+}} \vdash \{\lceil \mathtt{x}{=}X \rceil_R\}\, \mathtt{f()}\, \{\lceil \mathtt{x}{=}X{+}1 \rceil_R\}$$

Observe that instantiating R to $\emptyset$ yields a specification suitable for proving `g()`, while `h()` can be proved having set R to $R_{\mathtt{x+}}$. We now present a proof system for such 'parametric specifications' and formally derive the above one for `f()`.

In a parametric specification, the rely is replaced by a set of relies $\mathbb{R}$, and the pre and postconditions (denoted $\boldsymbol{p}$, $\boldsymbol{q}$, $\boldsymbol{r}$) become functions from relies to assertions. We shall use λ-calculus notation to describe such functions.

Definition 4. $\mathbb{R}, G \models_{\mathsf{P}} \{\boldsymbol{p}\}\, C\, \{\boldsymbol{q}\} \stackrel{\text{def}}{\iff} \forall R \in \mathbb{R}.\, R, G \models \{\boldsymbol{p}(R)\}\, C\, \{\boldsymbol{q}(R)\}$.

As the definition above shows, a parametric specification represents a family of specifications, one for each rely in $\mathbb{R}$. A selection of proof rules for parametric specifications are presented in Fig. 4; those not depicted are lifted in the obvious way. (See [21] for the full set.)

$$\textsc{P-Weaken}\ \frac{\mathbb{R}', G' \vdash_{\mathsf{P}} \{\boldsymbol{p}'\}\, C\, \{\boldsymbol{q}'\} \quad \boldsymbol{p} \Rightarrow_{\mathbb{R}} \boldsymbol{p}' \quad \boldsymbol{q}' \Rightarrow_{\mathbb{R}} \boldsymbol{q} \quad \mathbb{R} \subseteq \mathbb{R}' \quad G' \subseteq G}{\mathbb{R}, G \vdash_{\mathsf{P}} \{\boldsymbol{p}\}\, C\, \{\boldsymbol{q}\}}$$

$$\textsc{P-Par}\ \frac{\mathbb{R} \cup G_2, G_1 \vdash_{\mathsf{P}} \{\boldsymbol{p}_1\}\, C_1\, \{\boldsymbol{q}_1\} \quad \mathbb{R} \cup G_1, G_2 \vdash_{\mathsf{P}} \{\boldsymbol{p}_2\}\, C_2\, \{\boldsymbol{q}_2\}}{\mathbb{R}, G_1 \cup G_2 \vdash_{\mathsf{P}} \{\boldsymbol{p}_1\ {}_{G_2}\|_{G_1}\ \boldsymbol{p}_2\}\, C_1 \,\|\, C_2\, \{\boldsymbol{q}_1\ {}_{G_2}\|_{G_1}\ \boldsymbol{q}_2\}}$$

$$\textsc{P-Basic}\ \frac{\vdash \{p\}\, c\, \{q\} \quad \overleftarrow{p} \cap c \subseteq G}{\mathbb{U}, G \vdash_{\mathsf{P}} \{\lambda R.\, \lfloor p \rfloor_R\}\, c\, \{\lambda R.\, \lceil q \rceil_R\}} \qquad \textsc{P-Skip}\ \frac{}{\mathbb{U}, G \vdash_{\mathsf{P}} \{\lambda \_.\, p\}\ \mathtt{skip}\ \{\lambda R.\, \lceil p \rceil_R\}}$$

Abbreviations:

$$\boldsymbol{p}_1 \Rightarrow_{\mathbb{R}} \boldsymbol{p}_2 \stackrel{\text{def}}{=} \forall R \in \mathbb{R}.\, \boldsymbol{p}_1(R) \Rightarrow \boldsymbol{p}_2(R) \qquad \mathbb{R} \cup R \stackrel{\text{def}}{=} \{R' \cup R \mid R' \in \mathbb{R}\}$$

$$\boldsymbol{p}_1\ {}_{R_1}\|_{R_2}\ \boldsymbol{p}_2 \stackrel{\text{def}}{=} \lambda R.\, \boldsymbol{p}_1(R \cup R_1) \wedge \boldsymbol{p}_2(R \cup R_2) \qquad \mathbb{U} \stackrel{\text{def}}{=} \text{universal set of all relies}$$

Fig. 4: Selected proof rules for parametric specifications

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \{p\}\ \mathtt{x{+}{+}}\ \{p[\mathtt{x}-1/\mathtt{x}]\}}\ \textsc{Floyd's Assignment Axiom}}{\vdash \{\lceil \mathtt{x}{=}X \rceil_R\}\ \mathtt{x{+}{+}}\ \{\lceil \mathtt{x}{=}X \rceil_R [\mathtt{x}-1/\mathtt{x}]\}}\ \text{Instantiate } p \text{ to } \lceil \mathtt{x}{=}X \rceil_R}{\mathbb{U}, G_{\mathtt{x}+} \vdash_{\mathsf{P}} \{\lambda R.\, \lceil \mathtt{x}{=}X \rceil_R\}\ \mathtt{x{+}{+}}\ \{\lambda R.\, \lceil \lceil \mathtt{x}{=}X \rceil_R [\mathtt{x}-1/\mathtt{x}] \rceil_R\}}\ \textsc{P-Basic}}{\mathsf{comm}(\mathtt{x{+}{+}}), G_{\mathtt{x}+} \vdash_{\mathsf{P}} \{\lambda R.\, \lceil \mathtt{x}{=}X \rceil_R\}\ \mathtt{x{+}{+}}\ \{\lambda R.\, \lceil \mathtt{x}{=}X{+}1 \rceil_R\}}\ \textsc{P-Weaken}$$

Fig. 5: Derivation of parametric specification for `f()`

The P-Par rule has grown considerably more complex. The reason is that at the fork and join of parallel commands, the rely changes. If the rely is R initially, then within the component commands the rely becomes either $R \cup G_2$ or $R \cup G_1$, and after joining, it reverts to R. Our rule simply reflects this progression.

The P-Basic and P-Skip rules both deduce specifications that feature the universal set of relies, which enables their use in *any* environment. The P-Weaken rule can then be used to shrink this set, typically removing the bigger relies. Doing so restricts a specification's reusability, but it enhances the applicability of the $\Rightarrow_{\mathbb{R}}$ relation that allows it to be simplified.

Theorem 5. *The proof rules of parametric stability are sound, that is:*

$$\mathbb{R}, G \vdash_{\mathsf{P}} \{\boldsymbol{p}\}\, C\, \{\boldsymbol{q}\} \implies \mathbb{R}, G \models_{\mathsf{P}} \{\boldsymbol{p}\}\, C\, \{\boldsymbol{q}\}$$

and they encode the proof rules of Fig. 1 (in which assertions do not contain explicit stabilisation), both completely and soundly, that is:

$$R, G \vdash \{p\}\, C\, \{q\} \implies \mathcal{P}(R), G \vdash_{\mathsf{P}} \{\lambda \_.\, p\}\, C\, \{\lambda \_.\, q\}$$
$$R, G \models \{p\}\, C\, \{q\} \impliedby \mathcal{P}(R), G \models_{\mathsf{P}} \{\lambda \_.\, p\}\, C\, \{\lambda \_.\, q\}$$

Here, the use of powersets lets the P-Weaken *rule emulate the* Weaken *rule.*

Figure 5 shows the derivation of our specification for `f()`. In applying the P-Basic rule, we utilised the identity $\lfloor \lceil \mathtt{x}{=}X \rceil_R \rfloor_R \Leftrightarrow \lceil \mathtt{x}{=}X \rceil_R$. The specification

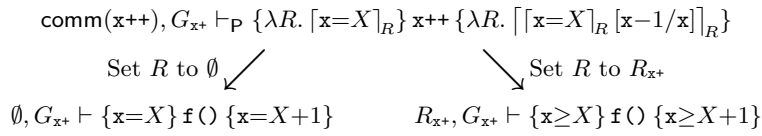

Fig. 6: Instantiating the specification

on the third line is the most general, as it allows the rely to be instantiated freely. Yet we do not stop there. We restrict the rely to the set comm(x++) of those that 'commute' with the x++ operation; that is, for which $\lceil p \rceil_R\ [x{-}1/x] \Leftrightarrow \lceil p[x{-}1/x] \rceil_R$ holds for all p. Using this property we can simplify the postcondition.

Figure 6 shows informally how the parametric specification can then be instantiated to two ordinary specifications, for use in proving the two clients g() and h(). Really, this 'instantiation' is an application of the P-Weaken rule to restrict $\mathbb{R}$ to the singletons $\{\emptyset\}$ and $\{R_{x+}\}$ respectively.[2]

In conclusion, we find that the 'most general' specifications that our parametric scheme can deduce are, though sometimes desirable, inhibited by their complexity. The specification on the third line of Fig. 5 contains two stabilisation operations in its postcondition – and this is for just a single basic command. A sequence of n basic commands, specified in a similar way, may contain up to $n + 1$ stabilisation operations in the postcondition (modelling the environmental interference before, between and after the commands). The complexity of the specification is thus comparable to the implementation it describes. Accordingly, it is crucial that our scheme allows specifications to be specialised to restricted sets of relies, and thence, simplified.

5 Explicit Stabilisation for RGSep

We now bring explicit stabilisation to RGSep [20], an RG-style logic that reasons about concurrent heap-manipulating programs by splitting the heap into shared and thread-local parts. The development in this section builds upon our application of explicit stabilisation to RG (Sect. 3), but we shall now leave behind the parametric specifications of Sect. 4.

Though designed for concurrency, we show (Sect. 5.3) how RGSep can be applied to sequential modules by reinterpreting the 'shared' heap as that part owned by the module (its so-called 'internal heap'). Our extension of RGSep with explicit stabilisation enables an InfoHiding proof rule, by which a module can hide from clients the interference that affects its internal heap. We demonstrate our approach in Sect. 6, by verifying the Version 7 Unix memory manager.

[2] Interestingly, although the relies $\emptyset$ and R_{x+} are both in comm(x++), the same is not true of all those in $\mathcal{P}(R_{x+})$: for instance, the rely that only increments x from 1 to 2.

$$P ::= e \overset{k}{\mapsto} e \mid \mathsf{emp} \mid e = e \mid e > e \mid \mathsf{true} \mid \neg P \mid P \Rightarrow P \mid P * P \mid \exists x.\, P \mid \lfloor P \rfloor_R \mid \lceil P \rceil_R$$

where $k \in (0, 1]$ and e is a pure expression

$$\begin{array}{lll}
h, i \models_{\mathsf{SL}} e_0 \overset{k}{\mapsto} e_1 & \overset{\text{def}}{\iff} & h = \{[\![e_0]\!]_i \overset{k}{\mapsto} [\![e_1]\!]_i\} \\
h, i \models_{\mathsf{SL}} \mathsf{emp} & \overset{\text{def}}{\iff} & h = \emptyset \\
h, i \models_{\mathsf{SL}} P_0 * P_1 & \overset{\text{def}}{\iff} & \exists h_0, h_1.\, h_0 \bot h_1 \ \wedge\ h = h_0 \uplus h_1 \ \wedge\ h_0, i \models_{\mathsf{SL}} P_0 \ \wedge\ h_1, i \models_{\mathsf{SL}} P_1 \\
h, i \models_{\mathsf{SL}} \lfloor P \rfloor_R & \overset{\text{def}}{\iff} & \forall h'.\, (h, h') \in R^* \implies h', i \models_{\mathsf{SL}} P \\
h, i \models_{\mathsf{SL}} \lceil P \rceil_R & \overset{\text{def}}{\iff} & \exists h'.\, (h', h) \in R^* \ \wedge\ h', i \models_{\mathsf{SL}} P
\end{array}$$

where $h \bot h'$ means $\mathrm{dom}(h)$ and $\mathrm{dom}(h')$ are disjoint.

Fig. 7: Syntax and (selected) semantics of separation logic assertions

$$p ::= P \mid \boxed{P} \mid p * p \mid p \wedge p \mid p \vee p \mid \exists x.\, p \mid \forall x.\, p \mid \lfloor p \rfloor_R \mid \lceil p \rceil_R$$

$$\begin{array}{lll}
l, s, i \models P & \overset{\text{def}}{\iff} & l, i \models_{\mathsf{SL}} P \\
l, s, i \models \boxed{P} & \overset{\text{def}}{\iff} & l = \emptyset \ \wedge\ s, i \models_{\mathsf{SL}} P \\
l, s, i \models p_0 * p_1 & \overset{\text{def}}{\iff} & \exists s_0, s_1.\, s_0 \bot s_1 \ \wedge\ s = s_0 \uplus s_1 \ \wedge\ l, s_0, i \models p_0 \ \wedge\ l, s_1, i \models p_1 \\
l, s, i \models \lfloor p \rfloor_R & \overset{\text{def}}{\iff} & \forall s'.\, (s, s') \in (R \backslash l)^* \implies l, s', i \models p \\
l, s, i \models \lceil p \rceil_R & \overset{\text{def}}{\iff} & \exists s'.\, (s', s) \in (R \backslash l)^* \ \wedge\ l, s', i \models p
\end{array}$$

Fig. 8: Syntax and (selected) semantics of RGSep assertions

5.1 Introduction to RGSep

RGSep extends ordinary RG reasoning with conceptual divisions of the heap into thread-local and shared parts. The rely and guarantee need specify only changes to the shared part, and thus become far more compact.

RGSep inherits its ability to reason naturally about heap-manipulating programs from separation logic [13, 18], the assertion language of which is presented in Fig. 7. States comprise a heap h mapping locations to values and a store i mapping variables to values. The $*$ operator attempts to split the heap using the $\uplus$ operator, such that the two (disjoint) parts respectively satisfy its two operands. We use the fractional permissions model [3], in which a heap may describe some locations only partially. For instance, the assertion $x \overset{1}{\mapsto} 3$ describes a heap comprising a single location x with value 3, and confers full (write) permission on that location. It may be split into several read-only permissions (e.g. $x \overset{.5}{\mapsto} 3 * x \overset{.5}{\mapsto} 3$) which may be shared between different threads. Threads communicate only via the heap, so the stabilisation operators can ignore the store.

Figure 8 presents the assertion language of RGSep, augmented with explicit stabilisation. The heap is split into disjoint local and shared regions, l and s, which are described by unboxed and boxed assertions respectively. The $*$ operator now splits only the local heap. The shared heap is never split, in order that all threads share the same view of it. For instance, if one thread's view of the overall state is described by $\boxed{P_s} * P_l$, and another's by $\boxed{Q_s} * Q_l$, then the $*$ operator combines them thus: $\boxed{P_s \wedge Q_s} * P_l * Q_l$.

Definition 6 (RGSep actions). *The* action $P \rightsquigarrow Q$, *defined* $\{(s \uplus s_0, s' \uplus s_0) \mid \exists i.\, s, i \models_{\mathsf{SL}} P \ \wedge\ s', i \models_{\mathsf{SL}} Q\}$, *replaces a part of the shared heap satisfying* P *with one satisfying* Q.

Definition 7 (Contextual actions). *The* contextual action $P \rightsquigarrow Q \mid F$, *defined* $\{(s \uplus s_F \uplus s_0, s' \uplus s_F \uplus s_0) \mid \exists i.\, s, i \models_{\mathsf{SL}} P \ \wedge\ s', i \models_{\mathsf{SL}} Q \ \wedge\ s_F, i \models_{\mathsf{SL}} F\}$, *requires a separate (unaffected) part of the heap that satisfies* F *to catalyse it.*

5.2 RGSep and stabilisation

Our semantics of $\lfloor p \rfloor_R$ and $\lceil p \rceil_R$ (Fig. 8) imposes the following restriction on R:

Definition 8 (Restricting the rely). $R \backslash l \stackrel{\text{def}}{=} \{(s, s') \in R \mid l \perp s \ \wedge\ l \perp s'\}$

The $R \backslash l$ operation removes from R impossible environmental actions that would make the shared heap overlap the current thread's local heap l.[3]

All of the properties detailed in Sect. 3.1 continue to hold. The following series of lemmas describe some additional RGSep-specific properties. Lemma 9 asserts that local assertions are vacuously stable.

Lemma 9 (Local assertions). $\lfloor P \rfloor_R \Leftrightarrow \lceil P \rceil_R \Leftrightarrow P$.

The next lemma says that we need not restrict the rely when stabilising a shared assertion. Such assertions imply that the local heap is empty (see Fig. 8), and thus unable to conflict with the shared heap.

Lemma 10 (Shared assertions). $\left\lfloor \boxed{P} \right\rfloor_R \Leftrightarrow \boxed{\lfloor P \rfloor_R}$ *and* $\left\lceil \boxed{P} \right\rceil_R \Leftrightarrow \boxed{\lceil P \rceil_R}$.

Finally, we describe the distributivity of the stabilisation operators over $*$.

Lemma 11 (Separately-conjoined assertions). $\lfloor p \rfloor_R * \lfloor q \rfloor_R \ \Rightarrow\ \lfloor p * q \rfloor_R$ *and* $\lceil p * q \rceil_R \ \Rightarrow\ \lceil p \rceil_R * \lceil q \rceil_R$.

Remark. Neither converse implication holds. Obtain a counterexample for the first from p as $\boxed{\mathtt{t} \mapsto 0} * \mathtt{x} \mapsto 0 \vee \boxed{\mathtt{t} \mapsto 1} * \mathtt{y} \mapsto 0$, q the same but with `x` and `y` swapped, and R as the single action $\mathtt{t} \mapsto 0 \rightsquigarrow \mathtt{t} \mapsto 1$. For the second, take p as $\boxed{\exists n.\, \mathtt{t} \mapsto n \wedge n < 0}$, q as $\boxed{\exists n.\, \mathtt{t} \mapsto n \wedge n > 0}$, and R able to increase `t`'s value.

The proof rules of RGSep can be adapted to use explicit stabilisation. Figure 9 shows the replacement for RGSep's frame rule (see [21] for the complete set of new rules). The original rule required the frame r (which must not mention any local variables modified by C) to be stable under both R and G in case any shared heap it specifies is mutated by either the environment or C itself. In the new rule, this check becomes an explicit stabilisation on r in the postcondition. As in the Skip-S rule (Fig. 3), the stabilisation could equally be done in the precondition instead.

$$\textsc{Frame-S}\quad \frac{R, G \vdash \{p\}\, C\, \{q\} \qquad \mathrm{fv}(r) \cap \mathrm{mods}(C) = \emptyset}{R, G \vdash \{p * r\}\, C\, \{q * \lceil r \rceil_{R \cup G}\}}$$

Fig. 9: New frame rule

[3] This approach slightly refines the presentation of stability in [19, Lem. 15], which did not consider such conflicts between shared and local heaps.

5.3 RGSep and sequential modules

This discussion lays the groundwork for the verification of a memory manager presented in Sect. 6. We shall assume a module comprises some state, including several heap locations, plus a collection of public routines that can manipulate this so-called 'internal heap'. A sequential module is one designed for single-threaded machines: its routines and all of its clients are sequential.

Sequential modules are analogous to the concurrent programs that RGSep was designed to verify. The RG method, of abstracting a command's environment by a rely, applies to both, albeit for different reasons. For concurrent programs, we must abstract the concurrently-running threads in order to avoid the combinatorial explosion that results from considering each possible interleaving of commands individually. For sequential modules, we must abstract clients' actions between module calls because we cannot know what clients will do. To verify sequential modules, we redeploy RGSep's 'shared' and 'thread-local' heaps to model the module's internal heap and, respectively, the heaps of its clients.

Consider a module M with several routines. A client first calls `init`(), which prepares part of M's state for this client, and may transfer ownership of some of M's heap cells. The return value x identifies subsequent calls in this sequence. The client then invokes some other routines of M – passing x as a parameter each time – before calling `finalise`(x) so that its parts of M's state can later be used for another client. We use 'client' here to refer to a sequence of calls parameterised on the same x.

The crux is to show that several interleaved clients can all interact with M safely. For instance: if one client executes `x := init`(), then another executes `y := init`() followed by a sequence of calls parameterised on `y`, can the first client be sure that M is still in a state of readiness for a sequence of calls parameterised on `x`, and that the intervening events have not affected its part of M's state?

This is actually a matter of stability: we are seeking to prove that the post-condition of `x := init`() is stable under an environment that can execute M's routines arbitrarily (excepting those parameterised on `x`). We need only consider an environment that calls M's routines: other activities do not affect M's internal state, so can be deemed local.

To define such an environment, we require `x := init`() to return a *token*(`x`) predicate, to reside in the client's local heap. The predicate is *abstract* [16], which means that its definition is out of scope. Later module calls by this client (which we name $C_{\mathtt{x}}$) shall require the token's presence in its local heap, and the `finalise`(`x`) call shall confiscate it. The postcondition of `x := init`() is thus of the form $\boxed{P(\mathtt{x})} * \mathit{token}(\mathtt{x})$, where $P(\mathtt{x})$ describes an internal heap with a part initialised for $C_{\mathtt{x}}$. Let G be the set of RGSep actions by which M's routines can mutate its internal heap. Alone, $\boxed{P(\mathtt{x})}$ is not stable under G, for G includes actions that mutate $C_{\mathtt{x}}$'s part of the internal heap. Yet it becomes stable when combined with the local assertion *token*(`x`). Why? Because the presence of the *token*(`x`) in $C_{\mathtt{x}}$'s local state prohibits any *other* client having it and thus being able to continue the sequence of calls parameterised on `x`. It is vital that our refined notion of stability considers such conflicts between local and shared heaps

(Definition 8). Since stability occupies such a central role here, perhaps explicit stabilisation can be usefully applied? It can, in the following two ways.

Clarifying the stable parts of assertions. We have claimed $\boxed{P(\mathtt{x})} * \mathit{token}(\mathtt{x})$ to be a suitable – and stable – postcondition for `init`. Using explicit stabilisation, we now propose $\lfloor \boxed{P(\mathtt{x})} * \mathit{token}(\mathtt{x}) \rfloor_G$ instead. Strengthening the postcondition in this way *is* sound here, because the stabilisation has no effect on the already-stable assertion. Thus, the presence of $\lfloor\ \rfloor$ operators in the postcondition (and, dually, $\lceil\ \rceil$ in the precondition) serves to assert that their operands are stable. (In fact, $p \Leftrightarrow \lfloor p \rfloor_R$ exactly characterises those assertions that are stable under R.) We arrive at the following prototype specification:

$$G \vdash \left\{ \lceil \boxed{P} \rceil_G \right\} \quad \mathtt{x := init()} \quad \left\{ \lfloor \boxed{P(\mathtt{x})} * \mathit{token}(\mathtt{x}) \rfloor_G * Q \right\}.$$

We omit here and henceforth the rely from specifications, there being only one thread. We retain the guarantee, whose abstraction of the module calls that the thread may make is utilised by the Frame-S rule. The unparameterised P describes any valid internal heap of the module. See how the assertion Q, which describes cells that are transferred into the client's local heap, can be added outside the stabilised part: a client can mutate this part of the heap without concern for stability, the changes being purely local (see Lem. 9). Not all local changes can be treated so flippantly – indeed, the local assertion $\mathit{token}(\mathtt{x})$ is crucial to stability – but by delimiting the important assertions with the stabilisation syntax, we certify exactly which bits can and cannot be touched. Clients who obey this can be free of stability considerations, and instead rely on general properties of stabilisation, such as those detailed in Sect. 3.1.

Information hiding. Because the clients need not perform stabilisation, they need not even know the set of actions under which the assertions must be stable. That is, the definition of G can be kept internal to the module. This observation inspires the following proof rule.

$$\textsc{InfoHiding}\ \frac{\begin{array}{lll} \text{Module:} & & \left(\Delta, G \vdash \{p_i\}\ \langle C_i \rangle\ \{q_i\}\right)_{i=0}^{n} \\ \text{Client:} & \Delta' \subseteq \Delta & \Delta', \left(G \vdash \{p_i\}\ f_i\ \{q_i\}\right)_{i=0}^{n}, G \vdash \{p\}\, C\, \{q\} \end{array}}{\text{Whole system:} \qquad \vdash \{p\}\, \mathtt{let}\ (f_i = C_i)_{i=0}^{n}\ \mathtt{in}\ C\, \{q\}}$$

The rule concerns a sequential module comprising routines f_1 to f_n with implementations C_1 to C_n. The first line specifies each routine, in which G is the set of actions that clients of the module can perform. (In order to be able to access the module's internal heap, RGSep requires C_i to appear in angled brackets.) Δ denotes a set of predicate definitions, including the definition of *token* for instance. It also includes the definition of G, which we shall treat as an abstract predicate too. The second line specifies a client of the module, C. The Δ' it uses excludes the definitions of any predicates that are to remain abstract, and crucially, omits G's definition. Doing so makes the specification more reusable

– even in the event that G changes – and hence more conducive to modular reasoning. Explicit stabilisation is vital here: the stabilisation operations in the p_i's and q_i's refer to a particular G in the module specifications, and an arbitrary G in the client specification.

Theorem 12. *The* InfoHiding *rule is sound.*

Proof. The only departure from a typical rule for `let` commands is to remove G's definition from the client's specification, which logically strengthens one of the rule's assumptions.

6 Case study: Verification of a memory manager

We now reify the concepts of Sect. 5 by verifying the Version 7 Unix memory manager. This illustrates both our extension of explicit stabilisation to RGSep, and the use of the InfoHiding rule to hide a sequential module's internal interference from its clients. The verification itself is not only believed to be the first for this program; it also reveals a latent bug. The proof is one of safety: we prove neither termination nor that blocks are allocated in any particular fashion.

To begin, consider the following natural specifications, from [16], for `malloc` and `free`. Assume `malloc` cannot fail, and suppose a word is `WORD` bytes long.

$$\{\mathtt{emp}\}\ \ \mathtt{x} := \mathtt{malloc}(\mathtt{n} \times \mathtt{WORD})\ \ \{\mathit{token}(\mathtt{x},\mathtt{n}) * \mathtt{x}{\mapsto}\_ * \cdots * \mathtt{x}{+}\mathtt{n}{-}1{\mapsto}\_\}$$
$$\{\exists n.\, \mathit{token}(\mathtt{x},n) * \mathtt{x}{\mapsto}\_ * \cdots * \mathtt{x}{+}n{-}1{\mapsto}\_\}\ \ \mathtt{free}(\mathtt{x})\ \ \{\mathtt{emp}\}$$

The `malloc` routine gives each client an abstract *token* predicate, which the client later uses to certify to `free` that the block being returned was truly allocated by `malloc` (`free`'s behaviour being undefined otherwise). These specifications could be realised naïvely by implementing $\mathit{token}(x,n)$ as $x{-}1{\mapsto}n$; that is, by storing the length of each block in the preceding cell.

Real memory managers are far more complex. The one we shall examine forms the cells that precede each block into a monotonically-increasing chain of pointers, linking all the allocated and free blocks. Such a manager must maintain in its internal heap the pointer chain, plus any free blocks, while the allocated blocks are conceptually held by each respective client. For a token, we can now afford only *half* of the cell preceding the block, because the manager must retain at least read-permission on this cell for later traversals of the pointer chain. Note that by creating the token from part of the existing datastructure, our proof avoids the need for auxiliary state.

The crux of the verification is to prove that a block allocated to a client remains allocated until, and only until, that client frees it; that is, it is not invalidated by other calls to `malloc` and `free`. Defining G as the set of actions of `malloc` and `free`, we are asking if `malloc`'s postcondition is stable under G.

It is easy to show that it is unaffected when these actions are applied to blocks other than the current one. And although the environment is *allowed* to apply these actions to the current block, it is actually *unable* to do so. Why? Because

$$\mathsf{G} \vdash \left\{ \lceil \boxed{\mathit{arena}} \rceil_{\mathsf{G}} \right\} \ \mathtt{x := malloc(n \times WORD)} \ \left\{ \begin{array}{r} \lfloor \boxed{\mathit{arena}} \rfloor_{\mathsf{G}} * \lfloor \mathit{arenatoken}(\mathtt{x}, \mathtt{n}) \rfloor_{\mathsf{G}} \\ * \ \mathtt{x} \mapsto \_ * \cdots * \mathtt{x+n-1} \mapsto \_ \end{array} \right\}$$

$$\mathsf{G} \vdash \left\{ \begin{array}{r} \exists n. \lceil \boxed{\mathit{arena}} \rceil_{\mathsf{G}} * \lceil \mathit{arenatoken}(\mathtt{x}, n) \rceil_{\mathsf{G}} \\ * \ \mathtt{x} \mapsto \_ * \cdots * \mathtt{x}+n-1 \mapsto \_ \end{array} \right\} \ \mathtt{free(x)} \ \left\{ \lfloor \boxed{\mathit{arena}} \rfloor_{\mathsf{G}} \right\}$$

Fig. 10: Specifications of `malloc` and `free`

$\mathsf{G} \vdash \left\{ \lceil \boxed{\mathit{arena}} \rceil_{\mathsf{G}} \right\}$
`x := malloc(2*WORD);`
3 $\left\{ \lfloor \boxed{\mathit{arena}} \rfloor_{\mathsf{G}} * \lfloor \mathit{arenatoken}(\mathtt{x}, 2) \rfloor_{\mathsf{G}} * \mathtt{x} \mapsto \_, \_ \right\}$
$\Longrightarrow \left\{ \lceil \boxed{\mathit{arena}} \rceil_{\mathsf{G}} * \lfloor \mathit{arenatoken}(\mathtt{x}, 2) \rfloor_{\mathsf{G}} * \mathtt{x} \mapsto \_, \_ \right\}$
`y := malloc(3*WORD);`
6 $\left\{ \lfloor \boxed{\mathit{arena}} \rfloor_{\mathsf{G}} * \lfloor \mathit{arenatoken}(\mathtt{x}, 2) \rfloor_{\mathsf{G}} * \mathtt{x} \mapsto \_, \_ * \lfloor \mathit{arenatoken}(\mathtt{y}, 3) \rfloor_{\mathsf{G}} * \mathtt{y} \mapsto \_, \_, \_ \right\}$
`[y+1] := 7;`
$\left\{ \lfloor \boxed{\mathit{arena}} \rfloor_{\mathsf{G}} * \lfloor \mathit{arenatoken}(\mathtt{x}, 2) \rfloor_{\mathsf{G}} * \mathtt{x} \mapsto \_, \_ * \lfloor \mathit{arenatoken}(\mathtt{y}, 3) \rfloor_{\mathsf{G}} * \mathtt{y} \mapsto \_, 7, \_ \right\}$
9 $\Longrightarrow \left\{ \lceil \boxed{\mathit{arena}} \rceil_{\mathsf{G}} * \lceil \mathit{arenatoken}(\mathtt{x}, 2) \rceil_{\mathsf{G}} * \mathtt{x} \mapsto \_, \_ * \lfloor \mathit{arenatoken}(\mathtt{y}, 3) \rfloor_{\mathsf{G}} * \mathtt{y} \mapsto \_, 7, \_ \right\}$
`free(x);`
$\left\{ \lfloor \boxed{\mathit{arena}} \rfloor_{\mathsf{G}} * \lfloor \mathit{arenatoken}(\mathtt{y}, 3) \rfloor_{\mathsf{G}} * \mathtt{y} \mapsto \_, 7, \_ \right\}$

Fig. 11: Verification of a simple client

the current block cannot be accidentally re-allocated, since to do so would give the client a duplicate token, which the $*$ operator forbids. And neither can it be accidentally freed, without yielding its token.

Using explicit stabilisation, here is a first attempt to specify `malloc`:

$$\mathsf{G} \vdash \left\{ \lceil \boxed{\mathit{arena}} \rceil_{\mathsf{G}} \right\} \ \mathtt{x := malloc(n \times WORD)} \ \left\{ \begin{array}{l} \lfloor \boxed{\mathit{arena}(\mathtt{x}, \mathtt{n})} * \mathit{token}(\mathtt{x}, \mathtt{n}) \rfloor_{\mathsf{G}} \\ * \ \mathtt{x} \mapsto \_ * \cdots * \mathtt{x+n-1} \mapsto \_ \end{array} \right\}$$

The *arena* predicate asserts that the manager's internal heap is valid, while $\mathit{arena}(x, n)$ additionally asserts that the block at x is missing. Note that the stability of $\boxed{\mathit{arena}(x, n)}$ relies on the $\mathit{token}(x, n)$ predicate in the local heap.

This specification exposes too much of the manager's innards. We address this in the improved specifications in Fig. 10, by collapsing $\boxed{\mathit{arena}(x, n)} * \mathit{token}(x, n)$ into a single abstract predicate, $\mathit{arenatoken}(x, n)$. We also append the $\boxed{\mathit{arena}}$ predicate to both `malloc`'s postcondition and `free`'s precondition. Strictly, this is redundant, for $\boxed{\mathit{arena}}$ is entailed by *arenatoken*, but having `malloc`'s postcondition reestablish its precondition simplifies the verification of successive calls to `malloc` and allows the predicates to remain fully abstract.

Now consider the simple client in Fig. 11. Because the *content* of the block lies outside the scope of the stabilisation, the client can mutate it (line 7) without having to reconsider stability. The allocation of the block at `y` (line 5) does not affect the block at `x`: such a deduction is enabled by the Frame-S rule of Fig. 9. (Although this rule imposes a stabilisation on the entire frame, we can leave this implicit for the local parts, by Lem. 9.) See how the use of explicit stabilisation

allows the client's verifier to rely only on general properties of stabilisation: for instance, the deduction of the assertion on line 4 follows straight from $\lfloor p \rfloor_R \Rightarrow p \Rightarrow \lceil p \rceil_R$. The definition of G is thus not needed by the client, so we can use our InfoHiding rule to keep it internal to the module.

The rest of this section concerns the implementation (Sect. 6.1) and verification (Sect. 6.2) of the memory manager. The source code is provided in Appx. A; our full proof is in [21]. We omit an optimisation that tells `malloc` where to begin its search, because it contains a bug, which we explain in Sect. 6.3. Section 6.4 describes some peripheral details of the implementation and the verification.

6.1 Implementation of the memory manager

The memory manager controls the allocation and deallocation of blocks of main memory to and from client processes. The portion of memory it controls (shown in Fig. 12) contains both free and allocated blocks. The grey cells form a cyclic chain of pointers and the white blocks in between can be allocated to clients. Since blocks are word-aligned, the least significant bit in each pointer is redundant, and is hence employed to signal the availability of the following block. In the figure, black and white squares indicate that this so-called 'busy' bit is set and, respectively, unset. The module-level variables `s` and `t` respectively identify the first and last pointers in the arena. Because it is not followed by an allocatable block, the last pointer's busy bit is permanently set.

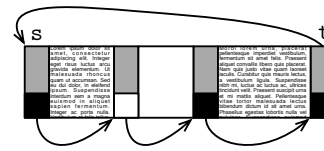

Fig. 12: An arena

A client requests a block of n bytes by calling `malloc`(n). For clarity of exposition we shall keep n a multiple of the word size, `WORD`. The routine traverses pointers until it finds a free block that is sufficiently large, returning the null pointer in the case of failure. It coalesces consecutive free blocks throughout the search. Should the block it finds be exactly the right size, a pointer to it is returned, and should it be too large, it is divided into two and a pointer to the first is returned. The client can later invoke `free`(x), x being the address of the first cell in the block. Observe that `free` is not parameterised by the length of the block, because the length was recorded when `malloc` allocated it.

6.2 Details of the verification

Figure 13 defines some auxiliary predicates used in the specifications and proof. $x \underset{\mathsf{u}}{\rightarrow} y$ describes an unallocated block between x and y. Upon being allocated a block of size n with first cell x, the client is also given $\mathit{token}(x, n)$, which contains a half permission on the block's pointer; the manager retains the other half. We write $x \mapsto y_{\mathrm{busy}}$ to mean that upon unsetting x's busy bit, it would contain the address of y. $x \curvearrowright y$ says that y is the special pointer at the end of the arena that points back to x, the start of the arena. $x \twoheadrightarrow y$ denotes a possibly-empty monotonically-increasing chain of pointers from x to y (including any unallocated blocks), the definition of which abbreviates $x \rightarrow x' * x' \twoheadrightarrow y$ to $x \rightarrow x' \twoheadrightarrow y$.

$$
\begin{array}{rcl}
x \xrightarrow[\mathtt{u}]{} y & \stackrel{\text{def}}{=} & x{<}y \wedge x{\mapsto}y * (x{+}1){\mapsto}\_ * \cdots * (y{-}1){\mapsto}\_ \\
x \xrightarrow[\mathtt{a}]{} y & \stackrel{\text{def}}{=} & x{<}y \wedge x \overset{.5}{\mapsto} y_{\text{busy}} \\
x \rightarrow y & \stackrel{\text{def}}{=} & x \xrightarrow[\mathtt{u}]{} y \vee x \xrightarrow[\mathtt{a}]{} y \\
x \curvearrowright y & \stackrel{\text{def}}{=} & x{<}y \wedge y{\mapsto}x_{\text{busy}} \\
x \twoheadrightarrow y & \stackrel{\text{def}}{=} & (\exists x'.\, x \rightarrow x' \twoheadrightarrow y) \vee (x{=}y \wedge \mathsf{emp}) \\
\mathit{arena} & \stackrel{\text{def}}{=} & \mathtt{s} \twoheadrightarrow \mathtt{t} * \mathtt{s} \curvearrowright \mathtt{t} \\
\mathit{arena}(x,n) & \stackrel{\text{def}}{=} & \mathtt{s} \twoheadrightarrow (x{-}1) \xrightarrow[\mathtt{a}]{} (x{+}n) \twoheadrightarrow \mathtt{t} * \mathtt{s} \curvearrowright \mathtt{t} \\
\mathit{token}(x,n) & \stackrel{\text{def}}{=} & (x{-}1) \overset{.5}{\mapsto} (x{+}n)_{\text{busy}} \\
\mathit{arenatoken}(x,n) & \stackrel{\text{def}}{=} & \boxed{\mathit{arena}(x,n)} * \mathit{token}(x,n)
\end{array}
$$

Fig. 13: Predicates

Coalesce:	$a \xrightarrow[\mathtt{u}]{} b \xrightarrow[\mathtt{u}]{} c$	$\rightsquigarrow$	$a \xrightarrow[\mathtt{u}]{} c$	$\mid$	$\mathtt{s} \twoheadrightarrow a$
AllocateEntire:	$a \xrightarrow[\mathtt{u}]{} b$	$\rightsquigarrow$	$a \xrightarrow[\mathtt{a}]{} b$	$\mid$	$\mathtt{s} \twoheadrightarrow a$
AllocatePart:	$a \xrightarrow[\mathtt{u}]{} b$	$\rightsquigarrow$	$a \xrightarrow[\mathtt{a}]{} (b{-}n) \xrightarrow[\mathtt{u}]{} b$	$\mid$	$\mathtt{s} \twoheadrightarrow a$
Free:	$a \xrightarrow[\mathtt{a}]{} b$	$\rightsquigarrow$	$a \xrightarrow[\mathtt{u}]{} b$	$\mid$	$\mathtt{s} \twoheadrightarrow a$

Fig. 14: Main actions

Figure 14 formalises the ways in which the internal heap of the module may be mutated by clients calling `malloc` and `free`. Only one routine can execute at once, so it would suffice to list a single action for each. We prefer to split them into several simple actions. The first coalesces two consecutive free blocks. The second allocates an entire block to a client, while the third allocates just the initial part. The fourth frees a block. The context $\mathtt{s} \twoheadrightarrow a$ ensures that the blocks that are acted upon are really in the arena. G is the union of all these actions.

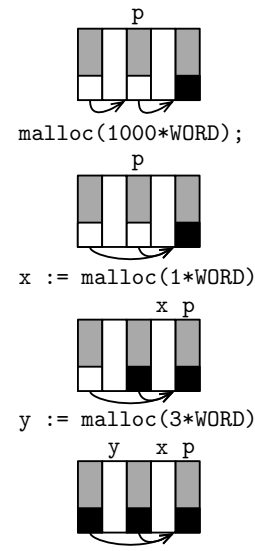

Fig. 15: The bug

6.3 A (faulty) optimisation

The following bug was discovered during the verification process.

The manager maintains a global variable `p` (named `allocp` in the original source code) that, after a block is allocated, is pointed to the successive block, and after a block is freed, is pointed to that block. It serves to identify a good place for the next call to `malloc` to begin its search. The implementation does not update `p` if allocation fails, however, and therein lies the bug: `p` *should* be updated in case the block to which it points has been coalesced with its predecessor, lest it be left pointing inside a block.

Figure 15 demonstrates how this bug could wreak havoc. Our contrived arena contains just two one-word blocks, both of which are free, and `p` initially points

to the second. The first `malloc` call fails, but has the side-effect of leaving `p` inside the coalesced block. We then allocate a small block at `x`, before wrapping around to the start of the arena and allocating a larger block at `y`, thereby reaching a situation in which the contents of the smaller block is allocated twice.

The discovery of this bug was prompted by the failure of the invariant $\mathtt{s}{\twoheadrightarrow}\mathtt{p}$, which states that `p` identifies a valid pointer in the arena. We have successfully executed our exploit to confirm that the bug is real.

6.4 Other issues

There are several other issues involved in the implementation and verification of the memory manager, which we explain now. These issues have been sidestepped so far in order to focus on the crucial parts of the verification.

Allocation failure. To handle the case where `malloc` fails, its postcondition should be disjoined with the following assertion: $\lfloor \boxed{\mathit{arena}} \rfloor_{\mathsf{G}} * \mathtt{x}{=}0$.

Extending the arena. Once the search for a block has exhausted the arena, `malloc` invokes `sbrk` to ask the system for another block of memory. This block will be located at an address above `t` because, in Version 7 Unix, memory allocated via `sbrk` is never returned. The following three actions should be added to G, to formalise these calls to `sbrk`:[4]

$$\begin{array}{ll}
\mathsf{ExtendGap}: & \mathtt{\&t}{\mapsto}t * \mathtt{s}^{\curvearrowleft}t * \mathit{brk}(b) \wedge b{>}t{+}1 \;\rightsquigarrow\; \mathtt{\&t}{\mapsto}t' * t \xrightarrow{\mathtt{a}} b \xrightarrow{\mathtt{u}} t' * \mathtt{s}^{\curvearrowleft}t' * \mathit{brk}(t'{+}1) \\
\mathsf{ExtendNoGap}: & \mathtt{\&t}{\mapsto}t * \mathtt{s}^{\curvearrowleft}t * \mathit{brk}(t{+}1) \;\rightsquigarrow\; \mathtt{\&t}{\mapsto}t' * t \xrightarrow{\mathtt{u}} (t{+}1) \xrightarrow{\mathtt{u}} t' * \mathtt{s}^{\curvearrowleft}t' * \mathit{brk}(t'{+}1) \\
\mathsf{AdvBreak}: & \mathit{brk}(b) \;\rightsquigarrow\; \exists n{>}0.\, \mathit{brk}(b{+}n)
\end{array}$$

The first extends the arena with a new block, leaving a gap that is filled with an unfreeable dummy block to maintain the illusion of a contiguous arena. The second is similar, but without the gap. The third action, which advances the 'break value' (the cell at which the next successful call to `sbrk` will return a block), is kept distinct to reflect that it may be performed in other situations.

An issue with dummy blocks. When the arena is extended via the ExtendGap action, the resulting gap is filled with a dummy block that is permanently allocated. In order to allocate such a block, we need to hand the caller the *token* predicate, yet there is no client in this situation. We thus add a true predicate to the arena, which can 'soak up' these spare tokens. Considering this and the previous points, the arena (see Fig. 13) can be more precisely defined as follows:

$$\mathit{arena} \;\stackrel{\text{def}}{=}\; \exists s, t, b.\, \mathsf{true} * \mathtt{\&s}{\mapsto}s * \mathtt{\&t}{\mapsto}t * s{\twoheadrightarrow}t * s^{\curvearrowleft}t * \mathit{brk}(b) \wedge t < b$$

7 Related Work

Explicit stabilisation arose out of 'mid stability' [19, §4.1], a variation of RG reasoning that places stability checks not on the pre and postconditions of basic

[4] We are now treating module-level variables more carefully: the variable `t` is modelled as a heap cell at address `&t`, thus allowing its value to be altered by these actions.

commands, but at the points of sequential and parallel composition instead. This more strategic placement eliminates redundant checks, and also allows libraries comprising just one basic command to be verified without considering stability. Our parametric proof system (Fig. 4) extends this to *all* library functions (and encodes mid stability soundly and completely).

RG-style reasoning has been used before to verify concurrent library code (e.g. [10]). The specifications of that approach involve a particular rely, whereas our parametric specifications do not require a particular rely to be instantiated.

RG has also furnished proofs of sequential modules before (e.g. [22]), but we believe ours to be the first that hides the module's internal interference. The InfoHiding rule that enables this feat is related to the hypothetical frame rule [15]: the latter rule hides the module's *state* from the client, while ours hides the module's *interference*. Perhaps the hypothetical frame rule could be used to remove the $\boxed{arena}$ predicate from the verification given in Fig. 11, thus revealing to the client neither the module's state nor its internal interference.

SAGL [9], like RGSep, is a descendant of RG and separation logic, to which explicit stabilisation could also be applied. Local Rely-Guarantee (LRG) [8] is a third descendant that addresses an inherent flaw in the modularity of its siblings: that the shared heap must be globally known. It defines a $*$ operator over interference, which allows the shared heap to be split into portions that are shared between just a few threads. The application of explicit stabilisation to LRG could simplify the verification of clients that invoke multiple modules, for our approach currently handles only one.

Explicit stabilisation can be seen as a bridge between theory and implementation: tools, such as SmallfootRG [4], that automate RG-style reasoning may defer stability checks rather than perform them at the point of rule application, and explicit stabilisation can help to formalise this 'lazy' approach. We have not considered the implementation of stabilisation; this issue is explored in [2].

8 Conclusion

We have proposed explicit stabilisation as a new way to deal with stability in RG reasoning. The central idea is to record information about an assertion's stability into its syntactic form. The main benefits are in modular reasoning:

Library code can be verified independently of clients. In Sect. 4, we showed how an approach based upon explicit stabilisation enables RG reasoning to verify concurrent library code. Essentially, the stabilisation in the library's specification is evaluated so lazily that it actually becomes an obligation of the client.

Client code can be verified independently of a sequential module. We showed in Sect. 5 how the application of explicit stabilisation to RGSep gives rise to an InfoHiding rule that allows a sequential module to hide its internal interference from its clients. Such information hiding is crucial for modular reasoning, because it allows the specification of a client to be reused, even despite changes to the specification of this internal interference. Section 6 demonstrated this reasoning by verifying a memory manager.

It would be interesting to investigate whether these two forms of modularity can be combined; that is, can we verify both a library and its clients, modularly, at the same time? It looks feasible. The specification for the library in Sect. 4 used explicit stabilisation with an arbitrary rely R, which became specific for each client in turn. Meanwhile, the specifications for the memory manager in Sect. 6 used explicit stabilisation with the specific G of the module, which was then generalised to an arbitrary G for the clients, so as to provide information hiding. Perhaps a combination of these approaches would parameterise on both the rely and the guarantee?

We also plan to apply explicit stabilisation to more advanced logics based on RG, such as LRG, Deny-Guarantee [7], and the logic of Gotsman et al. for proving liveness [11]. The notions of stability in such logics are becoming ever more demanding, so it is increasingly important to have a solid basis upon which to reason about stability. We believe explicit stabilisation provides such a basis.

Acknowledgements

The idea of parameterising RG specifications on the 'current rely' is due to Hongseok Yang. Richard Bornat introduced us to the malloc example. We also thank Joey Coleman, Xinyu Feng, Erica Fulbrook, Cliff Jones, Alexander Malkis, Tom Ridge and Viktor Vafeiadis for feedback and helpful discussions. This work was supported by EPSRC grant F019394/1. Parkinson is supported by a Royal Academy of Engineering/EPSRC fellowship.

References

1. M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit substitutions. In *POPL*, 1990.
2. H. Amjad and R. Bornat. Towards automatic stability analysis for rely-guarantee proofs. In *VMCAI*, 2009.
3. R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson. Permission accounting in separation logic. In *POPL*, 2005.
4. C. Calcagno, M. Parkinson, and V. Vafeiadis. Modular safety checking for fine-grained concurrency. In *SAS*, 2007.
5. J. W. Coleman. Expression decomposition in a Rely/Guarantee context. In *VSTTE*, 2008.
6. E. W. Dijkstra. *A Discipline of Programming.* Prentice Hall, Inc., 1976.
7. M. Dodds, X. Feng, M. Parkinson, and V. Vafeiadis. Deny-Guarantee reasoning. In *ESOP*, 2009.
8. X. Feng. Local rely-guarantee reasoning. In *POPL*, 2009.
9. X. Feng, R. Ferreira, and Z. Shao. On the relationship between concurrent separation logic and assume-guarantee reasoning. In *ESOP*, 2007.
10. C. Flanagan, S. N. Freund, S. Qadeer, and S. A. Seshia. Modular verification of multithreaded programs. *Theor. Comput. Sci.*, 338(1-3), 2005.
11. A. Gotsman, B. Cook, M. Parkinson, and V. Vafeiadis. Proving that non-blocking algorithms don't block. In *POPL*, 2009.

12. C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10), 1969.
13. S. S. Ishtiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. In *POPL*, 2001.
14. C. B. Jones. *Development methods for computer programs including a notion of interference*. PhD thesis, University of Oxford, 1981.
15. P. W. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. In *POPL*, 2004.
16. M. Parkinson and G. Bierman. Separation logic and abstraction. In *POPL*, 2005.
17. L. Prensa Nieto. The Rely-Guarantee method in Isabelle/HOL. In *ESOP*, 2003.
18. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, 2002.
19. V. Vafeiadis. *Modular fine-grained concurrency verification*. PhD thesis, University of Cambridge, 2007.
20. V. Vafeiadis and M. Parkinson. A marriage of Rely/Guarantee and separation logic. In *CONCUR*, 2007.
21. J. Wickerson, M. Dodds, and M. Parkinson. Explicit Stabilisation for Modular Rely-Guarantee Reasoning. Technical report, University of Cambridge, 2010.
22. G. Yorsh, A. Skidanov, T. Reps, and M. Sagiv. Automatic assume/guarantee reasoning for heap-manipulating programs: Ongoing work. In *AIOOL*, 2005.

A Source code of Unix V7 memory manager

Abridged and corrected. Retrieved from the Unix Heritage Society.[5]

```
#define WORD sizeof(st)
#define BLOCK 1024
#define testbusy(p) ((int)(p)&1)
#define setbusy(p) (st *)((int)(p)|1)
#define clearbusy(p) (st *)((int)(p)&~1)
struct store { struct store *ptr; };
typedef struct store st;
static st s[2]; /*initial arena*/
// static struct store *allocp; (bug removed)
static st *t;   /*arena top*/
char* sbrk();
char* malloc(unsigned nbytes) {
  register st *p, *q;
  register nw;  static temp;
  // omitted: initialisation code
  nw = (nbytes+WORD+WORD-1)/WORD;
  for(p=s; ; ) {
    for(temp=0; ; ) {
      if(!testbusy(p->ptr)) {
        while(!testbusy((q=p->ptr)->ptr))
          p->ptr = q->ptr;
        if(q>=p+nw && p+nw>=p) goto found;
      }
      q = p; p = clearbusy(p->ptr);
      if(p>q) ;
      else if(q!=t || p!=s) return 0;
      else if(++temp>1) break;
    }
    temp = ((nw+BLOCK/WORD)
              /(BLOCK/WORD))*(BLOCK/WORD);
    q = (st *)sbrk(0);
    if(q+temp < q) return 0;
    q = (st *)sbrk(temp*WORD);
    if((int)q == -1) return 0;
    t->ptr = q;
    if(q!=t+1) t->ptr = setbusy(t->ptr);
    t = q->ptr = q+temp-1;
    t->ptr = setbusy(s);
  }
found:
  if(q>p+nw) ((st *)(p+nw))->ptr = p->ptr;
  p->ptr = setbusy(p+nw);
  return((char *)(p+1));
}
free(register char *ap) {
  register st *p = ((st *)ap)-1;
  p->ptr = clearbusy(p->ptr);
}
```

[5] http://minnie.tuhs.org/UnixTree/V7/usr/src/libc/gen/malloc.c.html