Deny-Guarantee Reasoning

Mike Dodds[1], Xinyu Feng[2], Matthew Parkinson[1], and Viktor Vafeiadis[3]

[1] University of Cambridge, UK
[2] Toyota Technological Institute at Chicago, USA
[3] Microsoft Research Cambridge, UK

Abstract. Rely-guarantee is a well-established approach to reasoning about concurrent programs that use parallel composition. However, parallel composition is not how concurrency is structured in real systems. Instead, threads are started by 'fork' and collected with 'join' commands. This style of concurrency cannot be reasoned about using rely-guarantee, as the life-time of a thread can be scoped dynamically. With parallel composition the scope is static.

In this paper, we introduce deny-guarantee reasoning, a reformulation of rely-guarantee that enables reasoning about dynamically scoped concurrency. We build on ideas from separation logic to allow interference to be dynamically split and recombined, in a similar way that separation logic splits and joins heaps. To allow this splitting, we use *deny* and *guarantee* permissions: a deny permission specifies that the environment cannot do an action, and guarantee permission allow us to do an action. We illustrate the use of our proof system with examples, and show that it can encode all the original rely-guarantee proofs. We also present the semantics and soundness of the deny-guarantee method.

1 Introduction

Rely-guarantee [10] is a well-established compositional proof method for reasoning about concurrent programs that use parallel composition. Parallel composition provides a structured form of concurrency: the lifetime of each thread is statically scoped, and therefore interference between threads is also statically known. In real systems, however, concurrency is not structured like this. Instead, threads are started by a 'fork' and collected with 'join' commands. The lifetime of such a thread is dynamically scoped in a similar way to the lifetime of heap-allocated data.

In this paper, we introduce *deny-guarantee* reasoning, a reformulation of rely-guarantee that enables reasoning about such dynamically scoped concurrency. We build on ideas from separation logic to allow interference to be dynamically split and recombined, in a similar way that separation logic splits and joins heaps.

In rely-guarantee, interference is described using two binary relations: the *rely*, R, and the *guarantee*, G. Specifications of programs consist of a precondition, a postcondition and an interference specification. This setup is sufficient to reason about lexically-scoped parallel composition, but not about dynamically-scoped threads. With dynamically-scoped threads, the interference at the end of the program may be quite different from the interference at the beginning of the program, because during execution other threads may have been forked or joined. Therefore, just as in Hoare logic

G. Castagna (Ed.): ESOP 2009, LNCS 5502, pp. 363–377, 2009.
© Springer-Verlag Berlin Heidelberg 2009

a program's precondition and postcondition may differ from each other, so in deny-guarantee logic a thread's pre-interference and post-interference specification may differ from each other.

Main results. The main contributions of this paper are summarized below:

- We introduce deny-guarantee logic and apply it to an example (see §3 and §4).
- We present an encoding of rely-guarantee into deny-guarantee, and show that every rely-guarantee proof can be translated into a deny-guarantee proof (see §5).
- We prove that our proof rules are sound (see §6).
- We have formalized our logic and all the proofs in Isabelle [4].

For clarity of exposition, we shall present deny-guarantee in a very simple setting where the memory consists only of a pre-allocated set of global variables. Our solution extends easily to a setting including memory allocation and deallocation (see §7).

Related work. Other work on concurrency verification has generally ignored fork/join, preferring to concentrate on the simpler case of parallel composition. This is true of all of the work on traditional rely-guarantee reasoning [10, 11]. This is unsurprising, as the development of deny-guarantee depends closely on the abstract characterization of separation logic [3]. However, even approaches such as SAGL [5] and RGSep [12] which combine rely-guarantee with separation logic omit fork/join from their languages.

There exist already some approaches to concurrency that handle fork. Feng *et al.* [6] and Hobor *et al.* [9] both handle fork. However, both omit join with the justification that it can be handled by synchronization between threads. However, this approach is not compositional: it forces us to specify interference globally. Gotsman *et al.* [7] propose an approach to locks in the heap which includes both fork and join. However, this is achieved by defining an invariant over protected sections of the heap, which makes compositional reasoning about inter-thread interference impossible (see the next section for an example of this). Haack and Hurlin [8] have extended Gotsman *et al.*'s work to reason about fork and join in Java, where a thread can be joined multiple times.

2 Towards Deny-Guarantee Logic

Consider the very simple program given in Fig. 1. If we run the program in an empty environment, then at the end, we will get $\mathtt{x} = 2$. This happens because the main thread will block at line `L3` until thread `t1` terminates. Hence, the last assignment to `x` will either be that of thread `t2` or of the main thread, both of which write the value 2 into `x`. We also know that the `error` in the forked code on `L1` and `L2` will never be reached.

```
L0: x := 0;
L1: t1 := fork(if(x==1) error;
               x := 1);
L2: t2 := fork(x := 2;
               if (x==3) error);
L3: join t1;
L4: x := 2;
L5: join t2;
```

Fig. 1. Illustration of fork/join

Now, suppose we want to prove that this program indeed satisfies the postcondition $\mathtt{x} = 2$. Unfortunately, this is not possible with existing compositional proof methods.

Invariant-based techniques (such as Gotsman *et al.* [7]) cannot handle this case, because they cannot describe interference. Unless we introduce auxiliary state to specify a more complex invariant, we cannot prove the postcondition, as it does not hold throughout the execution of the program.

Rely-guarantee can describe interference, but still cannot handle this program. Consider the parallel rule:

$$\frac{R_1, G_1 \vdash \{P_1\}\, C_1\, \{Q_1\} \quad G_1 \subseteq R_2 \quad R_2, G_2 \vdash \{P_2\}\, C_2\, \{Q_2\} \quad G_2 \subseteq R_1}{R_1 \cap R_2, G_1 \cup G_2 \vdash \{P_1 \wedge P_2\}\, C_1 \parallel C_2\, \{Q_1 \wedge Q_2\}}$$

In this rule, the interference is described by the rely, R, which describes what the environment can do, and the guarantee, G, which describes what the code is allowed to do. The rely and guarantee do not change throughout the execution of the code, they are 'statically scoped' interference, whereas the scope of the interference introduced by `fork` and `join` commands is dynamic.

Separation logic solves this kind of problem for dynamically allocated memory, also known as the heap. It uses the star operator to partition the heap into heap portions and to pass the portions around dynamically. The star operator on heaps is then lifted to assertions about heaps. In this work, we shall use the star operator to partition the *interference* between threads, and then lift it to assertions about the interference.

Let us assume we have an assertion language which can describe interference. It has a separation-logic-like star operation. We would like to use this star to split and join interference, so that we can use simple rules to deal with `fork` and `join`:

$$\frac{\{P_1\}\, C\, \{P_2\} \quad \ldots}{\{P * P_1\}\, x := \textbf{fork}\ C\, \{P * \mathsf{Thread}(x, P_2)\}}\ (\textsc{fork}) \qquad \frac{\ldots}{\{P * \mathsf{Thread}(E, P')\}\, \textbf{join}\ E\, \{P * P'\}}\ (\textsc{join})$$

The FORK rule simply removes the interference, P_1, required by the forked code, C, and returns a token $\mathsf{Thread}(x, P_2)$ describing the final state of the thread. The JOIN rule, knowing the thread E is dead, simply takes over its final state[1].

Now, we will consider how we might prove our motivating example. Let us imagine we have some assertions that both allow us to do updates to the state, and forbid the environment from doing certain updates. We provide the full details in §4, and simply present the outline (Fig. 2) and an informal explanation of the permissions here. The first thread we fork can be verified using the T_1 and $\mathtt{x} \neq 1$, where T_1 allows us to update `x` to be `1`, and prevents any other thread updating `x` to be `1`. Next, we use G_2 which allows us to update `x` to be `2`; and D_3 which prevents the

$\{T_1 * G_2 * D_3 * L * \mathtt{x} \neq 1\}$
```
  t1 := fork (if(x==1) error;
              x := 1);
```
$\{G_2 * D_3 * L * \mathsf{Thread}(\mathtt{t1}, T_1)\}$
```
  t2 := fork (x := 2;
              if(x==3) error );
```
$\{L * \mathsf{Thread}(\mathtt{t1}, T_1) * \mathsf{Thread}(\mathtt{t2}, G_2 * D_3)\}$
```
  join t1;
```
$\{T_1 * L * \mathsf{Thread}(\mathtt{t2}, G_2 * D_3)\}$
```
  x := 2;
```
$\{T_1 * L * \mathsf{Thread}(\mathtt{t2}, G_2 * D_3) * \mathtt{x} = 2\}$
```
  join t2
```
$\{T_1 * G_2 * D_3 * L * \mathtt{x} = 2\}$

Fig. 2. Proof outline

[1] As in the pthread library, we allow a thread to be joined only once. We could also adapt the work of Haack and Hurlin [8] to our deny-guarantee setting to handle Java-style join.

environment from updating `x` to be `3`. These two permissions are sufficient to verify the second thread. Finally, L is a leftover permission which prevents any other thread updating `x` to be any value other than `1` or `2`. When we get to the assignment, we have $T_1 * L$ which forbids the environment performing any update except assigning `x` with `2`. Hence, we know that the program will terminate with `x` = 2.

Now, we consider how to build a logic to represent the permission on interference used in the proof outline. Let us consider the information contained in a rely-guarantee pair. For each state change it has one of four possibilities presented in Fig. 3: *guar* permission, allowed by both the thread and the environment (me,env); 1 permission, allowed by the thread, and not allowed for the environment (me,¬env); 0 permission, not allowed by the thread, but allowed by the environment (¬me,env); and *deny* permission, not allowed by the thread or the environment (¬me,¬env).

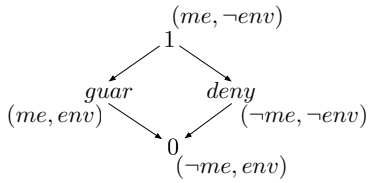

Fig. 3. Possible interference

To allow inter-thread reasoning about interference, we want to split full permissions 1 into either *deny* permissions or *guar* permissions. We also want to further split *deny*, or *guar*, permissions into smaller *deny* or *guar* permissions respectively. The arrows of Fig. 3 show the order of permission strength captured by splitting. If a thread has a *deny* on a state change, it can give another thread a *deny* and keep one itself while preserving the fact that the state change is prohibited for itself and the environment. The same holds for *guar*.

To preserve soundness, we cannot allow unrestricted copying of permissions – we must treat them as *resources*. Following Boyland [2] and Bornat *et al.* [1] we attach weights to splittable resources. In particular we use fractions in the interval (0,1). For example, we can split an $(a+b)$*deny* into an (a)*deny* and a (b)*deny*, and similarly for *guar* permissions. We can also split a full permission 1 into (a)*deny* and (b)*deny*, or (a)*guar* and (b)*guar*, where $a+b=1$.

In the following sections we will show how these permissions can be used to build deny-guarantee, a separation logic for interference.

Aside. Starting with the parallel composition rules of rely-guarantee and of separation logic, you might wonder if we can define our star as $(R_1,G_1)*(R_2,G_2) = (R_1 \cap R_2, G_1 \cup G_2)$ provided $G_1 \subseteq R_2$ and $G_2 \subseteq R_1$, and otherwise it is undefined. Here we have taken the way rely-guarantee combines the relations, and added it to the definition of $*$.

This definition, however, does not work. The star we have defined is not *cancellative*, a condition that is required for proving that separation is sound [3]. Cancellativity says that for all x, y and z, if $x*y$ is defined and $x*y = x*z$, then $y = z$. Intuitively, the problem is that $\cap$ and $\cup$ lose information about the overlap.

3 The Logic

Language. The language is defined in Fig. 4. This is a standard language with two additional commands for forking a new thread and for joining with an existing thread.

$$
\begin{array}{rrcl}
\text{(Expr)} & E & ::= & x \mid n \mid E + E \mid E - E \mid \ldots \\
\text{(BExp)} & B & ::= & \mathsf{true} \mid \mathsf{false} \mid E = E \mid E \neq E \mid \ldots \\
\text{(Stmts)} & C & ::= & x := E \mid \textbf{skip} \mid C;C \mid \textbf{if } B \textbf{ then } C \textbf{ else } C \mid \textbf{while } B \textbf{ do } C \mid x := \textbf{fork } C \mid \textbf{join } E
\end{array}
$$

Fig. 4. The Language

Informally, the $x :=$ **fork** C command allocates an unused thread identifier t, creates a new thread with thread identifier t and body C, and makes it run in parallel with the rest of the program. Finally, it returns the thread identifier t by storing it in x. The command **join** E blocks until thread E terminates; it fails if E is not a valid thread identifier. For simplicity, we assume each primitive operation is atomic. The formal operational semantics is presented in §6.

Deny-Guarantee Permissions. The main component of our logic is the set of deny-guarantee permissions, PermDG. A deny-guarantee permission is a function that maps each action altering a single variable[2] to a certain deny-guarantee fraction:

$$
\begin{array}{rrcl}
 & \mathsf{Vars} & \stackrel{\text{def}}{=} & \{x, y, z, \ldots\} \\
n \in & \mathsf{Vals} & \stackrel{\text{def}}{=} & \mathbb{Z} \\
\sigma \in & \mathsf{States} & \stackrel{\text{def}}{=} & \mathsf{Vars} \to \mathsf{Vals} \\
a \in & \mathsf{Actions} & \stackrel{\text{def}}{=} & \{\sigma[x \mapsto n], \sigma[x \mapsto n'] \mid \sigma \in \mathsf{States} \ \wedge \ n \neq n'\} \\
f \in & \mathsf{FractionDG} & \stackrel{\text{def}}{=} & \{(\mathsf{deny}, \pi) \mid \pi \in (0,1)\} \cup \{(\mathsf{guar}, \pi) \mid \pi \in (0,1)\} \cup \{0, 1\} \\
pr \in & \mathsf{PermDG} & \stackrel{\text{def}}{=} & \mathsf{Actions} \to \mathsf{FractionDG}
\end{array}
$$

We sometimes write deny-guarantee fractions in FractionDG in shorthand, with $\pi\mathbf{d}$ for (deny, π), and $\pi\mathbf{g}$ for (guar, π).

The fractions represent a permission or a prohibition to perform a certain action. The first two kinds of fractions are symmetric: (deny, π) says that nobody can do the action; (guar, π) says that everybody can do the action. The last two are not: 1 represents full control over the action (only I can do the action), whereas 0 represents no control over an action (others can do it, but I cannot).

From a deny-guarantee permission, pr, we can extract a pair of rely-guarantee conditions. The rely contains those actions permitted to the environment, while the guarantee contains those permitted to the thread (see Fig. 3).

$$
\begin{array}{rcl}
[\![_]\!] & \in & \mathsf{PermDG} \to \mathcal{P}(\mathsf{Actions}) \times \mathcal{P}(\mathsf{Actions}) \\
[\![pr]\!] & \stackrel{\text{def}}{=} & (\{a \mid pr(a) = (\mathsf{guar}, _) \vee pr(a) = 0\}, \\
 & & \quad \{a \mid pr(a) = (\mathsf{guar}, _) \vee pr(a) = 1\})
\end{array}
$$

As shorthand notations, we will use $pr.R$ and $pr.G$ to represent the first and the second element in $[\![pr]\!]$ respectively.

[2] We do not consider updates to simultaneous locations as it complicates the presentation.

$$
\begin{array}{lcl}
\sigma, pr, \gamma \models B & \iff & ([\![B]\!]_\sigma = \mathsf{tt}) \wedge (\forall a.\, pr(a) = 0) \wedge (\gamma = \emptyset) \\
\sigma, pr, \gamma \models pr' & \iff & (\gamma = \emptyset) \wedge (pr = pr') \\
\sigma, pr, \gamma \models \mathsf{full} & \iff & (\gamma = \emptyset) \wedge (\forall a.\, pr(a) = 1) \\
\sigma, pr, \gamma \models \mathsf{Thread}(E, P) & \iff & \gamma = [[\![E]\!]_\sigma \mapsto P] \\
\sigma, pr, \gamma \models P_1 * P_2 & \iff & \exists pr_1, pr_2, \gamma_1, \gamma_2.\, pr = pr_1 \oplus pr_2 \wedge \gamma = \gamma_1 \uplus \gamma_2 \\
 & & \qquad \wedge (\sigma, pr_1, \gamma_1 \models P_1) \wedge (\sigma, pr_2, \gamma_2 \models P_2) \\
 & & \text{where } \uplus \text{ means the union of disjoint sets.} \\
\sigma, pr, \gamma \models P_1 \mathrel{-\!\!*} P_2 & \iff & \forall pr_1, pr_2, \gamma_1, \gamma_2.\, pr_2 = pr \oplus pr_1 \wedge \gamma_2 = \gamma \uplus \gamma_1 \\
 & & \qquad \wedge (\sigma, pr_1, \gamma_1 \models P_1) \text{ implies } (\sigma, pr_2, \gamma_2 \models P_2)
\end{array}
$$

Fig. 5. Semantics of Assertions

Note that the deny and guar labels come with a fractional coefficient. These coefficients are used in defining the addition of two deny-guarantee fractions.

$$
\begin{array}{rcl}
0 \oplus x \stackrel{\mathbf{def}}{=} x \oplus 0 & \stackrel{\mathbf{def}}{=} & x \\
(\mathsf{deny}, \pi) \oplus (\mathsf{deny}, \pi') & \stackrel{\mathbf{def}}{=} & \mathsf{if}\ \pi + \pi' < 1\ \mathsf{then}\ (\mathsf{deny}, \pi + \pi') \\
 & & \quad \mathsf{else\ if}\ \pi + \pi' = 1\ \mathsf{then}\ 1\ \mathsf{else\ undef} \\
(\mathsf{guar}, \pi) \oplus (\mathsf{guar}, \pi') & \stackrel{\mathbf{def}}{=} & \mathsf{if}\ \pi + \pi' < 1\ \mathsf{then}\ (\mathsf{guar}, \pi + \pi') \\
 & & \quad \mathsf{else\ if}\ \pi + \pi' = 1\ \mathsf{then}\ 1\ \mathsf{else\ undef} \\
1 \oplus x \stackrel{\mathbf{def}}{=} x \oplus 1 & \stackrel{\mathbf{def}}{=} & \mathsf{if}\ x = 0\ \mathsf{then}\ 1\ \mathsf{else\ undef}
\end{array}
$$

The addition of two deny-guarantee permissions, $pr = pr_1 \oplus pr_2$, is defined so that for all $a \in \mathsf{Actions}$, $pr(a) = pr_1(a) \oplus pr_2(a)$. The permission inverse inv is defined so $\mathsf{inv}(1) = 0$, $\mathsf{inv}(0) = 1$, $\mathsf{inv}(\mathsf{guar}, \pi) = (\mathsf{guar}, 1 - \pi)$, and $\mathsf{inv}(\mathsf{deny}, \pi) = (\mathsf{deny}, 1 - \pi)$.

It is easy to show that addition is commutative, associative, cancellative, and has 0 as a unit element. This allows us to define a separation logic over PermDG.

Assertions and Judgements. The assertions are defined below.

$$P, Q ::= B \mid pr \mid \mathsf{full} \mid \mathsf{false} \mid \mathsf{Thread}(E, P) \mid P \Rightarrow Q \mid P * Q \mid P \mathrel{-\!\!*} Q \mid \exists x.\, P$$

An assertion P is interpreted as a predicate over a program state σ, a permission token pr, and a thread queue γ. A thread queue, as defined below, is a finite partial function mapping thread identifiers to the postcondition established by the thread when it terminates.

$$t \in \mathsf{ThreadIDs} \stackrel{\mathbf{def}}{=} \mathbb{N} \qquad \gamma \in \mathsf{ThreadQueues} \stackrel{\mathbf{def}}{=} \mathsf{ThreadIDs} \rightharpoonup_{\mathrm{fin}} \mathsf{Assertions}$$

Semantics of assertions is defined in Fig. 5.

The judgments for commands are in the form of $\{P\}\ C\ \{Q\}$. As in Hoare Logic, a command is specified by a precondition (P) and a postcondition (Q). Informally, it means that if the precondition, P, holds in the initial configuration and the environment

$$\frac{P_1 \text{ precise} \quad \{P_1\}\ C\ \{P_2\} \quad x \notin \mathsf{fv}(P_1 * P_3) \quad \mathsf{Thread}(x, P_2) * P_3 \Rightarrow P_4 \quad \mathsf{allowed}([\![x := *]\!], P_3)}{\{P_1 * P_3\}\ x := \mathbf{fork}_{[P_1,P_2]}\ C\ \{P_4\}}\ (\textsc{fork})$$

$$\frac{}{\{P * \mathsf{Thread}(E, P')\}\ \mathbf{join}\ E\ \{P * P'\}}\ (\textsc{join}) \qquad \frac{P_1 \Rightarrow P'_1 \quad \{P'_1\}\ C\ \{P'_2\} \quad P'_2 \Rightarrow P_2}{\{P_1\}\ C\ \{P_2\}}\ (\textsc{cons})$$

$$\frac{\{P\}\ C\ \{P'\} \quad \mathsf{stable}(P_0)}{\{P * P_0\}\ C\ \{P' * P_0\}}\ (\textsc{frame}) \qquad \frac{P \Rightarrow [E/x]P' \quad \mathsf{allowed}([\![x := E]\!], P)}{\{P\}\ x := E\ \{P'\}}\ (\textsc{assn})$$

Fig. 6. Proof Rules

adheres to its specification, then the command C is safe to execute; moreover every forked thread will fulfil its specification and if C terminates, the final configuration will satisfy Q. A formal definition of the semantics is presented in §6.

The main proof rules are shown in Fig. 6. The proof rules are covered by a general side-condition requiring that any assertion we write in a triple is *stable*. Intuitively this means that the assertion still holds under any interference from the environment, as expressed in the deny. Requiring stability for every assertion in a triple removes the need for including explicit stability checks in the proof rules, simplifying the presentation.

Definition 1 (Stability). *An assertion P is* stable *(written* $\mathsf{stable}(P)$*) if and only if, for all σ, σ', pr and γ, if $\sigma, pr, \gamma \models P$ and $(\sigma, \sigma') \in pr.R$, then $\sigma', pr, \gamma \models P$.*

The fork and assign rules include allowed-statements, which assert that particular rewrites are permitted by deny-guarantee assertions. Rewrites are given as relations over states. In the rules, we write $[\![x := E]\!]$ for the relation over states denoted by assigning E to x, where E can be $*$ for non-deterministic assignment.

Definition 2 (Allowed). *Let K be a relation over states. Then* $\mathsf{allowed}(K, P)$ *holds if and only if, for all σ, σ', pr and γ, if $\sigma, pr, \gamma \models P$ and $(\sigma, \sigma') \in K$, then $(\sigma, \sigma') \in pr.G$.*

The assignment rule is an adaptation of Hoare's assignment axiom for sequential programs. In order to deal with concurrency, it checks that the command has enough permission (pr) to update the shared state.

The fork and join rules modify the rules given in [7]. The fork rule takes a precondition and converts it into a Thread-predicate recording the thread's expected postcondition. The rule checks that any pr satisfying the context P_3 is sufficient to allow assignment to the thread variable x. It requires that the variable x used to store the thread identifier is not in $\mathsf{fv}(P_1 * P_3)$, the free variables for the precondition. As with Gotsman et al. [7], the rule also requires that the precondition P_1 is precise.

The join rule takes a thread predicate and replaces it with the corresponding postcondition. The frame and consequence rules are modified from standard separation-logic rules. Other rules are identical to the standard Hoare logic rules.

$\{T_1 * G_2 * G_2 * D_3 * D_3 * L' * \mathtt{x} \neq 1\}$
`t1 := fork`$_{[T_1 * (\mathtt{x} \neq 1), T_1]}$ `(if(x==1) error; x := 1)`
$\{G_2 * G_2 * D_3 * D_3 * L' * \mathsf{Thread}(\mathtt{t1}, T_1)\}$
`t2 := fork`$_{[G_2 * D_3, G_2 * D_3]}$ `(x := 2; if(x==3) error)`
$\{G_2 * D_3 * L' * \mathsf{Thread}(\mathtt{t1}, T_1) * \mathsf{Thread}(\mathtt{t2}, G_2 * D_3)\}$
`join t1;`
$\{T_1 * G_2 * D_3 * L' * \mathsf{Thread}(\mathtt{t2}, G_2 * D_3)\}$
`x := 2;`
$\{T_1 * G_2 * D_3 * L' * \mathsf{Thread}(\mathtt{t2}, G_2 * D_3) * \mathtt{x} = 2\}$
`join t2;`
$\{T_1 * G_2 * G_2 * D_3 * D_3 * L' * \mathtt{x} = 2\}$

where $T_1 \stackrel{\text{def}}{=} [\mathtt{x}\colon \mathbb{Z} \rightsquigarrow 1]_1$, $G_2 \stackrel{\text{def}}{=} [\mathtt{x}\colon \mathbb{Z} \rightsquigarrow 2]_{\frac{1}{2}\mathbf{g}}$, $D_3 \stackrel{\text{def}}{=} [\mathtt{x}\colon \mathbb{Z} \rightsquigarrow 3]_{\frac{1}{2}\mathbf{d}}$,
and $L' \stackrel{\text{def}}{=} [\mathtt{x}\colon \mathbb{Z} \rightsquigarrow \{1,2,3\}]_1 \mathrel{-\!\!*} \mathsf{full}$

Fig. 7. Proof outline of the fork / join example

4 Two-Thread Example

In §2 we said that the program shown in Fig. 1 cannot be verified in conventional rely-guarantee reasoning. We now show that deny-guarantee allows us to verify this example. The proof outline is given in Fig. 7.

We use the following notation to represent permissions. Here $x \in \mathsf{Vars}$, $A, B \subseteq \mathsf{Vals}$ and $f \in \mathsf{FractionDG}$.

$$x\colon A \rightsquigarrow B \stackrel{\text{def}}{=} \{(\sigma[x \mapsto v], \sigma[x \mapsto v']) \mid \sigma \in \mathsf{State} \wedge v \in A \wedge v' \in B \wedge v \neq v'\}$$

$$[X]_f \stackrel{\text{def}}{=} \lambda a. \begin{cases} f & \text{if } a \in X \\ 0 & \text{otherwise} \end{cases}$$

Lemma 3 (Permission splitting).

$$[x\colon A \rightsquigarrow B \uplus B']_f \iff [x\colon A \rightsquigarrow B]_f * [x\colon A \rightsquigarrow B']_f$$
$$[x\colon A \rightsquigarrow B]_{f \oplus f'} \iff [x\colon A \rightsquigarrow B]_f * [x\colon A \rightsquigarrow B]_{f'}$$

Lemma 4 (Permission subtraction). *If P is precise and satisfiable, then* $(P \mathrel{-\!\!*} \mathsf{full}) * P \iff \mathsf{full}$.

Proof. Holds because $(P \mathrel{-\!\!*} Q) * P \iff Q \wedge (P * \mathsf{true})$ and $\mathsf{full} \Rightarrow P * \mathsf{true}$ hold for any precise and satisfiable P and any Q. □

The fork / join program has precondition $\{\mathsf{full} * \mathtt{x} \neq 1\}$, giving the full permission, 1, on every action. The permission $[\mathtt{x}\colon \mathbb{Z} \rightsquigarrow \{1,2,3\}]_1$ permits any rewrite of the variable x to the value 1, 2 or 3, and prohibits all other rewrites. By Lemma 4,

$$\mathsf{full} \iff ([\mathtt{x}\colon \mathbb{Z} \rightsquigarrow \{1,2,3\}]_1 \mathrel{-\!\!*} \mathsf{full}) * [\mathtt{x}\colon \mathbb{Z} \rightsquigarrow \{1,2,3\}]_1$$

By Lemma 3 can split $[\mathtt{x}\colon \mathbb{Z} \rightsquigarrow \{1,2,3\}]_1$ as follows

$$\begin{aligned}[\mathtt{x}\colon \mathbb{Z} \rightsquigarrow \{1,2,3\}]_1 &\iff [\mathtt{x}\colon \mathbb{Z} \rightsquigarrow 1]_1 * [\mathtt{x}\colon \mathbb{Z} \rightsquigarrow 2]_1 * [\mathtt{x}\colon \mathbb{Z} \rightsquigarrow 3]_1 \\ &\iff T_1 * G_2 * G_2 * D_3 * D_3\end{aligned}$$

where T_1, G_2 and D_3 are defined in Fig. 7. We define L' as $([\mathtt{x}\colon \mathbb{Z} \rightsquigarrow \{1,2,3\}]_1 \mathrel{-\!\!*} \mathsf{full})$ (the L used in the proof sketch in Fig. 2 is $L' * G_2 * D_3$). Consequently, we can derive the precondition $\{T_1 * G_2 * G_2 * D_3 * D_3 * L' * \mathtt{x} \neq 1\}$

The specification for thread `t1` is shown below. Note that $\mathtt{x} \neq 1$ is stable because T_1 prevents the environment from writing 1 into `x`. The post-condition does not include $\mathtt{x} = 1$, because T_1 does not prohibit the environment from writing other values into `x`.

$$\{\, T_1 * \mathtt{x} \neq 1\} \quad \texttt{if(x==1) error; x := 1;} \quad \{\, T_1\}$$

The specification for thread `t2` is shown below. The assertion $x \neq 3$ is stable because the permission D_3 is a deny prohibiting the environment from writing 3 in `x`. Note that a deny is used rather than full permission because another instance of D_3 is needed to ensure stability of the assertion on line 9, before the main thread joins `t2`.

$$\{G_2 * D_3\} \quad \texttt{x := 2;} \quad \{G_2 * D_3 * x \neq 3\} \quad \texttt{if(x==3) error} \quad \{G_2 * D_3\}$$

The specifications for `t1` and `t2` allow us to apply the fork rule (lines 2 and 4). We then join the thread `t1` and recover the permission T_1 (line 6). Then we apply the assignment rule for the assignment `x := 2` (line 8).

The post-condition $\mathtt{x} = 2$ on line 9 is stable because $T_1 * L'$ gives the exclusive permission, 1, on every rewrite except rewrites of `x` with value 2 or 3, and the deny D_3 prohibits rewrites of `x` with value 3. Consequently the only permitted interference from the environment is to write 2 into `x`, so $\mathtt{x} = 2$ is stable.

Finally we apply the join rule, collect the permissions held by the thread `t2`, and complete the proof.

5 Encoding Rely-Guarantee Reasoning

In this section, we show that the traditional rely-guarantee reasoning can be embedded into our deny-guarantee reasoning. First, we present an encoding of parallel composition using the fork and join commands, and derive a proof rule. Then, we prove that every rely-guarantee proof for programs using parallel composition can be translated into a corresponding deny-guarantee proof.

5.1 Adding Parallel Composition

We encode parallel composition into our language by the following translation:

$$C_1 \parallel_{(\mathtt{x},P_1,Q_1)} C_2 \stackrel{\text{def}}{=} \mathtt{x} := \textbf{fork}_{[P_1,Q_1]}\, C_1;\, C_2;\, \textbf{join}\ \mathtt{x}$$

Here the annotations P_1, Q_1 are required to provide the translation onto the **fork** , which requires annotations. `x` is an intermediate variable used to hold the identifier for thread

C_1. We assume that $\mathtt{x}$ is a fresh variable that is not used in C_1 or C_2. The parallel composition rule for deny-guarantee is as follows:

$$\frac{\{P_1\}\, C_1\, \{Q_1\} \quad \{P_2\}\, C_2\, \{Q_2\} \quad \mathtt{x} \notin \mathsf{fv}(P_1, P_2, C_1, C_2, Q_1, Q_2) \quad P_1 \text{ precise}}{\{P_1 * P_2 * \mathsf{full}(\mathtt{x})\}\, C_1 \parallel_{(\mathtt{x}, P_1, Q_1)} C_2\, \{Q_1 * Q_2 * \mathsf{full}(\mathtt{x})\}} \ (\textsc{par})$$

Modulo the side-conditions about $\mathtt{x}$ and precision, and the $\mathsf{full}(\mathtt{x})$ star-conjunct, this is the same rule as in separation logic. The assertion $\mathsf{full}(\mathtt{x})$ stands for the full permission on the variable $\mathtt{x}$; that is, we have full permission to assign any value to $\mathtt{x}$.

$$\mathsf{full}(\mathtt{x})(\sigma, \sigma') \stackrel{\textbf{def}}{=} \text{ if } \sigma[x \mapsto v] = \sigma' \wedge v \neq \sigma(x) \text{ then } 1, \text{ else } 0$$

We extend this notation to sets of variables: $\mathsf{full}(\{\mathtt{x}_1, \ldots, \mathtt{x}_n\}) \stackrel{\textbf{def}}{=} \mathsf{full}(\mathtt{x}_1) \oplus \ldots \oplus \mathsf{full}(\mathtt{x}_n)$.

Precision is required as the underlying **fork** rule requires it. This makes this rule weaker than if we directly represented the parallel composition in the semantics.

Lemma 5. *The parallel composition rule can be derived from the rules given in Fig. 6.*

Proof. The proof has the following outline.

$$\begin{array}{l}
\{P_1 * P_2 * \mathsf{full}(\mathtt{x})\} \\
\quad \mathtt{x} := \textbf{fork}_{[P_1, Q_1]}\, C_1 \\
\{\mathsf{Thread}(\mathtt{x}, Q_1) * P_2 * \mathsf{full}(\mathtt{x})\} \\
\quad C_2 \\
\{\mathsf{Thread}(\mathtt{x}, Q_1) * Q_2 * \mathsf{full}(\mathtt{x})\} \\
\quad \textbf{join}\ \mathtt{x} \\
\{Q_1 * Q_2 * \mathsf{full}(\mathtt{x})\}
\end{array}$$

The first step uses the first premise, and the frame and fork rules. The second step uses the second premise and the frame rule. The final step uses the frame and join rules.

5.2 Translation

Now let us consider the translation of rely-guarantee proofs into the deny-guarantee framework. The encoding of parallel composition into **fork** and **join** introduces extra variables, so we partition variables in constructed fork-join programs into two kinds: Vars, the original program variables, and TVars, variables introduced to carry thread identifiers. We will assume that the relies and guarantees from the original proof assume that the TVars are unchanged.

In §3, we showed how to extract a pair of rely-guarantee conditions from permissions $pr \in \mathsf{PermDG}$. Conversely, we can encode rely-guarantee pairs into sets of PermDG permissions as follows:

$$\begin{array}{ll}
[\![_]\!] & \in \mathcal{P}(\mathsf{Actions}) \times \mathcal{P}(\mathsf{Actions}) \to \mathcal{P}(\mathsf{PermDG}) \\
[\![R, G]\!] & \stackrel{\textbf{def}}{=} \{\langle R, G\rangle_F \mid F \in \mathsf{Actions} \to (0, 1)\} \\
\langle R, G\rangle_F & \stackrel{\textbf{def}}{=} \lambda a. \begin{cases} (\mathsf{guar}, F(a)) & a \in R \wedge a \in G \\ 0 & a \in R \wedge a \notin G \\ 1 & a \notin R \wedge a \in G \\ (\mathsf{deny}, F(a)) & a \notin R \wedge a \notin G \end{cases}
\end{array}$$

First, we show that our translation is non-empty: each pair maps to something:

Lemma 6 (Non-empty translation). $\forall R, G.\ [\![R, G]\!] \neq \emptyset$

By algebraic manipulation, we can show that the definition above corresponds to the following more declarative definition:

Lemma 7. $[\![R, G]\!] = \{pr \mid [\![pr]\!] = (R, G)\}$

Moreover, as R and G assume that the TVars are unchanged, the following lemma holds:

Lemma 8. *If $pr \in [\![R, G]\!]$, and $X \subseteq \mathsf{TVars}$, then $\mathsf{full}(X) \oplus pr$ is defined.*

Now, we can translate rely-guarantee judgements into a non-empty set of equivalent triples in deny-guarantee. Non-emptiness follows from Lemmas 6 and 8.

Definition 9 (Triple translation).

$$[\![R, G \vdash_{\mathrm{RG}} \{P\}\ C\ \{Q\}]\!]_X \stackrel{\text{def}}{=} \forall pr \in [\![R, G]\!].\ \exists C'.\ \vdash \{P * pr * \mathsf{full}(X)\}\ C'\ \{Q * pr * \mathsf{full}(X)\} \\ \wedge\ C = erase(C')$$

where the set $X \subseteq \mathsf{TVars}$ carries the set of identifiers used in the parallel compositions, and $erase(C')$ is C' with all annotations removed from parallel compositions.

Note that the judgement $R, G \vdash_{\mathrm{RG}} \{P\}\ C\ \{Q\}$ in traditional rely-guarantee reasoning does not need annotations in C. The C is a cleaned-up version of some annotated statement C'. We elide the standard rely-guarantee rules here. This translation allows us to state the following theorem:

Theorem 10 (Complete embedding). *If $R, G \vdash_{\mathrm{RG}} \{P\}\ C\ \{Q\}$ is derivable according to the rely-guarantee proof rules, then $[\![R, G \vdash_{\mathrm{RG}} \{P\}\ C\ \{Q\}]\!]_X$ holds.*

In other words, given a proof in rely-guarantee, we can construct an equivalent proof using deny-guarantee. We prove this theorem by considering each rely-guarantee proof rule separately, and showing that the translated versions of the rely-guarantee proof rules are sound in deny-guarantee. Below we give proofs of the two most interesting rules: the rule of parallel composition and of weakening. For each of these, we first need a corresponding helper lemma for the translation of the rely-guarantee conditions. These helper lemmas follow from the definitions of PermDG and $[\![R, G]\!]$.

Lemma 11 (Composition). *If $G_1 \subseteq R_2$, $G_2 \subseteq R_1$, and $pr \in [\![R_1 \cap R_2, G_1 \cup G_2]\!]$, then there exist pr_1, pr_2 such that $pr = pr_1 \oplus pr_2$ and $pr_1 \in [\![R_1, G_1]\!]$ and $pr_2 \in [\![R_2, G_2]\!]$.*

Lemma 12 (Soundness of translated parallel rule).
If $G_2 \subseteq R_1$, $G_1 \subseteq R_2$, $[\![R_1, G_1 \vdash_{\mathrm{RG}} \{P_1\} C_1 \{Q_1\}]\!]_X$ and $[\![R_2, G_2 \vdash_{\mathrm{RG}} \{P_2\} C_2 \{Q_2\}]\!]_Y$, then $[\![R_1 \cap R_2, G_1 \cup G_2 \vdash_{\mathrm{RG}} \{P_1 \wedge P_2\} C_1 \parallel C_2 \{Q_1 \wedge Q_2\}]\!]_{\{x\} \uplus X \uplus Y}$

Lemma 13 (Weakening). *If $R_2 \subseteq R_1$, $G_1 \subseteq G_2$, and $pr \in [\![R_2, G_2]\!]$ then there exist permissions pr_1, pr_2 such that $pr = pr_1 \oplus pr_2$ and $pr_1 \in [\![R_1, G_1]\!]$.*

Lemma 14 (Soundness of translated weakening rule). *If $R_2 \subseteq R_1$, $G_1 \subseteq G_2$, and $[\![R_1, G_1 \vdash_{\mathrm{RG}} \{P\} C \{Q\}]\!]_X$, then $[\![R_2, G_2 \vdash_{\mathrm{RG}} \{P\} C \{Q\}]\!]_X$.*

6 Semantics and Soundness

The operational semantics of the language is defined in Fig. 8. The semantics is divided into two parts: the local semantics and the global semantics. The local semantics is closely related to the interpretation of the logical judgements, while the global semantics can easily be erased to a machine semantics. This erasure and other additional definitions and proofs can be found in the associated technical report [4].

Local Semantics. The local semantics represents the view of execution from a single thread. It is defined using the constructs described in §3. The commands all work with an abstraction of the environment: γ abstracts the other threads, and carries their final states; and pr abstracts the interference from other threads and the interference that it is allowed to generate. The semantics will result in **abort** if it does not respect the abstraction.

The first two rules, in Fig. 8, deal with assignment. If the assignment is allowed by pr, then it executes successfully, otherwise the program aborts signalling an error. The next two rules handle the joining of threads. If the thread being joined with is in γ, then that thread's terminal pr' and γ' are added to the current thread before the current thread continues executing. We annotate the transition with **join** (t, pr', γ'), so the semantics can be reused in the global semantics. If the thread identifier is not in γ, we signal an error as we are joining on a thread that we do not have permission to join. The next two rules deal with forking new threads. If part of the state satisfies P then we remove that part of the state, and extend our environment with a new thread that will terminate in a state satisfying Q. If there is no part of the state satisfying P, then we will raise an error as we do not have the permission to give to the new thread. The remaining local rules deal with sequential composition.

In the next section of Fig. 8, we define $\overset{r}{\rightsquigarrow}$, which represents the environment performing an action. We also define $\rightsquigarrow^*$ as the transitive and reflexive closure of the operational semantics extended with the environment action.

Given this semantics, we say a local thread is safe if it will not reach an error state.

Definition 15. $\vdash (C, \sigma, pr, \gamma)\ \mathit{safe} \iff \neg((C, \sigma, pr, \gamma) \rightsquigarrow^* \mathbf{abort})$

We can give the semantics of the judgements from earlier in terms of this local operational semantics.

Definition 16 (Semantics of a triple). $\models \{P\}C\{Q\}$ *asserts that, if* $\sigma, pr, \gamma \models P$*, then*

- *(1)* $\vdash (C, \sigma, pr, \gamma)\ \mathit{safe}$*; and*
- *(2) if* $(C, \sigma, pr, \gamma) \rightsquigarrow^* (\mathbf{skip}, \sigma', pr', \gamma')$*, then* $\sigma', pr', \gamma' \models Q$*.*

As the programs carry annotations for each **fork**, we need to define programs that are well-annotated, that is, the code for each fork satisfies its specification.

Definition 17 (Well-annotated command). *We define a command as well-annotated,* $\vdash C\ \mathit{wa}$*, as follows*

$$\begin{aligned} \vdash \mathbf{fork}_{[P,Q]}\ C\ \mathit{wa} &\iff \models \{P\}C\{Q\} \ \wedge\ \vdash C\ \mathit{wa} \\ \vdash \mathbf{skip}\ \mathit{wa} &\iff \mathit{always} \\ \vdash C_1; C_2\ \mathit{wa} &\iff \vdash C_1\ \mathit{wa} \ \wedge\ \vdash C_2\ \mathit{wa} \\ &\ldots \end{aligned}$$

Local semantics

$$\frac{[\![E]\!]_\sigma = n \quad (\sigma, \sigma[x \mapsto n]) \in pr.G}{(x := E, \sigma, pr, \gamma) \rightsquigarrow (\mathbf{skip}, \sigma[x \mapsto n], pr, \gamma)} \qquad \frac{[\![E]\!]_\sigma = n \quad (\sigma, \sigma[x \mapsto n]) \notin pr.G}{(x := E, \sigma, pr, \gamma) \rightsquigarrow \mathbf{abort}}$$

$$\frac{[\![E]\!]_\sigma = t \quad \gamma(t) = Q \quad \sigma, pr', \gamma' \models Q}{(\mathbf{join}\ E, \sigma, pr, \gamma) \overset{\mathbf{join}\ (t, pr', \gamma')}{\rightsquigarrow} (\mathbf{skip}, \sigma, pr \oplus pr', (\gamma \setminus t) \uplus \gamma')} \qquad \frac{[\![E]\!]_\sigma = t \quad t \notin dom(\gamma)}{(\mathbf{join}\ E, \sigma, pr, \gamma) \rightsquigarrow \mathbf{abort}}$$

$$\frac{t \notin dom(\gamma) \quad \sigma, pr', \gamma' \models P \quad pr = pr' \oplus pr'' \quad \gamma = \gamma' \uplus \gamma'' \quad (\sigma, \sigma[x \mapsto t]) \in pr.G}{(x := \mathbf{fork}_{[P,Q]}\ C, \sigma, pr, \gamma) \overset{\mathbf{fork}\ (t, C, pr', \gamma')}{\rightsquigarrow} (\mathbf{skip}, \sigma[x \mapsto t], pr'', \gamma''[t \mapsto Q])}$$

$$\frac{\sigma, pr, \gamma \not\models P * \mathsf{true}}{(x := \mathbf{fork}_{[P,Q]}\ C, \sigma, pr, \gamma) \rightsquigarrow \mathbf{abort}} \qquad \frac{(\sigma, \sigma[x \mapsto t]) \notin pr.G}{(x := \mathbf{fork}_{[P,Q]}\ C, \sigma, pr, \gamma) \rightsquigarrow \mathbf{abort}}$$

$$\frac{(C, \sigma, pr, \gamma) \rightsquigarrow (C', \sigma', pr', \gamma')}{(C; C'', \sigma, pr, \gamma) \rightsquigarrow (C'; C'', \sigma', pr', \gamma')} \qquad \frac{}{(\mathbf{skip}; C, \sigma, pr, \gamma) \rightsquigarrow (C, \sigma, pr, \gamma)}$$

$$\frac{(C, \sigma, pr, \gamma) \rightsquigarrow \mathbf{abort}}{(C; C', \sigma, pr, \gamma) \rightsquigarrow \mathbf{abort}}$$

Interference

$$\frac{(\sigma, \sigma') \in pr.R}{(C, \sigma, pr, \gamma) \overset{r}{\rightsquigarrow} (C, \sigma', pr, \gamma)} \qquad \frac{\forall (t \mapsto C, pr, \gamma) \in \delta.\ (\sigma, \sigma') \in pr.R}{(\sigma, \delta) \overset{r}{\Longmapsto} (\sigma', \delta)}$$

Global semantics

$$\frac{(C, \sigma, pr, \gamma) \rightsquigarrow (C', \sigma', pr', \gamma') \quad (\sigma, \delta) \overset{r}{\Longmapsto} (\sigma', \delta')}{(\sigma, [t \mapsto C, pr, \gamma] \uplus \delta) \Longmapsto (\sigma', [t \mapsto C', pr', \gamma'] \uplus \delta')}$$

$$\frac{(C, \sigma, pr, \gamma) \overset{\mathbf{fork}\ (t_2, C_2, pr_2, \gamma_2)}{\rightsquigarrow} (C', \sigma', pr', \gamma') \quad (\sigma, \delta) \overset{r}{\Longmapsto} (\sigma', \delta')}{(\sigma, [t_1 \mapsto C, pr, \gamma] \uplus \delta) \Longmapsto (\sigma', [t \mapsto C', pr', \gamma'] \uplus [t_2 \mapsto C_2, pr_2, \gamma_2] \uplus \delta')}$$

$$\frac{(C, \sigma, pr, \gamma) \overset{\mathbf{join}\ (t_2, pr_2, \gamma_2)}{\rightsquigarrow} (C', \sigma', pr', \gamma') \quad (\sigma, \delta) \overset{r}{\Longmapsto} (\sigma', \delta')}{(\sigma, [t_1 \mapsto C, pr, \gamma] \uplus [t_2 \mapsto \mathbf{skip}, pr_2, \gamma_2] \uplus \delta) \Longmapsto (\sigma', [t \mapsto C', pr', \gamma'] \uplus \delta')}$$

$$\frac{(C, \sigma, pr, \gamma) \rightsquigarrow \mathbf{abort}}{(\sigma, [t \mapsto C, pr, \gamma] \uplus \delta) \Longmapsto \mathbf{abort}} \qquad \frac{(C, \sigma, pr, \gamma) \overset{-}{\rightsquigarrow} (C, \sigma', pr', \gamma') \quad \neg(\exists \delta'.\ (\sigma, \delta) \overset{r}{\Longmapsto} (\sigma', \delta'))}{(\sigma, [t \mapsto C, pr, \gamma] \uplus \delta) \Longmapsto \mathbf{abort}}$$

$$\frac{(C, \sigma, pr, \gamma) \overset{\mathbf{join}\ (t_2, pr_3, \gamma_3)}{\rightsquigarrow} (C', \sigma', pr', \gamma') \quad \neg((C, \sigma, pr, \gamma) \overset{\mathbf{join}\ (t_2, pr_2, \gamma_2)}{\rightsquigarrow} (C', \sigma', pr', \gamma') \quad)}{(\sigma, [t_1 \mapsto C, pr, \gamma] \uplus [t_2 \mapsto \mathbf{skip}, pr_2, \gamma_2] \uplus \delta) \Longmapsto \mathbf{abort}}$$

Fig. 8. Operational Semantics

Given these definitions we can now state soundness of our logic with respect to the local semantics.

Theorem 18 (Local soundness). *If* $\vdash \{P\}C\{Q\}$*, then* $\models \{P\}C\{Q\}$ *and* $\vdash C$ wa.

Global Semantics. Now we will consider the operational semantics of the whole machine, that is, for all the threads. This semantics is designed as a stepping stone between the local semantics and the concrete machine semantics. We need an additional abstraction of the global thread-queue.

$$\delta \in \mathsf{GThrdQ} \stackrel{\mathbf{def}}{=} \mathsf{ThreadIDs} \rightharpoonup_{\mathrm{fin}} \mathsf{Stmts} \times \mathsf{PermDG} \times \mathsf{ThreadQueues}$$

In the third part of Fig. 8, we present the global operational semantics. The first rule progresses one thread, and advances the rest with a corresponding environment action. The second rule deals with removing a thread from a machine when it is successfully joined. Here the label ensures that the local semantics uses the same final state for the thread as it actually has. The third rule creates a new thread. Again the label carries the information required to ensure the local thread semantics has the same operation as the global machine.

The three remaining rules deal with the cases when something goes wrong. The first rule says that if the local semantics can fault, then the global semantics can also. The second raises an error if a thread performs an action that cannot be accepted as a legal environment action by other threads. The final rule raises an error if a thread has terminated and another thread tries to join on it, but cannot join with the right final state.

We can prove the soundness of our logic with respect to this global semantics.

Theorem 19 (Global soundness). *If* $\vdash \{P\}C\{Q\}$ *and* $\sigma, 1, \emptyset \models P$*, then*

- $\neg((\sigma, [t \mapsto C, 1, \emptyset]) \Longrightarrow^* \mathbf{abort})$*; and*
- *if* $(\sigma, [t \mapsto C, 1, \emptyset]) \Longrightarrow^* (\sigma', [t \mapsto \mathbf{skip}, pr, \gamma])$ *then* $\sigma', pr, \gamma \models Q$.

This says, if we have proved a program and it does not initially require any other threads, then we can execute it without reaching **abort**, and if it terminates the final state will satisfy the postcondition.

7 Conclusions and Future Developments

In this paper we have demonstrated that deny-guarantee enables reasoning about programs using dynamically scoped threads, that is, programs using fork to create new threads and join to wait for their termination. Rely-guarantee cannot reason about this form of concurrency. Our extension borrows ideas from separation logic to enable an interference to be split dynamically with a logical operation, $*$.

We have applied the deny-guarantee method to a setting with only a pre-allocated set of global variables. However, deny-guarantee extends naturally to a setting with memory allocation and deallocation.

Deny-guarantee can be applied to separation logic in much the same way as rely-guarantee, because the deny-guarantee approach is largely orthogonal to the presence of the heap. Deny-guarantee permissions can be made into *heap permissions* by defining actions as binary relations over heaps, rather than over states with fixed global variables.

The SAGL [5] and RGSep [12] approaches can be easily extended to a setting with fork and join by using heap permissions in place of relies and guarantees.

Finally, deny-guarantee may allow progress on the problem of reasoning about dynamically-allocated locks in the heap. Previous work in this area, such as [7] and [9], has associated locks with invariants. With deny-guarantee we can associate locks with heap permissions, and make use of compositional deny-guarantee reasoning. However, considerable challenges remain, in particular the problems of recursive stability checking and of locks which refer to themselves (Landin's 'knots in the store'). We will address these challenges in future work.

Acknowledgements. We should like to thank Alexey Gotsman, Tony Hoare, Tom Ridge, Kristin Rozier, Sam Staton, John Wickerson and the anonymous referees for comments on this paper. We acknowledge funding from EPSRC grant EP/F019394/1 (Parkinson and Dodds) and a Royal Academy of Engineering / EPSRC fellowship (Parkinson).

References

[1] Bornat, R., Calcagno, C., O'Hearn, P., Parkinson, M.: Permission accounting in separation logic. In: POPL 2005, pp. 259–270. ACM Press, New York (2005)

[2] Boyland, J.: Checking interference with fractional permissions. In: Cousot, R. (ed.) SAS 2003. LNCS, vol. 2694, pp. 55–72. Springer, Heidelberg (2003)

[3] Calcagno, C., O'Hearn, P.W., Yang, H.: Local action and abstract separation logic. In: LICS 2007, pp. 366–378. IEEE Computer Society, Los Alamitos (2007)

[4] Dodds, M., Feng, X., Parkinson, M., Vafeiadis, V.: Deny-guarantee reasoning (extended version and formalization in Isabelle). Technical Report UCAM-CL-TR-736, University of Cambridge (2009), `http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-736.html`

[5] Feng, X., Ferreira, R., Shao, Z.: On the relationship between concurrent separation logic and assume-guarantee reasoning. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 173–188. Springer, Heidelberg (2007)

[6] Feng, X., Shao, Z.: Modular verification of concurrent assembly code with dynamic thread creation and termination. In: Proc. ICFP 2005, pp. 254–267. ACM Press, New York (2005)

[7] Gotsman, A., Berdine, J., Cook, B., Rinetzky, N., Sagiv, M.: Local reasoning for storable locks and threads. In: Shao, Z. (ed.) APLAS 2007. LNCS, vol. 4807, pp. 19–37. Springer, Heidelberg (2007)

[8] Haack, C., Hurlin, C.: Separation logic contracts for a java-like language with fork/join. In: Meseguer, J., Roşu, G. (eds.) AMAST 2008. LNCS, vol. 5140, pp. 199–215. Springer, Heidelberg (2008)

[9] Hobor, A., Appel, A.W., Nardelli, F.Z.: Oracle semantics for concurrent separation logic. In: Drossopoulou, S. (ed.) ESOP 2008. LNCS, vol. 4960, pp. 353–367. Springer, Heidelberg (2008)

[10] Jones, C.B.: Tentative steps toward a development method for interfering programs. ACM Trans. Program. Lang. Syst. 5(4), 596–619 (1983)

[11] Jones, C.B.: Annoted bibliography on rely/guarantee conditions (2007), `http://homepages.cs.ncl.ac.uk/cliff.jones/ftp-stuff/rg-hist.pdf`

[12] Vafeiadis, V., Parkinson, M.: A marriage of rely/guarantee and separation logic. In: Caires, L., Vasconcelos, V.T. (eds.) CONCUR 2007. LNCS, vol. 4703, pp. 256–271. Springer, Heidelberg (2007)