Technical Report

UCAM-CL-TR-736
ISSN 1476-2986

Number 736

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

Deny-guarantee reasoning

Mike Dodds, Xinyu Feng, Matthew Parkinson,
Viktor Vafeiadis

January 2009

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

Technical reports published by the University of Cambridge
Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

Deny-guarantee reasoning
(extended version and formalization in Isabelle)

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 [9] 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 a program's precondition and postcondition may differ

```
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;
```

Figure 1: Illustration of fork/join

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 (see Appendix D).

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 [9, 10]. 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 [4] and RGSep [11] 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.* [5] and Hobor *et al.* [8] 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.* [6] 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 [7] 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 `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

$\{T_1 * G_2 * D_3 * L * \mathtt{x} \neq 1\}$
```
  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
```
$\{T_1 * G_2 * D_3 * L * \mathtt{x} = 2\}$

Figure 2: Proof outline

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.

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.* [6]) 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].

[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 [7] to our deny-guarantee setting to handle Java-style join.

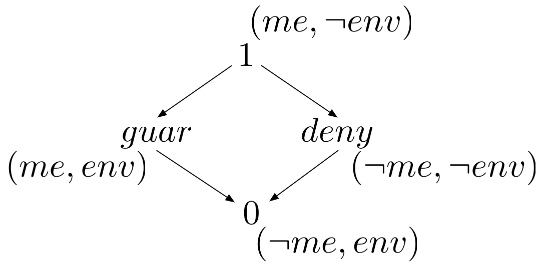

Figure 3: Possible interference

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 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 $\mathtt{x} = 2$.

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).

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.

2.1 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 $*$.

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

Figure 4: The Language

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

3.1 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. 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.

3.2 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:

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

We sometimes write deny-guarantee fractions in $\mathsf{FractionDG}$ 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

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

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).

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

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

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.

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

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.

3.3 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.

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

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 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.

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

Figure 5: Semantics of Assertions

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 $\mathsf{allowed}$-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 [6]. The fork rule takes a precondition and converts it into a Thread-predicate recording the thread's expected post-condition. 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. [6], the rule also requires that the precondition P_1 is precise.

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

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

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

$$\frac{}{\{P * \mathsf{Thread}(E, P')\}\ \mathbf{join}\ E\ \{P * P'\}}\ (\text{JOIN}) \qquad \frac{P_1 \Rightarrow P_1' \qquad \{P_1'\}\ C\ \{P_2'\} \qquad P_2' \Rightarrow P_2}{\{P_1\}\ C\ \{P_2\}}\ (\text{CONS})$$

$$\frac{\{P\}\ C\ \{P'\} \qquad \mathsf{stable}(P_0)}{\{P * P_0\}\ C\ \{P' * P_0\}}\ (\text{FRAME}) \qquad \frac{P \Rightarrow [E/x]P' \qquad \mathsf{allowed}([\![x := E]\!], P)}{\{P\}\ x := E\ \{P'\}}\ (\text{ASSN})$$

Figure 6: Proof 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 $L' \stackrel{\text{def}}{=} [\mathtt{x}\colon \mathbb{Z} \rightsquigarrow \{1,2,3\}]_1 \mathrel{-\!\!*} \mathsf{full}$

Figure 7: Proof outline of the fork / join example

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.

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

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.

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

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:

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

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 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 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 x; that is, we have full permission to assign any value to x.

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

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

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.

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

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 $pr \in$ PermDG. Conversely, we can encode rely-guarantee pairs into sets of PermDG permissions as follows:

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

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)**.**

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

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 $[\![R_1 \cap R_2, G_1 \cup G_2 \vdash_{\mathrm{RG}} \{P_1 \wedge P_2\}C_1 \parallel C_2\{Q_1 \wedge Q_2\}]\!]_{\{x\} \uplus X \uplus Y}$

Proof. By Lemma 11, we know for any $pr \in [\![R_1 \cap R_2, G_1 \cup G_2]\!]$, that there exists a pr_1 and pr_2 such that $pr = pr_1 \oplus pr_2$, $pr_1 \in [\![R_1, G_1]\!]$ and $pr_2 \in [\![R_2, G_2]\!]$. Using assumptions we have $\{P_1 * pr_1 * \mathsf{full}(X)\}\, C_1\, \{Q_1 * pr_1 * \mathsf{full}(X)\}$ and $\{P_2 * pr_2 * \mathsf{full}(Y)\}\, C_2\, \{Q_2 * pr_2 * \mathsf{full}(Y)\}$. Hence, by the deny-guarantee parallel rule
$\{(P_1 * pr_1 * P_2 * pr_2 * \mathsf{full}(X \uplus Y) * \mathsf{full}(\mathrm{x})\}\, C_1 \parallel_{(\mathrm{x}, P_1 * pr_1 * \mathsf{full}(X), Q_1 * pr_1 * \mathsf{full}(X))} C_2\, \{Q_1 * pr_1 * Q_2 * pr_2 * \mathsf{full}(X \uplus Y) * \mathsf{full}(\mathrm{x})\}$ and noting that $P_1 * P_2 \iff P_1 \wedge P_2$ and $Q_1 * Q_2 \iff Q_1 \wedge Q_2$ completes the proof. □ □

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 $pr_1 \in [\![R_1, G_1]\!]$.*

Proof. Assume $f_{pr}: \mathsf{Actions} \to (M - \{0,1\})$ is a function such that for each action a, $f_{pr}(a) = i$ if $pr(a) = (\_, i)$, and $f_{pr}(a)$ is arbitrary otherwise. Then we define pr_1 as $\langle R_1, G_1 \rangle_{f_{pr}}$. We construct pr_2 as follows.

$$pr_2(a) \stackrel{\mathbf{def}}{=} \begin{cases} 0 & a \notin (G_2 - G_1) \cup (R_1 - R_2) \\ 1 & a \in (R_1 \cap G_2) - (R_2 \cup G_1) \\ \mathsf{inv}(pr_1(a)) & a \in (G_2 - G_1 - R_1) \cup (G_1 \cap (R_1 - R_2)) \\ (\mathsf{deny}, f_{pr}(a)) & a \in R_1 - R_2 - G_2 \\ (\mathsf{guar}, f_{pr}(a)) & a \in (R_2 \cap G_2) - G_1 \end{cases}$$

We can prove that pr_2 is total and $pr_1 \oplus pr_2 = pr$. □ □

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 $[\![R_2, G_2 \vdash_{\mathrm{RG}} \{P\}C\{Q\}]\!]_X$.*

Proof. By lemma 13, we know for any $pr \in [\![R_2, G_2]\!]$ there must exist permissions pr_1, pr_2 such that $pr = pr_1 \oplus pr_2$ and $pr_1 \in [\![R_1, G_1]\!]$. Using assumption we have
$\{P * pr_1 * \mathsf{full}(X)\}\, C\, \{Q * pr_1 * \mathsf{full}(X)\}$ and hence, by the deny-guarantee frame rule
$\{P * pr * \mathsf{full}(X)\}\, C\, \{Q * pr * \mathsf{full}(X)\}$ as required. □ □

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. Additional definitions and proofs can be found in the appendix.

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

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) $\vdash (C, \sigma, pr, \gamma)\ \mathsf{safe}$; 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, $\vdash C\ \mathsf{wa}$, as follows*

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

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 $\vdash C\ \mathsf{wa}$.*

6.2 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.

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

Local semantics

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

$$\frac{\llbracket E \rrbracket_\sigma = t \quad \gamma(t) = Q \quad \sigma, pr', \gamma' \models Q}{(\mathbf{join}\ E, \sigma, pr, \gamma) \overset{\mathbf{join}\ (t, pr', \gamma')}{\leadsto} (\mathbf{skip}, \sigma, pr \oplus pr', (\gamma \setminus t) \uplus \gamma')} \qquad \frac{\llbracket E \rrbracket_\sigma = t \quad t \notin dom(\gamma)}{(\mathbf{join}\ E, \sigma, pr, \gamma) \leadsto \mathbf{abort}}$$

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

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

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

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

Interference

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

Global semantics

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

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

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

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

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

Figure 8: Operational Semantics

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*

- $\neg((\sigma, [t \mapsto C, 1, \emptyset]) \Longmapsto^* \textbf{abort})$*; and*
- *if* $(\sigma, [t \mapsto C, 1, \emptyset]) \Longmapsto^* (\sigma', [t \mapsto \textbf{skip}, pr, \gamma])$ *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 [4] and RGSep [11] 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 [6] and [8], 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] R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson. Permission accounting in separation logic. In *POPL'05*, pages 259–270. ACM Press, 2005.

[2] J. Boyland. Checking interference with fractional permissions. In *Proc. of SAS'03*, volume 2694 of *LNCS*, pages 55–72. Springer, 2003.

[3] C. Calcagno, P. W. O'Hearn, and H. Yang. Local action and abstract separation logic. In *LICS'07*, pages 366–378. IEEE Computer Society, 2007.

[4] X. Feng, R. Ferreira, and Z. Shao. On the relationship between concurrent separation logic and assume-guarantee reasoning. In *Proc. ESOP'07*, volume 4421 of *LNCS*, pages 173–188. Springer, 2007.

[5] X. Feng and Z. Shao. Modular verification of concurrent assembly code with dynamic thread creation and termination. In *Proc. ICFP'05*, pages 254–267. ACM Press, 2005.

[6] A. Gotsman, J. Berdine, B. Cook, N. Rinetzky, and M. Sagiv. Local reasoning for storable locks and threads. In *Proc. APLAS'07*, volume 4807 of *LNCS*, pages 19–37. Springer, 2007.

[7] C. Haack and C. Hurlin. Separation logic contracts for a Java-like language with fork/join. In *Proc. AMAST'08*, volume 5140 of *LNCS*, pages 199–215. Springer, 2008.

[8] A. Hobor, A. W. Appel, and F. Z. Nardelli. Oracle semantics for concurrent separation logic. In *Proc. ESOP'08*, volume 4960 of *LNCS*, pages 353–367. Springer, 2008.

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

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

[11] V. Vafeiadis and M. Parkinson. A marriage of rely/guarantee and separation logic. In *Proc. CONCUR'07*, volume 4703 of *LNCS*, pages 256–271. Springer, 2007.

A Proof of soundness for semantics

A.1 Proof of local soundness

To prove local soundness we must show that for any triple $\vdash \{P\}C\{Q\}$, that the following conditions hold:

(a) $\vdash C\ \mathsf{wa}$.

(b) If $\sigma, pr, \gamma \models P$, then $\vdash (C, \sigma, pr, \gamma)\ \mathsf{safe}$.

(c) If $\sigma, pr, \gamma \models P$ and $(C, \sigma, pr, \gamma) \rightsquigarrow^* (\mathbf{skip}, \sigma', pr', \gamma')$, then $\sigma', pr', \gamma' \models Q$.

In proving condition (c) (conformance to post-condition) we omit interference from the environment. As a result of the general assumption that pre- and post-conditions are stable, the pre or post-condition for a triple is always preserved by any environmental interference.

Fork Assume the fork rule holds.

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

By the induction hypothesis, $\models \{P_1\}\, C\, \{P_2\}$, so $\vdash x := \mathbf{fork}_{[P_1,P_2]}\ C\ \mathsf{wa}$ holds. Now consider a configuration such that $\sigma, pr, \gamma \models P_1 * P_3$. It must hold that $\sigma, pr, \gamma \models P_1 * \mathsf{true}$, so the abort rule in the semantics cannot apply. Therefore $\vdash (x := \mathbf{fork}_{[P_1,P_2]}\ C, \sigma, pr, \gamma)\ \mathsf{safe}$ holds.

Suppose we have a configuration $\sigma, pr, \gamma \models P_1 * P_3$ and a configuration $\sigma, pr', \gamma' \models P_1$ such that $pr = pr' \oplus pr'$ and $\gamma = \gamma' \oplus \gamma''$. By the precision of P_1 is must hold that must hold that $\sigma, pr'', \gamma'' \models P_3$. As x is not free in $P_1 * P_3$, it can be reassigned to any value without affecting the satisfaction of P_3. Consequently, the state $\sigma[x \mapsto t], pr'', \gamma''[t \mapsto Q] \models \mathsf{Thread}(x, Q) * P_3$, which suffices to prove the rule sound.

Join Assume the join rule holds.

$$\frac{}{\{P * \mathsf{Thread}(E, P')\}\ \mathbf{join}\ E\ \{P * P'\}}$$

$\vdash \mathbf{join}\ E\ \mathsf{wa}$ always holds by the definition of well annotated. Now consider a configuration such that $\sigma, pr, \gamma \models P * \mathsf{Thread}(E, P')$. By the semantics of Thread, it must hold that $[\![E]\!]_\sigma \in dom(\gamma)$, so the abort rule cannot apply, and $\vdash (\mathbf{join}\ E, \sigma, pr, \gamma)\ \mathsf{safe}$ holds.

Given $[\![E]\!]_\sigma = t$, it must hold that $\sigma, pr, (\gamma \setminus t) \models P$. Consider a configuration such that $\sigma, pr', \gamma' \models P'$ and configuration $\sigma, pr \oplus pr', (\gamma \setminus t) \oplus \gamma'$ is defined. By the semantics of the assertion language, it must hold that $\sigma, pr \oplus pr', (\gamma \setminus t) \oplus \gamma' \models P * P'$, which suffices to prove the rule sound.

Assignment Assume the assignment rule holds.

$$\frac{P \Rightarrow [E/x]P' \quad \mathsf{allowed}([\![x := E]\!], P)}{\{P\}\ x := E\ \{P'\}}$$

Under the semantics, assignments cannot abort, so $\vdash (x := E, \sigma, pr, \gamma)\ \mathsf{safe}$ holds for any configuration σ, pr, γ. Similarly $\vdash x := E\ \mathsf{wa}$ holds for any assignment statement.

Now consider a configuration such that $\sigma, pr, \gamma \models P$, and $[\![E]\!]_\sigma = n$ and $P \Rightarrow [E/x]P'$. By the premise, it must hold that $\sigma, pr, \gamma \models P'$, so it must also hold that $\sigma[x \mapsto n], pr, \gamma \models P'$. This suffices to prove soundness.

Frame Assume the frame rule holds.

$$\frac{\{P\}\ C\ \{P'\}}{\{P * P_0\}\ C\ \{P' * P_0\}}$$

$\vdash C\ \mathsf{wa}$ holds immediately by the induction hypothesis. Safety for any configuration (C, σ, pr, γ) such that $\sigma, pr, \gamma \models P * P_0$ follows immediately from the following lemma.

Lemma 20 (Failure monotonicity). *If* $\vdash (C,\sigma,pr,\gamma)$ *safe, then* $\vdash (C,\sigma,pr\oplus pr'',\gamma\oplus\gamma'')$ *safe.*

Conformance to the post-condition requires the following lemma.

Lemma 21 (Post-condition monotonicity). *If* $(C,\sigma,pr\oplus pr'',\gamma\oplus\gamma'') \rightsquigarrow (C',\sigma',pr_2,\gamma_2)$ *and* $\vdash (C,\sigma,pr,\gamma)$ *safe, then* $(C,\sigma,pr,\gamma) \rightsquigarrow (C',\sigma',pr_1,\gamma_1)$ *and* $pr_1\oplus pr'' = pr_2$ *and* $\gamma_1\oplus\gamma'' = \gamma_2$.

Consider state such that that $\sigma,pr,\gamma \models P$, and $\sigma,pr'',\gamma'' \models P_0$, and $\sigma,pr\oplus pr'',\gamma\oplus\gamma'' \models P * P_0$. Then by the above lemma, for any transition $(C,\sigma,pr\oplus pr'',\gamma\oplus\gamma'') \rightsquigarrow (C',\sigma',pr_2,\gamma_2)$ it must be true that $\sigma',pr_2,\gamma_2 \models P' * P_0$. This suffices to prove the rule sound.

Other rules Trivially sound.

A.2 Proof that reduction preserves well-formedness

Lemma 22 (Rely only alters state). *If* $(\sigma,\delta) \overset{r}{\Longmapsto} (\sigma',\delta')$*, then* $\delta = \delta'$.

Proof. By inspecting definition. □ □

Lemma 23 (Program actions only alters state according to guarantee). *If* $(C,\sigma,pr,\gamma) \rightsquigarrow (C',\sigma',pr',\gamma')$*, then* $\gamma = \gamma'$ *and* $pr = pr'$.

Proof. By inspecting definition. □ □

Lemma 24 (All actions only alters state according to guarantee). *If* $(C,\sigma,pr,\gamma) \bar{\rightsquigarrow} (C',\sigma',pr',\gamma')$*, then* $\sigma,\sigma' \in pr.G$.

Proof. By inspecting definition. □ □

Definition 25 (Well-formed configuration). *A configuration is well-formed,* $\vdash (\sigma,\delta)$ wf*, iff*

(a) $\delta\downarrow_i$ *is defined;*

(b) $\delta\downarrow_t$ *is defined;*

(c) $\forall t \in dom(\gamma)$*. If* $(C',\sigma,pr',\gamma') \rightsquigarrow^* (\mathbf{skip},\sigma'',pr'',\gamma'')$*, then* $\sigma'',pr'',\gamma'' \models \gamma(t)$*;*

(d) $\vdash C_1$ wa $\wedge \ldots \wedge \vdash C_n$ wa*; and*

(e) $\vdash (C_1,\sigma,pr_1,\gamma_1)$ safe*, …, and* $\vdash (C_n,\sigma,pr_n,\gamma_n)$ safe*.*

where $\delta = \{t_1 \mapsto (C_1,pr_1,\gamma_1),\ldots t_n \mapsto (C_n,pr_n,\gamma_n)\}$*;* $\delta\downarrow_i$ *is defined as* $pr_1\oplus\ldots\oplus pr_n$*; and* $\delta\downarrow_t$ *is defined as* $\gamma_1 \uplus \ldots \uplus \gamma_m$*;*

Lemma 26 (WF weakening). *If* $\vdash (\sigma,\delta\uplus\delta')$ wf*, then* $\vdash \sigma,\delta'$ wf*.*

Proof. Follows from definition. □ □

Lemma 27 (WF preserved by global rely). *If* $\vdash (\sigma,\delta)$ wf *and* $(\sigma,\delta) \overset{r}{\Longmapsto} (\sigma',\delta')$*, then* $\vdash (\sigma',\delta)$ wf*.*

Proof. By lemma 22, we know $\delta = \delta'$, hence (a), (b), and (d) are obviously hold for the new configuration. (e) and (c) are clearly preserved by reduction, hence, the new configuration is well-formed. □ □

Lemma 28. *If* $(\sigma,\sigma') \in pr.G$ *and* $pr * (\delta\downarrow_i)$*, then* $(\sigma,\delta) \overset{r}{\Longmapsto} (\sigma',\delta')$*.*

Proof. From the semantics of we know, if $pr_1 * pr_2$ is defined, then $pr_1.G \subseteq pr_2.R$. This lemma follows directly from this property. □ □

Lemma 29 (Reduction preserves well-formedness)**.** *If* $\vdash (\sigma,\delta)$ wf *and* $(\sigma,\delta) \Longmapsto (\sigma',\delta')$*, then* $\vdash (\sigma',\delta')$ wf*.*

Proof. Assume

$$\vdash (\sigma,\delta)\ \mathsf{wf} \tag{1}$$

$$(\sigma,\delta) \Longmapsto (\sigma',\delta') \tag{2}$$

Prove

$$\vdash (\sigma',\delta')\ \mathsf{wf} \tag{3}$$

Case analysis on the global reduction relation.

First case: Simplifying with Lemmas 22 and 23

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

Using Lemma 26 and 27, and assumption (1), we can prove (a) and (b), and only need to prove the remaining three properties for the configuration (C',σ',pr,γ). These are preserved as they have reduced from a state satisfying them.

Second case: Simplifying with Lemma 22 and inspecting when **join** labels can be generated:

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

Using the operational semantics, we know

$$pr \oplus pr_2 = pr' \tag{4}$$

$$(\gamma \setminus t_2) \oplus \gamma_2 = \gamma' \tag{5}$$

These facts are sufficient to prove (a) and (b) are preserved by this reduction step. The other three properties are trivially preserved by reduction.

Third case:

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

By operational semantics we know

$$pr = pr' \oplus pr_2 \tag{6}$$

$$\gamma = \gamma' \oplus \gamma_2 \tag{7}$$

These facts are sufficient to prove (a) and (b) are preserved by this reduction step. The other three properties are trivially preserved by reduction. □ □

Lemma 30 (WF configurations don't fault)**.** *If* $\vdash (\sigma,\delta)$ wf *then* $\neg((\sigma,\delta) \Longmapsto^{*} \textbf{abort})$.

Proof. We first prove that any single transition from a well-formed configuration can't result in a fault. Then by Lemma 29 the result extends to any sequence of transitions from a well-formed configuration.

Consider the four rules in the global semantics that lead to **abort**. The first rule cannot apply, as for every local configuration (C,σ,pr,γ) in a well-formed global configuration, it must hold that $\vdash (C,\sigma,pr,\gamma)$ safe.

For the second rule, using Lemma 24, we know for any transition $(C,\sigma,pr,\gamma) \stackrel{-}{\leadsto} (C',\sigma',pr',\gamma')$, that $\sigma,\sigma' \in pr.G$. Therefore, by the well-formedness of global configuration (σ,δ), and Lemma 28, it must hold that $(\sigma,\delta) \stackrel{r}{\Longmapsto} (\sigma',\delta')$. Consequently the second rule does not apply.

For the third rule, for the negated premise to hold, we must have $\sigma,pr,\gamma \not\models \gamma(t)$, but this is guaranteed by the well-formedness condition, so this rule does not apply. □ □

A.3 Proof of global soundness

We want to prove that if $\vdash \{P\}C\{Q\}$ and $\sigma,1,\emptyset \models P$, then

(a) $\neg((\sigma,[t \mapsto C,1,\emptyset]) \Longmapsto^{*} \textbf{abort})$; and

(b) if $(\sigma,[t \mapsto C,1,\emptyset]) \Longmapsto^{*} (\sigma',[t \mapsto \textbf{skip},pr,\gamma])$ then $\sigma',pr,\gamma \models Q$.

By Theorem 18 we know that (1) $\vdash C$ wa. (2) If $\sigma,pr,\gamma \models P$, then $\vdash (C,\sigma,pr,\gamma)$ safe. (3) If $\sigma,pr,\gamma \models P$ and $(C,\sigma,pr,\gamma) \leadsto^{*} (\textbf{skip},\sigma',pr',\gamma')$, then $\sigma',pr',\gamma' \models Q$. Consequently, we know that $\vdash (\sigma,[t \mapsto C,1,\emptyset])$ wf, which by Lemma 30 proves (a). To prove (b), we first prove the following lemma.

Lemma 31. *If* $\vdash (\sigma,\delta \uplus [t \mapsto C,pr,\gamma])$ wf *and there exists a transition*

$$(\sigma,\delta \uplus [t \mapsto C,pr,\gamma]) \Longmapsto (\sigma',\delta' \uplus [t \mapsto C'',pr'',\gamma''])$$

then $(C,\sigma,pr,\gamma) \leadsto (C'',\sigma',pr'',\gamma'')$.

Proof. Follows from definition. □ □

As $(\sigma,[t \mapsto C,1,\emptyset])$ does not include t in any thread queue, no other thread can join on t (consequence of the local fork rule in the semantics). Consequently Lemma 31 and (3) implies (b).

B Machine semantics and proof-construct erasure

The program configurations used in the operational semantics in Fig. 8 contain *logical* constructs that are introduced to simplify the formulation of the soundness. They can be safely erased when the language is actually implemented. The annotation $[P,Q]$ in the statement $x := \textbf{fork}_{[P,Q]}\ C$ can also be erased. Here we define a simpler *machine semantics* of the language, and show that the logical operational semantics in Fig. 8 is consistent with the machine semantics.

The machine semantics is defined in Fig. 9. As well as omitting annotations, a configuration in the machine semantics contains less information than a logical configuration. We define $\widehat{\delta}$ as

$$\frac{[\![E]\!]_\sigma = n}{(x := E, \sigma) \rightsquigarrow_m (\mathbf{skip}, \sigma[x \mapsto n])} \qquad \frac{}{(\mathbf{skip}; C, \sigma) \rightsquigarrow_m (C, \sigma)} \qquad \frac{(C, \sigma) \rightsquigarrow_m (C', \sigma')}{(C; C', \sigma) \rightsquigarrow_m (C'; C', \sigma')}$$

$$\frac{}{(x := \mathbf{fork}\ C, \sigma) \overset{\mathbf{fork}\ (t,C)}{\rightsquigarrow_m} (\mathbf{skip}, \sigma[x \mapsto t])} \qquad \frac{[\![E]\!]_\sigma = t}{(\mathbf{join}\ E, \sigma) \overset{\mathbf{join}\ t}{\rightsquigarrow_m} (\mathbf{skip}, \sigma)}$$

$$\frac{\widehat{\delta}(t) = C \quad (C, \sigma) \rightsquigarrow_m (C', \sigma')}{(\sigma, \widehat{\delta}) \Longmapsto_m (\sigma', \widehat{\delta}[t \mapsto C'])} \qquad \frac{\widehat{\delta}(t) = C \quad (C, \sigma) \overset{\mathbf{fork}\ (t_2,C_2)}{\rightsquigarrow_m} (C', \sigma') \quad t_2 \notin dom(\widehat{\delta})}{(\sigma, \widehat{\delta}) \Longmapsto_m (\sigma', \widehat{\delta}[t \mapsto C'] \uplus [t_2 \mapsto C_2])}$$

$$\frac{\widehat{\delta}(t) = C \quad (C, \sigma) \overset{\mathbf{join}\ t_2}{\rightsquigarrow_m} (C', \sigma') \quad \widehat{\delta}(t_2) = \mathbf{skip}}{(\sigma, \widehat{\delta}) \Longmapsto_m (\sigma', \widehat{\delta}[t \mapsto C'] \setminus t_2\})} \qquad \frac{\widehat{\delta}(t) = C \quad (C, \sigma) \overset{\mathbf{join}\ t_2}{\rightsquigarrow_m} (C', \sigma') \quad t_2 \notin dom(\widehat{\delta})}{(\sigma, \widehat{\delta}) \Longmapsto_m \mathbf{abort}}$$

Figure 9: Machine Semantics

the physical global thread queue. Unlike δ, it does not contain the permission pr or local thread queue γ.

$$\widehat{\delta} \overset{\mathbf{def}}{=} \mathsf{ThreadIDs} \rightharpoonup_{\mathrm{fin}} \mathsf{Stmts}$$

For a global queue δ, we define the erasure function e so that $e(\delta)(t) = C$ if and only if $\delta(t) = (C, pr, \gamma)$. For a logical global configuration (σ, δ), the corresponding machine global configuration is $(\sigma, e(\delta))$.

Lemma 32 (Local soundness)**.** *If* $(C, \sigma, pr, \gamma) \rightsquigarrow (C', \sigma', pr', \gamma')$*, then* $(C, \sigma) \rightsquigarrow_m (C', \sigma')$*.*

Proof. Holds trivially by erasure of proof constructs. □ □

Lemma 33 (Local completeness)**.** *If* $(C, \sigma) \rightsquigarrow_m (C', \sigma')$ *and* $\vdash (C, \sigma, pr, \gamma)$ safe*, then* $(C, \sigma, pr, \gamma) \rightsquigarrow (C', \sigma', pr', \gamma')$*.*

Proof. The result holds trivially when C is **skip**; C'. We prove the remaining cases by structural induction. The result holds immediately from the induction hypothesis when C is $C; C'$. The fork and join cases are dealt with explicitly (with conditions) in Lemma 34 and Lemma 35.

The remaining case is that C is $x := E$. The assignment rule in the machine semantics gives the transition

$$(x := E, \sigma) \rightsquigarrow_m (\mathbf{skip}, \sigma[x \mapsto n])$$

As $\vdash (C, \sigma, pr, \gamma)$ safe, it must hold that $(\sigma, \sigma[x \mapsto n]) \in pr.G$ for $n = [\![E]\!]_\sigma$ (otherwise the configuration could abort). Hence the assignment rule in the logical semantics applies, giving transition

$$(x := E, \sigma, pr, \gamma) \rightsquigarrow (\mathbf{skip}, \sigma[x \mapsto n], pr, \gamma)$$

as required. □ □

Lemma 34 (Local fork completeness)**.** *If* $(C, \sigma) \overset{\mathbf{fork}\ (t,C)}{\rightsquigarrow_m} (C', \sigma')$*, and* $\vdash (C, \sigma, pr, \gamma)$ safe *and* $t \notin dom(\gamma)$ *then* $(C, \sigma, pr, \gamma) \overset{\mathbf{fork}\ (t,C,pr',\gamma')}{\rightsquigarrow} (C', \sigma', pr', \gamma')$*.*

Proof. C must be $\mathbf{fork}_{[P,Q]}\, C'$ for some P,Q. The machine semantics gives a transition

$$(x := \mathbf{fork}\; C, \sigma) \overset{\mathbf{fork}\,(t,C)}{\leadsto_m} (\mathbf{skip}, \sigma[x \mapsto t])$$

It must hold that $\sigma, pr, \gamma \models P * \mathsf{true}$ and $(\sigma, \sigma[x \mapsto t]) \in pr.G$, otherwise the configuration could abort. Consequently we can construct permissions pr', pr'' and queues γ', γ'' such that $\sigma, pr', \gamma' \models P$ and $pr = pr' \oplus pr''$ and $\gamma = \gamma' \uplus \gamma''$. This means the join rule in the logical semantics applies, giving transition

$$(x := \mathbf{fork}_{[P,Q]}\; C, \sigma, pr, \gamma) \overset{\mathbf{fork}\,(t,C,pr',\gamma')}{\leadsto} (\mathbf{skip}, \sigma[x \mapsto t], pr'', \gamma''[t \mapsto Q])$$

as required. □ □

Lemma 35 (Local join completeness)**.** *If* $\vdash (C, \sigma, pr, \gamma)$ safe *and* $t = [\![E]\!]_\sigma$ *and* $\gamma(t) = Q$ *and* $\sigma, pr', \gamma' \models Q$ *and* $pr \oplus pr'$ *is defined, and* $(\gamma \setminus t) \uplus \gamma'$ *is defined, and* $(C, \sigma) \overset{\mathbf{join}\, t}{\leadsto_m} (C', \sigma')$*, then* $(C, \sigma, pr, \gamma) \overset{\mathbf{join}\,(t,C,pr',\gamma')}{\leadsto} (C', \sigma', pr', \gamma')$.

Proof. Assume the join rule in the machine semantics applies, giving transition

$$(\mathbf{join}\; E, \sigma) \overset{\mathbf{join}\, t}{\leadsto_m} (\mathbf{skip}, \sigma)$$

The join rule in the logical semantics must apply as an immediate consequence of the lemma's premises. This gives transition

$$(\mathbf{join}\; E, \sigma, pr, \gamma) \overset{\mathbf{join}\,(t,pr',\gamma')}{\leadsto} (\mathbf{skip}, \sigma, pr \oplus pr', (\gamma \setminus t) \uplus \gamma')$$

as required. □ □

Lemma 36 (Global machine failure)**.** *If* $\vdash \sigma, \delta$ wf *then no transition* $(\sigma, e(\delta)) \Longmapsto_m \mathbf{abort}$ *exists.*

Proof. The machine semantics only includes a single abort rule, based on join. If this rule can apply, then the abort rule in the logical semantics also applies, which is prohibited by well-formedness. Consequently the machine semantics abort rule cannot apply, which proves our result. □ □

Lemma 37 (Global machine soundness)**.** *If* $\vdash (\sigma, \delta)$ wf *and* $(\sigma, \delta) \Longmapsto (\sigma', \delta')$ *then* $(\sigma, e(\delta)) \Longmapsto_m (\sigma', e(\delta'))$.

Proof. There are three non-abort rules in the global logical semantics: a general rule, a fork rule and a join rule. We consider them case-by-case.

Assume the general rule applies. This means there exists a transition

$$(\sigma, [t \mapsto C, pr, \gamma] \uplus \delta) \Longmapsto (\sigma', [t \mapsto C', pr', \gamma'] \uplus \delta')$$

By the premise of the rule there exists transition

$$(C, \sigma, pr, \gamma) \leadsto (C', \sigma', pr', \gamma')$$

so by Lemma 32 we know that there exists a transition $(C,\sigma) \rightsquigarrow_m (C',\sigma')$. Consequently the first global rule of the machine semantics applies, giving a transition

$$(\sigma,\widehat{\delta}) \Longmapsto_m (\sigma',\widehat{\delta}[t \mapsto C'])$$

as required.

Assume the fork rule applies. This means there exists a transition

$$(\sigma,[t_1 \mapsto C,pr,\gamma] \uplus \delta) \Longmapsto (\sigma',[t \mapsto C',pr',\gamma'] \uplus [t_2 \mapsto C_2,pr_2,\gamma_2] \uplus \delta')$$

By the rule's premise there is a transition

$$(C,\sigma,pr,\gamma) \overset{\textbf{fork}\,(t_2,C_2,pr_2,\gamma_2)}{\rightsquigarrow} (C',\sigma',pr',\gamma')$$

so by Lemma 32 there exists a transition $(C,\sigma) \overset{\textbf{fork}\,(t_2,C_2)}{\rightsquigarrow_m} (C',\sigma')$. By the fact that the disjoint union is defined in the transition, $t_2 \notin dom(\widehat{\delta})$. Consequently the fork rule in the machine semantics applies, giving transition

$$(\sigma,\widehat{\delta}) \Longmapsto_m (\sigma',\widehat{\delta}[t \mapsto C'] \uplus [t_2 \mapsto C_2])$$

as required.

Assume the join rule applies, meaning there is a transition

$$(\sigma,[t_1 \mapsto C,pr,\gamma] \uplus [t_2 \mapsto \textbf{skip},pr_2,\gamma_2] \uplus \delta) \Longmapsto (\sigma',[t \mapsto C',pr',\gamma'] \uplus \delta')$$

By the rule's premise there is a transition

$$(C,\sigma,pr,\gamma) \overset{\textbf{join}\,(t_2,pr_2,\gamma_2)}{\rightsquigarrow} (C',\sigma',pr',\gamma')$$

so by Lemma 32 there is a transition $(C,\sigma) \overset{\textbf{join}\,t_2}{\rightsquigarrow_m} (C',\sigma')$. Consequently the join rule in the machine semantics applies, giving transition

$$(\sigma,\widehat{\delta}) \Longmapsto_m (\sigma',\widehat{\delta}[t \mapsto C'] \setminus t_2\})$$

as required. □ □

Lemma 38 (Global machine completeness)**.** *If* $\vdash \sigma,\delta$ wf *and* $(\sigma,e(\delta)) \Longmapsto_m (\sigma',e(\delta'))$ *then* $(\sigma,\delta) \Longmapsto (\sigma',\delta')$.

Proof. As with the logical semantics, there are three non-abort rules in the global machine semantics: a general rule, a fork rule and a join rule. We consider them by case.

Assume the general rule applies, meaning there is a transition

$$(\sigma,\widehat{\delta}) \Longmapsto_m (\sigma',\widehat{\delta}[t \mapsto C'])$$

By the rule's premise there is a transition $(C,\sigma) \rightsquigarrow_m (C',\sigma')$. By Lemma 33 there exists a corresponding transition in the logical semantics

$$(C,\sigma,pr,\gamma) \rightsquigarrow (C',\sigma',pr',\gamma')$$

The first premise of the general rule in the logical global semantics is satisfied by this transition. The second premise must be satisfied, or the configuration can abort (this holds for the other rules as well). Consequently, there exists a rewrite in the logical semantics

$$(\sigma, [t \mapsto C, pr, \gamma] \uplus \delta) \Longrightarrow (\sigma', [t \mapsto C', pr', \gamma'] \uplus \delta)$$

as required.

Assume the fork rule in the machine global semantics applies. This means there is a transition

$$(\sigma, \widehat{\delta}) \Longrightarrow_m (\sigma', \widehat{\delta}[t \mapsto C'] \uplus [t_2 \mapsto C_2])$$

By the rule's premise there must be a transition $(C, \sigma) \overset{\mathbf{fork}\,(t_2, C_2)}{\rightsquigarrow_m} (C', \sigma')$. We also know $t_2 \notin dom(\widehat{\delta})$, which implies that t_2 is also unused by any of the thread queues in δ. Consequently, by Lemma 34, there exists a transition

$$(C, \sigma, pr, \gamma) \overset{\mathbf{fork}\,(t_2, C_2, pr_2, \gamma_2)}{\rightsquigarrow} (C', \sigma', pr', \gamma')$$

Therefore the fork rule in the logical semantics applies, and so there exists a transition

$$(\sigma, [t_1 \mapsto C, pr, \gamma] \uplus \delta) \Longrightarrow (\sigma', [t \mapsto C', pr', \gamma'] \uplus [t_2 \mapsto C_2, pr_2, \gamma_2] \uplus \delta')$$

as required.

Assume the join rule in the machine global semantics applies. This means there is a transition

$$(\sigma, \widehat{\delta}) \Longrightarrow_m (\sigma', \widehat{\delta}[t \mapsto C'] \setminus t_2\})$$

By the rule's premise there is a transition $(C, \sigma) \overset{\mathbf{join}\, t_2}{\rightsquigarrow_m} (C', \sigma')$. It must be true in the corresponding logical state that $\delta(t_2) = \mathbf{skip}, pr_2, \gamma_2$, and $\gamma_2(t) = Q$. By the definition of well-formedness, it must hold that for any other thread identifier $t \in dom(\delta)$ such that $\delta(t') = (C', pr', \gamma')$ that $pr \oplus pr'$ and $(\gamma \setminus t) \uplus \gamma'$ are defined. Consequently, by Lemma 35, there must exist a transition

$$(C, \sigma, pr, \gamma) \overset{\mathbf{join}\,(t, C, pr', \gamma')}{\rightsquigarrow} (C', \sigma', pr', \gamma')$$

This means the join rule in the global semantics applies, giving transition

$$(\sigma, [t_1 \mapsto C, pr, \gamma] \uplus [t_2 \mapsto \mathbf{skip}, pr_2, \gamma_2] \uplus \delta) \Longrightarrow (\sigma', [t \mapsto C', pr', \gamma'] \uplus \delta')$$

as required. □ □

C Other proof rules

Figure 10 shows proof rules used in deny-guarantee reasoning that are substantially the same as the ones used in standard Hoare logic, and which we therefore have not included in the body of the paper.

D Isabelle proofs

This appendix contains the proofs that have been carried out in Isabelle.

$$\frac{\{P_1 * B\}\, C_1\, \{P_2\} \quad \{P_1 * \neg B\}\, C_2\, \{P_2\}}{\{P_1\}\ \textbf{if}\ B\ \textbf{then}\ C_1\ \textbf{else}\ C_2\ \{P_2\}}\ (\text{COND}) \qquad \frac{}{\{P\}\ \textbf{skip}\ \{P\}}\ (\text{SKIP})$$

$$\frac{\{P'\}\, C_1\, \{P''\} \quad \{P''\}\, C_2\, \{P'\}}{\{P\}\, C_1; C_2\, \{P'\}}\ (\text{SEQ}) \qquad \frac{\{P * B\}\, C\, \{P\}}{\{P\}\ \textbf{while}\ B\ \textbf{do}\ C\ \{\neg B * P\}}\ (\text{WHILE})$$

Figure 10: Further Proof Rules

D.1 Viktor's helper tactics

theory *VHelper*
imports *Main*
begin

This section contains many trivial theorems, mainly for doing forward reasoning.

definition *default-value*:: *'a* **where** *default-value* ≡ ($\epsilon x.$ *True*)

Forward reasoning rules

lemma *allD*: $[\![\ \forall a.\ P\ a\ ]\!] \Longrightarrow P\ a$
by (*erule allE*)+

lemma *all2D*: $[\![\ \forall a\ b.\ P\ a\ b\ ]\!] \Longrightarrow P\ a\ b$
by (*erule allE*)+

lemma *all3D*: $[\![\ \forall a\ b\ c.\ P\ a\ b\ c\ ]\!] \Longrightarrow P\ a\ b\ c$
by (*erule allE*)+

lemma *all4D*: $[\![\ \forall a\ b\ c\ d.\ P\ a\ b\ c\ d\ ]\!] \Longrightarrow P\ a\ b\ c\ d$
by (*erule allE*)+

lemma *all5D*: $[\![\ \forall a\ b\ c\ d\ e.\ P\ a\ b\ c\ d\ e]\!] \Longrightarrow P\ a\ b\ c\ d\ e$
by (*erule allE*)+

lemmas *impD = mp*

lemma *all-impD*: $[\![\ \forall a.\ P\ a \longrightarrow Q\ a;\ P\ a]\!] \Longrightarrow Q\ a$
by (*drule allD mp*)+

lemma *all2-impD*: $[\![\ \forall a\ b.\ P\ a\ b \longrightarrow Q\ a\ b;\ P\ a\ b]\!] \Longrightarrow Q\ a\ b$
by (*drule allD mp*)+

lemma *all3-impD*: $[\![\ \forall a\ b\ c.\ P\ a\ b\ c \longrightarrow Q\ a\ b\ c;\ P\ a\ b\ c]\!] \Longrightarrow Q\ a\ b\ c$
by (*drule allD mp*)+

lemma *all4-impD*: $[\![\ \forall a\ b\ c\ d.\ P\ a\ b\ c\ d \longrightarrow Q\ a\ b\ c\ d;\ P\ a\ b\ c\ d]\!] \Longrightarrow Q\ a\ b\ c\ d$
by (*drule allD mp*)+

lemma *all5-impD*: $[\![\ \forall a\ b\ c\ d\ e.\ P\ a\ b\ c\ d\ e \longrightarrow Q\ a\ b\ c\ d\ e;\ P\ a\ b\ c\ d\ e]\!] \Longrightarrow Q\ a\ b\ c\ d\ e$

by (*drule allD mp*)+

lemma *imp2D*: $[\![P \longrightarrow Q \longrightarrow R; P; Q]\!] \Longrightarrow R$
by (*drule* (*1*) *mp*)+

lemma *all-imp2D*: $[\![\forall a.\ P\ a \longrightarrow Q\ a \longrightarrow R\ a; P\ a; Q\ a]\!] \Longrightarrow R\ a$
by (*drule allD* | *drule* (*1*) *mp*)+

lemma *all2-imp2D*: $[\![\forall a\ b.\ P\ a\ b \longrightarrow Q\ a\ b \longrightarrow R\ a\ b; P\ a\ b; Q\ a\ b]\!] \Longrightarrow R\ a\ b$
by (*drule allD* | *drule* (*1*) *mp*)+

lemma *all3-imp2D*: $[\![\forall a\ b\ c.\ P\ a\ b\ c \longrightarrow Q\ a\ b\ c \longrightarrow R\ a\ b\ c; P\ a\ b\ c; Q\ a\ b\ c]\!] \Longrightarrow R\ a\ b\ c$
by (*drule allD* | *drule* (*1*) *mp*)+

lemma *all4-imp2D*: $[\![\forall a\ b\ c\ d.\ P\ a\ b\ c\ d \longrightarrow Q\ a\ b\ c\ d \longrightarrow R\ a\ b\ c\ d; P\ a\ b\ c\ d; Q\ a\ b\ c\ d]\!] \Longrightarrow R\ a\ b\ c\ d$
by (*drule allD* | *drule* (*1*) *mp*)+

lemma *all5-imp2D*:
$[\![\forall a\ b\ c\ d\ e.\ P\ a\ b\ c\ d\ e \longrightarrow Q\ a\ b\ c\ d\ e \longrightarrow R\ a\ b\ c\ d\ e; P\ a\ b\ c\ d\ e; Q\ a\ b\ c\ d\ e]\!] \Longrightarrow R\ a\ b\ c\ d\ e$
by (*drule allD* | *drule* (*1*) *mp*)+

lemma *imp3D*: $[\![P \longrightarrow Q \longrightarrow R \longrightarrow S; P; Q; R]\!] \Longrightarrow S$
by (*drule* (*1*) *mp*)+

lemma *imp4D*: $[\![P \longrightarrow Q \longrightarrow R \longrightarrow S \longrightarrow T; P; Q; R; S]\!] \Longrightarrow T$
by (*drule* (*1*) *mp*)+

lemma *imp5D*: $[\![P \longrightarrow Q \longrightarrow R \longrightarrow S \longrightarrow T \longrightarrow U; P; Q; R; S; T]\!] \Longrightarrow U$
by (*drule* (*1*) *mp*)+

end

D.2 Resource algebras

theory *Heaps*
imports *Main VHelper HOL/Real/Rational*
begin

We start with defining resource algebras. These are commutative, cancellative, non-negative partial monoids with a named top element (*H-top*), for which $x \odot$ *H-top* is undefined unless $x =$ *H-zero*.

Normally, $\odot$ is a partial function. We model it as a total function with a separate definedness predicate, *H-def*. As a convention, when $\odot$ is undefined, we assume it returns *H-top* (see *undef-star*). This allows us to state *star-assoc* without any sideconditions.

class *res-algebra* = *ord* +
fixes *H-zero* :: *'a*
and *H-top* :: *'a*
and *H-def* :: *'a* ⇒ *'a* ⇒ *bool*

and *H-star* :: $'a \Rightarrow 'a \Rightarrow 'a$ (**infixl** $\odot$ *100*)
assumes
less-res-def: $(x < y) \longleftrightarrow (x \leq y \wedge x \neq y)$
and *le-res-def*: $(x \leq y) \longleftrightarrow (\exists z.\ \textit{H-def}\ x\ z \wedge x \odot z = y)$
and *def-zero*: *H-def x H-zero*
and *def-comm*: $\textit{H-def}\ x\ y \longleftrightarrow \textit{H-def}\ y\ x$
and *def-assoc*: $[\![\textit{H-def}\ x\ y;\ \textit{H-def}\ x\ z;\ \textit{H-def}\ y\ z]\!] \Longrightarrow$
$\textit{H-def}\ (x \odot y)\ z \longleftrightarrow \textit{H-def}\ x\ (y \odot z)$
and *def-sub*: $\textit{H-def}\ (x \odot y)\ z \Longrightarrow \textit{H-def}\ x\ z$
and *def-top*: $\textit{H-def}\ x\ \textit{H-top} \longleftrightarrow (x = \textit{H-zero})$
and *star-zero*: $x \odot \textit{H-zero} = x$
and *star-comm*: $y \odot x = x \odot y$
and *star-assoc*: $(x \odot y) \odot z = x \odot (y \odot z)$
and *star-canc*: $[\![y \odot x = z \odot x;\ \textit{H-def}\ y\ x;\ \textit{H-def}\ z\ x]\!] \Longrightarrow y = z$
and *star-pos*: $x \odot y = \textit{H-zero} \Longrightarrow x = \textit{H-zero}$
and *top-nonzero*: $\textit{H-zero} \neq \textit{H-top}$
and *undef-star*: $\neg\textit{H-def}\ x\ y \Longrightarrow x \odot y = \textit{H-top}$
begin

Basic simplification rules

lemma *star-top*: $x \odot \textit{H-top} = \textit{H-top}$
apply (*cases H-def x H-top*)
 apply (*simp add*: *def-top trans* [*OF star-comm star-zero*])
by (*erule undef-star*)

lemmas *H-simps*[*simp*] =
def-zero *def-zero* [*THEN def-comm* [*THEN iffD1*]]
def-top *trans* [*OF def-comm def-top*]
star-zero *trans* [*OF star-comm star-zero*]
star-top *trans* [*OF star-comm star-top*]
top-nonzero *top-nonzero*[*THEN not-sym*]

Commutativity/associativity lemmas

lemma *star-left-comm*: $a \odot (b \odot c) = b \odot (a \odot c)$
apply (*simp only*: *def-comm star-assoc* [*THEN sym*])
apply (*simp add*: *def-comm star-comm*)
done

lemma *def-left-comm*:
$[\![\ \textit{H-def}\ a\ b;\ \textit{H-def}\ a\ c;\ \textit{H-def}\ b\ c\ ]\!] \Longrightarrow$
$\textit{H-def}\ a\ (b \odot c) = \textit{H-def}\ b\ (a \odot c)$
apply (*simp only*: *def-assoc* [*THEN sym*])
apply (*simp only*: *star-comm* [*of a b*])
apply (*rule trans* [*OF def-assoc*], *simp-all add*: *def-comm*)
done

Stronger versions of *def-assoc* and *def-left-assoc* for better simplification.

lemma *def-assoc2*:
$[\![\ \textit{H-def}\ x\ y;\ \textit{H-def}\ y\ z\ ]\!] \Longrightarrow \textit{H-def}\ (x \odot y)\ z = \textit{H-def}\ x\ (y \odot z)$

apply (*case-tac H-def x z*, *erule* (2) *def-assoc*)
apply (*rule iffI*)
apply (*erule contrapos-np*, *erule def-sub*)
apply (*erule contrapos-np*, *rule def-comm* [*THEN iffD1*])
apply (*rule-tac y=y* **in** *def-sub*)
apply (*simp add*: *def-comm star-comm*)
done

lemma *def-left-comm2*:
$\llbracket$ *H-def a c*; *H-def b c* $\rrbracket \Longrightarrow$
H-def a $(b \odot c)$ = *H-def b* $(a \odot c)$
apply (*case-tac H-def a b*, *rule def-left-comm*, *assumption+*)
apply (*rule iffI*)
apply (*erule contrapos-np*, *rule def-comm* [*THEN iffD1*],
rule-tac y=c **in** *def-sub*, *simp add*: *def-comm star-comm*)
apply (*erule contrapos-np*, *rule-tac y=c* **in** *def-sub*, *simp add*: *def-comm star-comm*)
done

lemmas *H-ac* =
def-comm def-assoc2 def-left-comm2
star-comm star-assoc star-left-comm

Lemmas about *H-top*

lemma *not-topD*:
$x \odot y \neq$ *H-top* $\Longrightarrow$ *H-def x y* $\wedge\ x \neq$ *H-top* $\wedge\ y \neq$ *H-top*
apply (*intro conjI*)
apply (*erule contrapos-np*, *erule undef-star*)
apply (*erule contrapos-nn*, *simp add*: *star-top*)
apply (*erule contrapos-nn*, *simp add*: *star-top*)
done

lemma *not-topD3*: $\llbracket\ x \odot (y \odot z) \neq$ *H-top* $\rrbracket \Longrightarrow$
H-def x y $\wedge$ *H-def x z* $\wedge$ *H-def y z*
$\wedge\ x \odot y \neq$ *H-top* $\wedge\ x \odot z \neq$ *H-top* $\wedge\ y \odot z \neq$ *H-top*
apply (*subgoal-tac* $(x \odot y) \odot z \neq$ *H-top* $\wedge$ $(x \odot z) \odot y \neq$ *H-top*)
prefer *2* **apply** (*simp add*: *H-ac*)
apply (*drule not-topD*, *clarsimp*)+
done

Cancellation lemmas

lemma *star-canc2*: $\llbracket\ x \odot y = x \odot z$; *H-def x y*; *H-def x z* $\rrbracket \Longrightarrow y = z$
by (*rule star-canc*, *auto simp*: *star-comm def-comm*)

lemma *H-canc3*: *H-def x y* $\Longrightarrow (x = x \odot y) = (y =$ *H-zero*$)$
by (*rule iffI*, *subgoal-tac x* $\odot$ *H-zero* = $x \odot y$, *drule star-canc2*, *simp+*)

lemma *H-canc4*: *H-def y x* $\Longrightarrow (x = y \odot x) = (y =$ *H-zero*$)$
by (*simp only*: *H-ac*, *simp add*: *H-canc3*)

lemma *H-pos*: $(x \odot y = H\text{-}zero) = (x = H\text{-}zero \wedge y = H\text{-}zero)$
apply (*rule iffI*, *rule conjI*)
apply (*rule star-pos*, *assumption*+)
apply (*rule-tac y=x* **in** *star-pos*, (*simp add*: *H-ac*)+)
done

lemma *H-canc1*: $[\![\ H\text{-}def\ x\ y;\ H\text{-}def\ x\ z\ ]\!] \Longrightarrow (x \odot y = x \odot z) = (y = z)$
by (*rule iffI*, *erule star-canc2*, *simp-all*)

lemma *H-canc1a*: $[\![\ H\text{-}def\ y\ x;\ H\text{-}def\ z\ x\ ]\!] \Longrightarrow (y \odot x = z \odot x) = (y = z)$
by (*rule iffI*, *erule star-canc*, *simp-all*)

lemma *H-canc2*: $[\![\ H\text{-}def\ y\ x;\ H\text{-}def\ z\ x\ ]\!] \Longrightarrow (x \odot y = z \odot x) = (y = z)$
by (*rule iffI*, *rule-tac x=x* **in** *star-canc*, *simp-all add*: *H-ac*)

lemma *lsym*: $((a = b) = c) \Longrightarrow ((b = a) = c)$ **by** *fast*

lemmas *H-canc* =
H-canc1 H-canc1a
H-canc2 H-canc2[*THEN lsym*]
H-canc3 H-canc3[*THEN lsym*]
H-canc4 H-canc4[*THEN lsym*]
H-pos H-pos[*THEN lsym*]

Lemmas about *H-def*

lemma *def-starD1*:
$H\text{-}def\ (x \odot y)\ z \Longrightarrow H\text{-}def\ x\ z \wedge H\text{-}def\ y\ z \wedge (z = H\text{-}zero \vee H\text{-}def\ x\ y)$
apply (*rule conjI*, *erule def-sub*)
apply (*rule conjI*, *rule-tac y=x* **in** *def-sub*, *simp add*: *H-ac*)
apply (*clarify*, *drule undef-star*, *simp*)
done

lemma *def-starD2*:
$H\text{-}def\ z\ (x \odot y) \Longrightarrow H\text{-}def\ x\ z \wedge H\text{-}def\ y\ z \wedge (z = H\text{-}zero \vee H\text{-}def\ x\ y)$
by (*drule def-comm* [*THEN iffD1*], *erule def-starD1*)

lemma *def-starE*[*elim*]:
$H\text{-}def\ (x \odot y)\ z \Longrightarrow H\text{-}def\ x\ z$
$H\text{-}def\ (x \odot y)\ z \Longrightarrow H\text{-}def\ y\ z$
$H\text{-}def\ z\ (x \odot y) \Longrightarrow H\text{-}def\ z\ x$
$H\text{-}def\ z\ (x \odot y) \Longrightarrow H\text{-}def\ z\ y$
$H\text{-}def\ z\ (x \odot y) \Longrightarrow H\text{-}def\ x\ z$
$H\text{-}def\ z\ (x \odot y) \Longrightarrow H\text{-}def\ y\ z$
apply (*drule def-starD1*, *simp*)+
apply (*drule def-starD2*, *simp add*: *H-ac*)+
done

lemma *def-starE1*:
$H\text{-}def\ z\ (x \odot y) \Longrightarrow H\text{-}def\ z\ x$

by (*drule def-starD2*, *simp add*: *H-ac*)

lemma *def-starE2*[*elim*]:
$[\![H\text{-}def\ x\ (y \odot z);\ H\text{-}def\ y\ z]\!] \Longrightarrow H\text{-}def\ (x \odot y)\ z$
$[\![H\text{-}def\ x\ (y \odot z);\ H\text{-}def\ y\ z]\!] \Longrightarrow H\text{-}def\ (x \odot z)\ y$
$[\![H\text{-}def\ x\ (y \odot z);\ H\text{-}def\ y\ z]\!] \Longrightarrow H\text{-}def\ (y \odot x)\ z$
$[\![H\text{-}def\ x\ (y \odot z);\ H\text{-}def\ y\ z]\!] \Longrightarrow H\text{-}def\ (z \odot x)\ y$
$[\![H\text{-}def\ x\ (y \odot z);\ H\text{-}def\ y\ z]\!] \Longrightarrow H\text{-}def\ y\ (x \odot z)$
$[\![H\text{-}def\ x\ (y \odot z);\ H\text{-}def\ y\ z]\!] \Longrightarrow H\text{-}def\ z\ (x \odot y)$
$[\![H\text{-}def\ x\ (y \odot z);\ H\text{-}def\ y\ z]\!] \Longrightarrow H\text{-}def\ y\ (z \odot x)$
$[\![H\text{-}def\ x\ (y \odot z);\ H\text{-}def\ y\ z]\!] \Longrightarrow H\text{-}def\ z\ (y \odot x)$
apply (*rule def-assoc2* [*THEN iffD2*], *erule def-starE*,
simp add: *H-ac*, *simp add*: *H-ac*)+
apply (*subst star-comm*, *rule def-assoc2* [*THEN iffD2*],
erule def-starE, *simp add*: *H-ac*, *simp add*: *H-ac*)+
apply (*subst star-comm*, *rule def-assoc2* [*THEN iffD1*],
(*erule def-starE*|*simp add*: *H-ac*)+)+
done

Lemmas about ≤ and <

subclass *order*
apply (*unfold-locales*, *simp-all add*: *less-res-def le-res-def*)
apply (*rule-tac x=H-zero* **in** *exI*, *simp*)
apply (*clarsimp*, *rename-tac y z*)
apply (*rule-tac* $x{=}y \odot z$ **in** *exI*, *frule def-starD1*, *simp add*: *H-ac*)
apply (*clarsimp*, *frule def-starD1*, *simp add*: *H-ac H-canc*)
done

lemma *le-starI1*[*elim*]: $H\text{-}def\ x\ y \Longrightarrow x \le x \odot y$
by (*simp add*: *le-res-def*, *rule-tac x=y* **in** *exI*, *simp add*: *H-ac*)

lemma *le-starI2*[*elim*]: $H\text{-}def\ y\ x \Longrightarrow x \le y \odot x$
by (*simp add*: *le-res-def*, *rule-tac x=y* **in** *exI*, *simp add*: *H-ac*)

lemma *star-mon*: $[\![\ x \le x';\ y \le y';\ H\text{-}def\ x'\ y'\ ]\!] \Longrightarrow x \odot y \le x' \odot y'$
apply (*subgoal-tac H-def x y*) **prefer** *2*
apply (*clarsimp simp add*: *le-res-def*, *rule def-starE*, *erule def-starE*)
apply (*clarsimp simp add*: *le-res-def*, *rename-tac z w*)
apply (*rule-tac* $x{=}z \odot w$ **in** *exI*, *simp add*: *H-ac*)
apply (*frule def-starD1*, *clarsimp simp add*: *H-ac*)
apply *fast*
done

end

D.2.1 Simple instantiations

This section considers some simple instantiations of permission algebras:

- Booleans

- Options
- Products
- Functions

Booleans

instantiation *bool* :: *res-algebra*
begin

definition *H-zero* ≡ *False*
definition *H-top* ≡ *True*
definition *H-def x y* ≡ ¬*x* ∨ ¬*y*
definition *x* ⊙ *y* ≡ *x* ∨ *y*

instance by (*intro-classes*)
(*auto simp add*: *H-zero-bool-def H-top-bool-def*
H-def-bool-def H-star-bool-def less-bool-def le-bool-def)

end

Options

instantiation *option* :: (*type*) *res-algebra*
begin

definition *H-zero* ≡ *None*
definition *H-top* ≡ *Some default-value*
definition *H-def x y* ≡ (*x* = *None* ∨ *y* = *None*)
definition *x* ⊙ *y* ≡ *case x of None* ⇒ *y* | *Some* - ⇒
(*case y of None* ⇒ *x* | *Some* - ⇒ *H-top*)
definition *x* ≤ *y* ≡ *case x of None* ⇒ *True* | *Some* - ⇒
(*case y of None* ⇒ *False* | *Some* - ⇒ *x*=*y*)
definition *x* < *y* ≡ *case x of None* ⇒
(*case y of None* ⇒ *False* | *Some* - ⇒ *True*)
| *Some* - ⇒ *False*

instance by (*intro-classes*)
(*auto split*: *option.splits*
simp add: *H-zero-option-def H-top-option-def*
H-def-option-def H-star-option-def
less-option-def less-eq-option-def)
end

Products

The cartesian product of two permission algebras is a permission algebra, where the product operations are the straightforward liftings.

instantiation $*$ :: (*res-algebra*, *res-algebra*) *res-algebra*
begin

definition *H-zero* ≡ (*H-zero*, *H-zero*)
definition *H-top* ≡ (*H-top*, *H-top*)
definition *H-def x y* ≡ (*H-def* (*fst x*) (*fst y*) ∧ *H-def* (*snd x*) (*snd y*))
definition $x \odot y$ ≡ *if H-def x y then* (*fst x* ⊙ *fst y*, *snd x* ⊙ *snd y*) *else H-top*
definition $x \leq y$ ≡ (*fst x* ≤ *fst y* ∧ *snd x* ≤ *snd y*)
definition $x < y$ ≡ $(x \leq y \wedge x \neq (y{::}'a \times 'b))$

instance
apply (*intro-classes*)
apply (*simp add*: *less-prod-def*)
apply (*clarsimp cong*: *conj-cong simp add*: *less-eq-prod-def H-def-prod-def H-star-prod-def le-res-def*, *fast*)
apply (*simp add*: *H-def-prod-def H-zero-prod-def*)
apply (*simp add*: *H-def-prod-def conj-ac H-ac*)
apply (*simp add*: *H-star-prod-def H-def-prod-def conj-ac H-ac*)
apply (*simp split*: *split-if-asm add*: *H-star-prod-def H-def-prod-def H-top-prod-def*, *fast elim*: *def-sub*)
apply (*clarsimp simp add*: *H-def-prod-def H-star-prod-def H-zero-prod-def H-top-prod-def*)
apply (*simp add*: *H-def-prod-def H-star-prod-def H-zero-prod-def*)
apply (*simp add*: *H-def-prod-def H-star-prod-def H-ac*)
defer — Postpone the associativity proof
apply (*clarsimp simp add*: *H-def-prod-def H-star-prod-def H-ac H-canc*)
apply (*clarsimp split*: *split-if-asm simp add*: *H-def-prod-def H-star-prod-def H-top-prod-def H-zero-prod-def H-ac H-canc*)
apply (*simp add*: *H-top-prod-def H-zero-prod-def H-ac*)
apply (*simp add*: *H-star-prod-def*)
— Finally, prove associativity.
apply (*clarsimp simp add*: *H-def-prod-def H-zero-prod-def H-top-prod-def H-star-prod-def H-ac H-canc*)
apply (*auto simp add*: *H-ac*)
apply (*erule contrapos-np*, *erule def-starE*)+
done

end

Functions

A function from any type into a permission algebra is a permission algebra, where the operations are the straightforward liftings.

instantiation *fun* :: (*type*, *res-algebra*) *res-algebra*
begin

definition *H-zero* ≡ (λx. *H-zero*)
definition *H-top* ≡ (λx. *H-top*)
definition *H-def f g* ≡ ($\forall x$. *H-def* (*f x*) (*g x*))
definition $f \odot g$ ≡ *if H-def f g then* (λx. *f x* ⊙ *g x*) *else H-top*

instance
apply (*intro-classes*)
apply (*simp add*: *less-fun-def*)
apply (*simp cong*: *conj-cong add*: *le-fun-def H-star-fun-def H-def-fun-def*
    expand-fun-eq le-res-def, *metis*)
apply (*simp-all cong*: *if-cong add*: *H-zero-fun-def H-top-fun-def*
    H-def-fun-def H-star-fun-def H-ac H-canc)
apply (*clarsimp split*: *split-if-asm*, *erule allE*, *drule def-starD2*, *erule* (*1*) *conjE*)
apply (*fast intro*: *ext*)
— Associativity
apply (*rule conjI*)
 apply (*clarify*, *rule conjI*)
  apply (*clarify*, *rename-tac a*, (*erule-tac x=a* **in** *allE*)+, *frule def-starD2*,
    simp add: *H-ac*)
 apply (*clarify*, *rename-tac a*, *rule conjI*)
  apply (*clarify*, (*erule-tac x=a* **in** *allE*)+, *simp*)
 apply (*clarify*, (*erule-tac x=a* **in** *allE*)+, *frule def-starD2*, *simp add*: *H-ac*)
apply (*clarify*, *rule conjI*, *rename-tac foo*)
 apply (*clarify*, (*erule-tac x=foo* **in** *allE*)+, *frule def-starD2*, *simp add*: *H-ac*)
apply (*clarify*, *rename-tac foo*)
apply ((*erule-tac x=foo* **in** *allE*)+, *simp*)
— Cancellation
apply (*rule ext*, *rename-tac a*, *rule-tac x=x a* **in** *star-canc*, *simp add*: *H-ac*)
  apply (*erule fun-cong*, *simp add*: *H-ac*, *simp add*: *H-ac*)
— Non-negative
apply (*simp split*: *split-if-asm add*: *expand-fun-eq H-ac H-canc*)
— top nonzero
apply (*clarify*, *drule fun-cong*, *simp*)
— undef star
apply (*fast*)
done

end

D.2.2 Further instantiations

In this section, we consider two further resource algebra instances:

- the standard heap model,
- fractional permissions,
- tagged permissions,
- biased pair permissions, and
- products with splitting.

Tagged permissions are used to encode rely-guarantee, whereas biased pair permissions are used to encode existence permissions.

Heap model

We follow a standard representation of heaps as a partial function from addresses to values. We take addresses to be natural numbers, and are polymorphic with respect to values.

datatype *'a heap = myH nat ⇒ 'a option*

instantiation *heap* :: (*type*) *res-algebra*
begin

definition *H-zero ≡ myH H-zero*
definition *H-top ≡ myH H-top*
definition *H-def h1 h2 ≡ case* (*h1*,*h2*) *of* (*myH m1*, *myH m2*) ⇒ *H-def m1 m2*
definition *h1 ⊙ h2 ≡ case* (*h1*,*h2*) *of* (*myH m1*, *myH m2*) ⇒ *myH* (*m1 ⊙ m2*)
definition *h1 ≤ h2 ≡ case* (*h1*,*h2*) *of* (*myH m1*, *myH m2*) ⇒ *m1 ≤ m2*
definition *h1 < h2 ≡ case* (*h1*,*h2*) *of* (*myH m1*, *myH m2*) ⇒ *m1 < m2*

instance
apply (*intro-classes*)
apply (*auto split*: *heap.splits*
    simp add: *H-zero-heap-def H-top-heap-def*
     H-def-heap-def H-star-heap-def
     less-heap-def less-eq-heap-def
     le-res-def less-res-def H-ac H-canc
    elim: *undef-star*)
apply (*case-tac z*, *fast*)
done

Here are alternative (equivalent) definitions for operations on heaps.

lemma *H-def-heap-def2*:
 H-def (*myH m1*) (*myH m2*) = (*dom m1 ∩ dom m2* = {})
by (*simp add*: *H-def-heap-def H-def-fun-def H-def-option-def Int-def dom-def*)

lemma *H-star-heap-def2*:
 (*myH h1*) ⊙ (*myH h2*) = (*if H-def h1 h2 then myH* (*h1* ++ *h2*) *else H-top*)
apply (*simp split*: *option.splits*
    add: *H-top-heap-def H-def-heap-def*
      H-star-heap-def H-star-fun-def H-star-option-def
      expand-fun-eq map-add-def)
apply (*clarsimp simp add*: *H-def-fun-def H-def-option-def*)
apply (*erule-tac x=x* **in** *allE*, *erule disjE*, *simp*, *simp*)
done

end

definition *H-singl* :: *nat ⇒ 'a ⇒ 'a heap* **where**
 H-singl x y ≡ myH (*λz. if z = x then Some y else None*)

Fractional permissions

Fractional permissions are rational numbers in the range [0,1].

typedef *myfrac* = $\{x::rat.\ 0 \le x \wedge x \le 1\}$
by (*rule-tac x=0* **in** *exI*, *simp*)

declare
Rep-myfrac-inverse [*simp*]
Abs-myfrac-inject [*simp*]
Abs-myfrac-inverse [*simp*]
myfrac-def [*simp*]

instantiation *myfrac* :: *res-algebra*
begin

definition *H-zero* ≡ *Abs-myfrac 0*
definition *H-top* ≡ *Abs-myfrac 1*
definition *H-def x y* ≡ (*Rep-myfrac x* + *Rep-myfrac y* ≤ *1*)
definition $x \odot y$ ≡ *if H-def x y then Abs-myfrac* (*Rep-myfrac x* + *Rep-myfrac y*) *else H-top*
definition $x \le (y::myfrac) \equiv (\exists z.\ H\text{-}def\ x\ z \wedge x \odot z = y)$
definition $x < (y::myfrac) \equiv (x \le y \wedge x \ne y)$

instance
apply (*intro-classes*)
apply (*simp-all split*: *split-if-asm add*: *less-myfrac-def less-eq-myfrac-def*
H-zero-myfrac-def H-top-myfrac-def H-def-myfrac-def H-star-myfrac-def)
apply (*rule-tac* [*1−11*] *x=x* **in** *Abs-myfrac-cases*,
case-tac [*6*] *Rep-myfrac x* ≤ *0*, *simp-all add*: *add-ac*)
apply (*rule-tac* [*1−5*] *x=y* **in** *Abs-myfrac-cases*,
rule-tac [*1−5*] *x=z* **in** *Abs-myfrac-cases*, *simp-all add*: *add-ac*)
done

end

A useful fraction is one-half (1/2).

definition *H-half* :: *myfrac* ⇒ *myfrac* **where**
H-half x ≡ *Abs-myfrac* (*Fract 1 2* ∗ *Rep-myfrac x*)

lemma *H-half-inject*[*simp*]:
(*H-half x* = *H-half y*) ⟷ (*x* = *y*)
apply (*rule iffI*, *simp-all add*: *H-half-def*)
apply (*rule-tac x=x* **in** *Abs-myfrac-cases*,
rule-tac x=y **in** *Abs-myfrac-cases*)
apply (*clarsimp*, *rename-tac x y*)
apply (*rule-tac q=x* **in** *Rat-cases*,
rule-tac q=y **in** *Rat-cases*)
apply (*simp add*: *mult-rat eq-rat le-rat zero-rat one-rat*)
done

lemma *mult2-rat*:
  y ≠ 0 ⟹ 2 ∗ Fract x y = Fract (*2 ∗ x*) *y*
by (*subst* (*1*) *mult-2*, *subst add-rat*, *simp-all add*: *eq-rat*)

lemma *H-def-half* [*simp*]:
  H-def (*H-half x*) (*H-half x*)
apply (*simp add*: *H-half-def H-def-myfrac-def*)
apply (*rule-tac x=x* **in** *Abs-myfrac-cases*, *clarsimp*, *rename-tac x*)
apply (*rule-tac q=x* **in** *Rat-cases*)
apply (*simp add*: *mult-rat mult2-rat le-rat zero-rat one-rat*)
done

lemma *H-star-half* [*simp*]:
  (*H-half x ⊙ H-half x*) = *x*
apply (*simp add*: *H-star-myfrac-def*, *simp add*: *H-half-def*)
apply (*rule-tac x=x* **in** *Abs-myfrac-cases*, *clarsimp*, *rename-tac x*)
apply (*rule-tac q=x* **in** *Rat-cases*)
apply (*simp add*: *mult-rat mult2-rat le-rat eq-rat zero-rat one-rat*)
done

Tagged permissions

Tagged permissions are pairs consisting of a tag and a resource algebra, where *H-zero* and *H-top* must have the default tag.

typedef (*'a*,*'b*) *permTag* =
  {(*p*::*'a* × (*'b*::*res-algebra*)).
        (*fst p = default-value ∧ snd p = H-zero*)
      ∨ (*fst p = default-value ∧ snd p = H-top*)
      ∨ (*snd p ≠ H-zero ∧ snd p ≠ H-top*)}
by (*rule-tac x*=(*default-value*, *H-zero*) **in** *exI*, *simp*)

lemma *permTag-simps*:
  (*default-value*, *H-top*) ∈ *permTag*
  (*default-value*, *H-zero*) ∈ *permTag*
by (*simp-all add*: *permTag-def*)

definition *permTag-def* :: *'a* × (*'b*::*res-algebra*) ⇒ *'a* × *'b* ⇒ *bool* **where**
  permTag-def x y ≡
    case x of (*x1*,*x2*) ⇒
    case y of (*y1*,*y2*) ⇒
     (*x1 = default-value ∧ x2 = H-zero*)
      ∨ (*y1 = default-value ∧ y2 = H-zero*)
      ∨ (*x1 = y1 ∧ H-def x2 y2*)

definition *permTag-star* :: *'a* × (*'b*::*res-algebra*) ⇒ *'a* × *'b* ⇒ *'a* × *'b* **where**
  permTag-star x y ≡
     case x of (*x1*,*x2*) ⇒
     case y of (*y1*,*y2*) ⇒
     if (*x1 = default-value ∧ x2 = H-zero*) *then* (*y1*, *y2*)

else if (*y1* = *default-value* ∧ *y2* = *H-zero*) *then* (*x1*, *x2*)
else if (*x1* = *y1* ∧ *x2* ⊙ *y2* ≠ *H-top*) *then* (*x1*, *x2* ⊙ *y2*)
else (*default-value*, *H-top*)

lemma *permTag-def-unit*:
x ∈ *permTag* ⟹ *permTag-def x* (*default-value*, *H-zero*)
by (*simp split*: *prod.splits add*: *permTag-def-def permTag-def*)

lemma *permTag-def-top*:
x ∈ *permTag* ⟹
permTag-def x (*default-value*, *H-top*) ⟷ (*x* = (*default-value*, *H-zero*))
by (*simp split*: *prod.splits add*: *permTag-def-def permTag-def*)

lemma *permTag-def-comm*:
permTag-def x y = *permTag-def y x*
by (*simp add*: *permTag-def-def*, *auto simp add*: *H-ac*)

lemma *permTag-def-assoc*:
⟦ *x* ∈ *permTag*; *y* ∈ *permTag*; *z* ∈ *permTag*;
permTag-def x y; *permTag-def x z*; *permTag-def y z* ⟧ ⟹
permTag-def (*permTag-star x y*) *z* = *permTag-def x* (*permTag-star y z*)
apply (*case-tac x*, *case-tac y*, *case-tac z*, *rename-tac x1 x2 y1 y2 z1 z2*)
apply (*simp split*: *prod.splits cong*: *let-weak-cong add*: *permTag-def permTag-def-def*)
apply (*clarsimp simp add*: *H-ac*)
apply (*simp add*: *permTag-star-def H-ac H-canc*)
apply (*case-tac x1* = *default-value* ∧ *x2* = *H-zero*, *simp-all add*: *H-ac*)
apply (*case-tac y1* = *default-value* ∧ *y2* = *H-zero*, *simp-all add*: *H-ac*)
apply (*case-tac z1* = *default-value* ∧ *z2* = *H-zero*, *simp-all add*: *H-ac*)
apply (*erule disjE*, *simp add*: *H-ac*)+
apply (*clarsimp split*: *split-if-asm simp add*: *H-ac H-canc*)
apply (*simp add*: *def-assoc* [*THEN sym*])
done

lemma *permTag-star1*:
⟦ *x* ∈ *permTag*; *y* ∈ *permTag* ⟧ ⟹ *permTag-star x y* ∈ *permTag*
by (*case-tac x*, *case-tac y*)
(*clarsimp split*: *prod.splits*
simp add: *permTag-def permTag-star-def H-ac H-canc*)

lemma *permTag-star-comm*:
⟦ *x* ∈ *permTag*; *y* ∈ *permTag* ⟧ ⟹ *permTag-star x y* = *permTag-star y x*
by (*case-tac x*, *case-tac y*)
(*clarsimp split*: *prod.splits*
simp add: *permTag-def permTag-star-def H-ac*)

lemma *permTag-star-top-unit*:
permTag-star x (*default-value*,*H-top*) = (*default-value*, *H-top*)
permTag-star (*default-value*,*H-top*) *x* = (*default-value*, *H-top*)
permTag-star x (*default-value*,*H-zero*) = *x*

permTag-star (*default-value*,*H-zero*) *x* = *x*
by (*simp-all split*: *prod.splits add*: *permTag-star-def star-top*)

lemma *permTag-star-assoc*:
⟦ *x* ∈ *permTag*; *y* ∈ *permTag*; *z* ∈ *permTag* ⟧ ⟹
permTag-star (*permTag-star x y*) *z* = *permTag-star x* (*permTag-star y z*)
apply (*case-tac x*, *case-tac y*, *case-tac z*, *rename-tac x1 x2 y1 y2 z1 z2*)
apply (*clarsimp simp add*: *permTag-def*)
apply ((*erule disjE*, *simp add*: *permTag-star-top-unit*)+, *clarify*)
apply (*clarsimp simp add*: *permTag-star-def H-ac H-canc*)
apply (*case-tac y2* ⊙ *z2* = *H-top*, *clarsimp simp add*: *H-ac H-canc*)
apply (*clarsimp simp add*: *H-ac H-canc star-top*)
apply (*auto simp add*: *star-assoc* [*THEN sym*] *star-top*)
done

lemma *permTag-star-canc*:
⟦ *permTag-star y x* = *permTag-star z x*;
x ∈ *permTag*; *y* ∈ *permTag*; *z* ∈ *permTag*;
permTag-def y x; *permTag-def z x* ⟧
⟹ *y* = *z*
apply (*case-tac x*, *rename-tac x1 x2*)
apply (*simp split*: *prod.splits add*: *permTag-def permTag-def-def permTag-star-def*)
apply (*simp split*: *split-if-asm add*: *H-ac H-canc*)
apply (*rule-tac x=x2* **in** *star-canc*, *simp-all add*: *H-ac*)
done

instantiation *permTag* :: (*type*,*res-algebra*) *res-algebra*
begin

definition *H-zero* ≡ *Abs-permTag* (*default-value*, *H-zero*)
definition *H-top* ≡ *Abs-permTag* (*default-value*, *H-top*)
definition *H-def x y* ≡ *permTag-def* (*Rep-permTag x*) (*Rep-permTag y*)
definition *x* ⊙ *y* ≡ *Abs-permTag* (*permTag-star* (*Rep-permTag x*) (*Rep-permTag y*))
definition *x* ≤ (*y*::(*'a*,*'b*) *permTag*) ≡ (∃*z*. *H-def x z* ∧ *x* ⊙ *z* = *y*)
definition *x* < (*y*::(*'a*,*'b*) *permTag*) ≡ (*x* ≤ *y* ∧ *x* ≠ *y*)

instance
apply (*intro-classes*)
apply (*simp add*: *less-permTag-def*)
apply (*simp add*: *less-eq-permTag-def*)
apply (*simp-all add*: *H-zero-permTag-def H-top-permTag-def H-def-permTag-def*
H-star-permTag-def Abs-permTag-inverse permTag-def)
apply (*simp-all add*: *Abs-permTag-inverse Abs-permTag-inject permTag-star1*
Rep-permTag permTag-simps)
apply (*rule permTag-def-unit* [*OF Rep-permTag*])
apply (*rule permTag-def-comm*)
apply (*rule permTag-def-assoc*, (*rule Rep-permTag*)+, *assumption*+)
apply (*rule-tac x=x* **in** *Abs-permTag-cases*,
rule-tac x=y **in** *Abs-permTag-cases*,

rule-tac x=z **in** *Abs-permTag-cases*,
clarsimp simp add: *permTag-def-top Abs-permTag-inverse*
Abs-permTag-inject permTag-simps,
clarsimp split: *split-if-asm*
simp add: *permTag-def permTag-star-def permTag-def-def H-ac H-canc*,
case-tac b=H-zero, *simp*, *clarsimp*, *case-tac bb=H-zero*, *simp*, *clarsimp*,
drule def-starD2, *simp*)
apply (*rule-tac x=x* **in** *Abs-permTag-cases*,
simp add: *permTag-def-top Abs-permTag-inverse Abs-permTag-inject permTag-simps*)
apply (*rule-tac x=x* **in** *Abs-permTag-cases*,
simp add: *permTag-def-top Abs-permTag-inverse Abs-permTag-inject permTag-simps*)
apply (*simp add*: *permTag-star-top-unit Rep-permTag-inverse Rep-permTag*)
apply (*simp add*: *permTag-star-comm Rep-permTag*)
apply (*simp add*: *permTag-star-assoc Rep-permTag*)
apply (*rule Rep-permTag-inject* [*THEN iffD1*],
erule permTag-star-canc, *simp-all add*: *Rep-permTag*)
apply (*rule-tac x=x* **in** *Abs-permTag-cases*, *clarify*, *rename-tac x1 x2*,
rule-tac x=y **in** *Abs-permTag-cases*, *clarify*, *rename-tac y1 y2*,
simp split: *split-if-asm*
add: *permTag-def-def Abs-permTag-inverse permTag-star-def H-canc*)
apply (*rule-tac x=x* **in** *Abs-permTag-cases*, *clarify*, *rename-tac x1 x2*,
rule-tac x=y **in** *Abs-permTag-cases*, *clarify*, *rename-tac y1 y2*,
clarsimp split: *prod.splits*
simp add: *permTag-def-def Abs-permTag-inverse permTag-star-def*,
case-tac x2 = H-zero, *simp*, *simp*, *case-tac y2 = H-zero*, *simp*, *clarsimp*,
drule undef-star, *simp*)
done

end

Biased pair permissions

These are used for dispose permissions. The first component is the *address* permission, which asserts that the memory cell exists. The second component is the *value* permission, which asserts something about the value of the memory cell.

typedef (*'a*,*'b*) *dperm* =
{ (*p*::(*'a*::*res-algebra*) × (*'b*::*res-algebra*)).
(*fst p* = *H-zero* ⟶ *snd p* = *H-zero*)
∧ (*fst p* = *H-top* ⟶ *snd p* = *H-top*) }
by (*rule-tac x*=(*H-zero*,*H-zero*) **in** *exI*, *simp*)

lemma [*simp*]:
(*H-zero*, *H-zero*) ∈ *dperm*
(*H-top*, *H-top*) ∈ *dperm*
by (*simp-all add*: *dperm-def conj-ac*)

definition *dperm-def* :: (*'a*::*res-algebra*) × (*'b*::*res-algebra*) ⇒ *'a* × *'b* ⇒ *bool* **where**
dperm-def x y ≡
case x of (*x1*,*x2*) ⇒

case y of $(y1,y2) \Rightarrow$
H-def x1 y1 $\wedge$ *H-def x2 y2* $\wedge$
$(x1 \odot y1 = H\text{-}top \longrightarrow x2 \odot y2 = H\text{-}top)$

definition *dperm-star* :: $('a::res\text{-}algebra) \times ('b::res\text{-}algebra) \Rightarrow 'a \times 'b \Rightarrow 'a \times 'b$ **where**
dperm-star x y $\equiv$
case x of $(x1,x2) \Rightarrow$
case y of $(y1,y2) \Rightarrow$
if H-def x1 y1 $\wedge$ *H-def x2 y2* $\wedge$ $x1 \odot y1 \neq H\text{-}top$ *then*
$(x1 \odot y1, x2 \odot y2)$
else (*H-top*, *H-top*)

lemma *dperm-star1*:
$[\![\, x \in dperm; y \in dperm \,]\!] \Longrightarrow$ *dperm-star x y* $\in$ *dperm*
apply (*case-tac x*, *case-tac y*, *clarify*, *rename-tac x1 x2 y1 y2*)
apply (*clarsimp simp add*: *dperm-def dperm-star-def H-ac H-canc*)
done

lemma *top-derive-zero*:
$[\![\, x \odot y = H\text{-}top; H\text{-}def\ (x \odot y)\ z; z = H\text{-}zero \Longrightarrow P \,]\!] \Longrightarrow P$
by *simp*

lemma *dperm-def-sub*:
$[\![$ *dperm-def* (*dperm-star x y*) *z*; $x \in dperm$; $y \in dperm$; $z \in dperm$ $]\!] \Longrightarrow$ *dperm-def x z*
apply (*case-tac x*, *rename-tac x1 x2*)
apply (*case-tac y*, *rename-tac y1 y2*)
apply (*case-tac z*, *rename-tac z1 z2*)
apply (*clarsimp split*: *split-if-asm*
simp add: *dperm-def dperm-def-def dperm-star-def H-ac*)
apply (*rule conjI*, *erule def-starE*)+
apply (*clarsimp*)
apply (*frule def-starD2*)
apply (*rule-tac z=y1* **in** *top-derive-zero*, *assumption*)
apply (*thin-tac ?x = H-top*, *simp add*: *H-ac*, *simp*)
done

lemma *dperm-star-assoc*:
$[\![x \in dperm; y \in dperm; z \in dperm \,]\!] \Longrightarrow$
dperm-star (*dperm-star x y*) *z* = *dperm-star x* (*dperm-star y z*)
apply (*case-tac x*, *rename-tac x1 x2*)
apply (*case-tac y*, *rename-tac y1 y2*)
apply (*case-tac z*, *rename-tac z1 z2*)
apply (*clarsimp split*: *prod.splits simp add*: *dperm-def-def dperm-def*)
apply (*case-tac x1 = H-zero*, *clarsimp simp add*: *dperm-star-def*, *simp*)
apply (*case-tac x1 = H-top*, *clarsimp simp add*: *dperm-star-def*, *simp*)
apply (*case-tac y1 = H-zero*, *clarsimp simp add*: *dperm-star-def*, *simp*)
apply (*case-tac y1 = H-top*, *clarsimp simp add*: *dperm-star-def*, *simp*)
apply (*case-tac z1 = H-zero*, *clarsimp simp add*: *dperm-star-def*, *simp*)
apply (*case-tac z1 = H-top*, *clarsimp simp add*: *dperm-star-def*, *simp*)

```
apply (simp add: dperm-star-def H-ac H-canc)
apply (rule conjI, clarify)
 apply (frule not-topD3, clarsimp simp add: H-ac)
 apply (rule conjI, clarify)
  apply (rule conjI, fast elim: def-starE)
  apply (clarify, frule undef-star, simp)
 apply (clarify, erule contrapos-np, erule def-starE)
apply (clarsimp, rule conjI, clarsimp simp add: H-ac, clarsimp simp add: H-ac)
apply (case-tac H-def x1 (y1 ⊙ z1), clarsimp simp add: H-ac)
 apply (frule not-topD3, clarsimp simp add: H-ac, fast)
apply (clarsimp simp add: H-ac, frule not-topD3, clarsimp simp add: H-ac)
done

instantiation dperm :: (res-algebra,res-algebra) res-algebra
begin

definition  H-zero ≡ Abs-dperm (H-zero, H-zero)
definition  H-top  ≡ Abs-dperm (H-top, H-top)
definition  H-def x y ≡ dperm-def (Rep-dperm x) (Rep-dperm y)
definition  x ⊙ y ≡ Abs-dperm (dperm-star (Rep-dperm x) (Rep-dperm y))
definition x ≤ (y::('a,'b) dperm) ≡ (∃z. H-def x z ∧ x ⊙ z = y)
definition x < (y::('a,'b) dperm) ≡ (x ≤ y ∧ x ≠ y)

instance
apply (intro-classes)
apply (simp add: less-dperm-def)
apply (simp add: less-eq-dperm-def)
apply (simp-all add: H-zero-dperm-def H-top-dperm-def H-def-dperm-def
           H-star-dperm-def Abs-dperm-inverse Abs-dperm-inject
           dperm-star1 Rep-dperm H-ac H-canc)
apply (rule-tac x=x in Abs-dperm-cases,
    clarsimp simp add: Abs-dperm-inverse dperm-def-def dperm-def)
apply (rule-tac x=x in Abs-dperm-cases,
    rule-tac x=y in Abs-dperm-cases,
    clarsimp simp add: Abs-dperm-inverse dperm-def-def dperm-def,
    simp cong: conj-cong add: H-ac H-canc)
— H-def is associative
apply (rule-tac x=x in Abs-dperm-cases,
    rule-tac x=y in Abs-dperm-cases,
    clarsimp simp add: Abs-dperm-inverse dperm-def,
    clarsimp simp add: dperm-star-def dperm-def-def H-ac H-canc star-top)
apply (rule conjI, clarify, rule iffI, simp, clarsimp)
 apply (frule-tac z=a in def-starD2,
    rule-tac x=a and z=x in top-derive-zero, assumption,
    (thin-tac ?x = H-top)+, simp add: H-ac, simp)
 apply (frule-tac z=b in def-starD2,
    rule-tac x=b and z=y in top-derive-zero, assumption,
    (thin-tac ?x = H-top)+, simp add: H-ac, simp)
apply (clarify, rule iffI, simp, clarsimp)
```

— Other properties of *H-def*
apply (*erule dperm-def-sub*, (*rule Rep-dperm*)+)
apply (*rule-tac x=x* **in** *Abs-dperm-cases*,
    clarsimp split: *prod.splits*
      simp add: *Abs-dperm-inverse Abs-dperm-inject dperm-def-def dperm-def*)
apply (*rule-tac x=x* **in** *Abs-dperm-cases*,
    clarsimp split: *prod.splits*
      simp add: *Abs-dperm-inverse Abs-dperm-inject dperm-def-def dperm-def*)
apply (*rule-tac x=x* **in** *Abs-dperm-cases*,
    clarsimp split: *prod.splits simp add*: *Abs-dperm-inverse Abs-dperm-inject*
      dperm-star-def dperm-def-def dperm-def)
apply (*rule-tac x=x* **in** *Abs-dperm-cases*, *rule-tac x=y* **in** *Abs-dperm-cases*,
    clarsimp split: *prod.splits simp add*: *Abs-dperm-inverse Abs-dperm-inject*
      dperm-star-def dperm-def-def dperm-def H-ac)
— Associativity
apply (*rule dperm-star-assoc*, (*rule Rep-dperm*)+)
— Cancellation
apply (*rule-tac x=x* **in** *Abs-dperm-cases*,
    rule-tac x=y **in** *Abs-dperm-cases*,
    rule-tac x=z **in** *Abs-dperm-cases*,
    clarsimp split: *prod.splits simp add*: *Abs-dperm-inverse Abs-dperm-inject*)
apply (*clarsimp split*: *prod.splits simp add*: *dperm-def-def dperm-def*)
apply (*simp split*: *split-if-asm add*: *dperm-star-def H-ac H-canc*)
apply (*rule conjI*)
apply (*rule-tac x=a* **in** *star-canc2*, *simp*, *assumption*, *assumption*)
apply (*rule-tac x=b* **in** *star-canc2*, *simp*, *assumption*, *assumption*)
— Non-zero
apply (*rule-tac x=x* **in** *Abs-dperm-cases*,
    rule-tac x=y **in** *Abs-dperm-cases*,
    clarsimp split: *prod.splits simp add*: *Abs-dperm-inverse Abs-dperm-inject*)
apply (*simp split*: *split-if-asm add*: *dperm-star-def H-ac H-canc*)
— undef star
apply (*clarsimp split*: *split-if-asm*
      simp add: *dperm-def-def dperm-star-def H-ac H-canc*)
done

end

Local and shared state pairs

These are used to define the semantics of RGSep assertions.

typedef *'a heap2* =
 {*x*::(*'a*::*res-algebra*) × *'a*.
    if fst x = H-zero then snd x = H-zero
    else if fst x = H-top then snd x = H-top
    else H-def (*fst x*) (*snd x*)}
by (*rule-tac x*=(*H-zero*, *H-zero*) **in** *exI*, *simp*)

definition *heap2-def* :: (*'a*::*res-algebra*) × *'a* ⇒ *'a* × *'a* ⇒ *bool* **where**

heap2-def x y ≡
  $x = (H\text{-}zero, H\text{-}zero)$
  $\vee$ $y = (H\text{-}zero, H\text{-}zero)$
  $\vee$ *H-def* (*fst x*) (*snd x*) $\wedge$ *H-def* (*fst x* ⊙ *snd x*) (*fst y*) $\wedge$ *snd x* = *snd y*

definition *heap2-star* :: ('*a*::*res-algebra*) × '*a* ⇒ '*a* × '*a* ⇒ '*a* × '*a* **where**
heap2-star x y ≡
  if x = (*H-zero*, *H-zero*) *then y*
  else if y = (*H-zero*, *H-zero*) *then x*
  else if heap2-def x y $\wedge$ *fst x* ⊙ *fst y* ≠ *H-top then* (*fst x* ⊙ *fst y*, *snd x*)
  else (*H-top*, *H-top*)

lemma *heap2-cont*:
(*H-zero*, *H-zero*) ∈ *heap2*
(*H-top*, *H-top*) ∈ *heap2*
⟦*x* ∈ *heap2*; *y* ∈ *heap2* ⟧ ⟹ *heap2-star x y* ∈ *heap2*
by (*auto split*: *prod.splits*
  simp add: *heap2-def heap2-star-def heap2-def-def H-ac H-canc*)

lemma *heap2-def-zero*:
heap2-def x (*H-zero*,*H-zero*)
by (*simp add*: *heap2-def-def*)

lemma *heap2-def-comm*:
heap2-def x y = *heap2-def y x*
by (*auto simp add*: *heap2-def-def H-ac*)

lemma *heap2-star-comm*:
heap2-star x y = *heap2-star y x*
apply (*simp add*: *heap2-star-def heap2-def-comm H-ac*)
apply (*rule conjI*)
apply (*clarsimp simp add*: *heap2-def-def*)+
done

lemma *heap2-star-zero*:
heap2-star x (*H-zero*, *H-zero*) = *x*
by (*simp add*: *heap2-star-def*)

lemmas *heap2-simps* = *heap2-cont*
  heap2-def-zero heap2-def-zero [*THEN heap2-def-comm* [*THEN iffD1*]]
  heap2-star-zero trans [*OF heap2-star-comm heap2-star-zero*]

lemma *heap2-def-sub*:
heap2-def (*heap2-star x y*) *z* ⟹ *heap2-def x z*
apply (*cases x*, *cases y*, *cases z*)
apply (*clarsimp split*: *split-if-asm*
    simp add: *heap2-star-def heap2-def-def H-ac H-canc*)
apply (*erule disjE*, *clarsimp*)+
apply (*clarsimp*, *simp only*: *star-assoc* [*THEN sym*])

apply (*erule notE*, *erule def-starE*)
done

lemma *heap2-def-assoc*:
⟦*heap2-def x y*; *heap2-def x z*; *heap2-def y z*; *x* ∈ *heap2*; *y* ∈ *heap2*; *z* ∈ *heap2*⟧
⟹ *heap2-def* (*heap2-star x y*) *z* = *heap2-def x* (*heap2-star y z*)
apply (*case-tac x* = (*H-zero*, *H-zero*), *simp add*: *heap2-simps*)
apply (*case-tac y* = (*H-zero*, *H-zero*), *simp add*: *heap2-simps*)
apply (*case-tac z* = (*H-zero*, *H-zero*), *simp add*: *heap2-simps*)
apply (*clarsimp simp add*: *heap2-star-def*)
apply (*cases x*, *rename-tac x1 x2*,
cases y, *rename-tac y1 y2*,
cases z, *rename-tac z1 z2*,
clarsimp split: *split-if-asm simp add*: *heap2-def heap2-def-def*)
apply (*clarsimp simp add*: *H-ac H-canc*)
apply (*frule-tac z=y1* **in** *def-starD2*, *simp add*: *H-ac*)
apply (*intro impI conjI iffI*, *simp-all add*: *H-ac*, *fast*)
apply (*subst star-assoc* [*THEN sym*], *simp*)
apply (*subst def-assoc* [*THEN sym*], *fast*, *assumption*, *assumption*)
apply (*clarsimp simp add*: *H-canc*)
done

lemma *heap2-star-assoc*:
⟦*x* ∈ *heap2*; *y* ∈ *heap2*; *z* ∈ *heap2*⟧ ⟹
heap2-star (*heap2-star x y*) *z* = *heap2-star x* (*heap2-star y z*)
apply (*case-tac x* = (*H-zero*, *H-zero*), *simp add*: *heap2-simps*)
apply (*case-tac y* = (*H-zero*, *H-zero*), *simp add*: *heap2-simps*)
apply (*case-tac z* = (*H-zero*, *H-zero*), *simp add*: *heap2-simps*)
apply (*cases x*, *rename-tac x1 x2*,
cases y, *rename-tac y1 y2*,
cases z, *rename-tac z1 z2*,
clarsimp simp add: *heap2-def*)
apply (*simp split*: *split-if-asm add*: *H-ac H-canc*)
apply (*simp-all add*: *heap2-star-def H-ac H-canc*)
apply (*clarsimp simp add*: *heap2-def-def H-ac H-canc*)
apply (*rule conjI*, *clarsimp simp add*: *H-ac*)
apply (*frule-tac z=z2* **in** *def-starD2*, *clarsimp simp add*: *H-ac*)
apply (*frule-tac z=z1* **in** *def-starD2*, *clarsimp simp add*: *H-ac*)
apply (*rule conjI*, *clarsimp simp add*: *H-ac*)
apply (*rule conjI*, *rule impI*, *rule def-starE2*, *simp add*: *H-ac*, *erule* (*1*) *def-starE2*)
apply (*erule contrapos-nn*, *simp*)
apply (*clarsimp simp add*: *star-assoc* [*THEN sym*])
apply (*clarsimp simp add*: *H-ac*)
apply (*rule conjI*, *clarsimp simp add*: *H-ac*)
apply (*frule def-starD1*, *clarsimp simp add*: *H-ac*)
apply (*subgoal-tac H-def x1* (*y1* ⊙ *z2*), *simp add*: *H-ac*)
apply (*erule notE*, *erule* (*1*) *def-starE2*)
apply (*rule-tac y=z1* **in** *def-starE1*, *simp add*: *H-ac*)
apply (*clarsimp simp add*: *H-ac*)

apply (*frule def-starD1*, *clarsimp simp add*: *H-ac*)
apply (*subgoal-tac H-def x1* (*y1* ⊙ *z2*), *simp add*: *star-assoc* [*THEN sym*])
apply (*rule-tac y=z1* **in** *def-starE1*, *simp add*: *H-ac*)
done

lemma *heap2-star-canc*:
  ⟦*heap2-star y x* = *heap2-star z x*; *heap2-def x y*; *heap2-def x z*;
    x ∈ *heap2*; *y* ∈ *heap2*; *z* ∈ *heap2*⟧ ⟹ *y* = *z*
apply (*simp add*: *heap2-star-comm*)
apply (*case-tac x* = (*H-zero*, *H-zero*), *simp add*: *heap2-simps*)
apply (*cases x*, *rename-tac x1 x2*,
      cases y, *rename-tac y1 y2*,
      cases z, *rename-tac z1 z2*,
      clarsimp split: *split-if-asm simp add*: *heap2-def*)
apply (*simp-all split*: *split-if-asm add*: *heap2-star-def heap2-def-def H-ac H-canc*)
   apply (*drule def-starD2*, *simp add*: *H-ac H-canc*)
  apply (*drule def-starD2*, *simp add*: *H-ac H-canc*)
 apply (*drule def-starD2*, *drule def-starD2*, *simp add*: *H-ac H-canc*)
apply (*rule-tac x=x1* **in** *star-canc*, *simp add*: *H-ac*, *fast*, *fast*)
done

instantiation *heap2* :: (*res-algebra*) *res-algebra*
begin

definition *H-zero* ≡ *Abs-heap2* (*H-zero*, *H-zero*)
definition *H-top* ≡ *Abs-heap2* (*H-top*, *H-top*)
definition *H-def x y* ≡ *heap2-def* (*Rep-heap2 x*) (*Rep-heap2 y*)
definition *x* ⊙ *y* ≡ *Abs-heap2* (*heap2-star* (*Rep-heap2 x*) (*Rep-heap2 y*))

definition *h1* ≤ (*h2*::(*'a*::*res-algebra*) *heap2*) ≡ (∃*z*. *H-def h1 z* ∧ *h1* ⊙ *z* = *h2*)
definition *h1* < (*h2*::(*'a*::*res-algebra*) *heap2*) ≡ (*h1* ≤ *h2* ∧ *h1* ≠ *h2*)

instance
apply (*intro-classes*)
apply (*simp add*: *less-heap2-def*)
apply (*simp add*: *less-eq-heap2-def*)
apply (*simp-all add*: *H-zero-heap2-def H-top-heap2-def H-def-heap2-def H-star-heap2-def*
      Abs-heap2-inject Abs-heap2-inverse Rep-heap2 heap2-simps)
apply (*rule heap2-def-comm*)
apply (*erule* (2) *heap2-def-assoc*, (*rule Rep-heap2*)+)
apply (*erule heap2-def-sub*)
apply (*rule-tac x=x* **in** *Abs-heap2-cases*,
      clarsimp simp add: *heap2-def-def H-ac H-canc*
        Abs-heap2-inverse Abs-heap2-inject heap2-cont H-ac H-canc,
      fast)
apply (*rule-tac x=x* **in** *Abs-heap2-cases*, *simp add*: *Abs-heap2-inject Abs-heap2-inverse*)
apply (*rule heap2-star-comm*)
apply (*rule heap2-star-assoc*, (*rule Rep-heap2*)+)
apply (*subst Rep-heap2-inject* [*THEN sym*],

```
    erule heap2-star-canc, (simp add: heap2-def-comm)+, (rule Rep-heap2)+)
apply (rule-tac x=x in Abs-heap2-cases,
    rule-tac x=y in Abs-heap2-cases,
    clarsimp split: split-if-asm
      simp add: Abs-heap2-inject Abs-heap2-inverse Rep-heap2
           heap2-cont heap2-star-def H-ac H-canc)
apply (rule-tac x=x in Abs-heap2-cases,
    rule-tac x=y in Abs-heap2-cases,
    clarsimp simp add: Abs-heap2-inject Abs-heap2-inverse Rep-heap2
                heap2-cont heap2-star-def H-ac H-canc,
    simp split: split-if-asm add: heap2-def heap2-def-def H-ac)
done

end
```

D.2.3 Atomic elements

```
definition H-atom :: 'a::res-algebra ⇒ bool where
  H-atom h ≡ (h ≠ H-zero ∧ (∀h'. h' < h ⟶ h' = H-zero))
```

An element of a resource algebra is atomic if and only if it cannot be divided into smaller non-zero elements.

```
lemma H-atom-singl:
  H-atom (H-singl x y)
apply (simp split: heap.splits cong: conj-cong
      add: H-atom-def H-singl-def le-res-def less-res-def
         H-star-heap-def2 H-def-heap-def map-add-def
         H-zero-heap-def H-zero-fun-def H-zero-option-def)
apply (rule conjI, rule notI, drule-tac x=x in fun-cong, simp)
apply (clarify, rename-tac a b, case-tac a, case-tac b, clarsimp)
apply (rule ext, erule contrapos-np, rule ext, rename-tac m1 m2)
apply (frule-tac x=m1 in fun-cong)
apply (drule-tac x=m2 in fun-cong)
apply (simp split: split-if-asm option.splits add: H-def-fun-def H-def-option-def)
apply (erule-tac x=x in allE, erule disjE, simp, simp)
done

end
```

D.3 Deny-guarantee program logic

```
theory DGLogic
imports Main Heaps VHelper
begin
```

This section defines deny-guarantee permissions, assertions, the local and the global operational semantics from the paper. It also contains the proofs that the deny-guarantee proof rules are sound with respect to the operational semantics.

D.3.1 Syntax

First, we define the following useful operations on partial functions:

definition *fdisj* :: $('a \rightharpoonup 'b) \Rightarrow ('a \rightharpoonup 'b) \Rightarrow bool$ **where**
$fdisj\ f\ g \equiv (\forall x.\ f\ x = None \lor g\ x = None)$

definition *fsingl* :: $'a \Rightarrow 'b \Rightarrow ('a \rightharpoonup 'b)$ **where**
$fsingl\ x\ y \equiv (\lambda z.\ if\ z = x\ then\ Some\ y\ else\ None)$

definition *fminus* :: $('a \rightharpoonup 'b) \Rightarrow 'a \Rightarrow ('a \rightharpoonup 'b)$ **where**
$fminus\ f\ x \equiv (\lambda y.\ if\ y = x\ then\ None\ else\ f\ y)$

lemma *fdisj-def2*:
$fdisj\ a\ b = (dom\ a \cap dom\ b = \{\})$
by (*simp add*: *fdisj-def dom-def Int-def*)

lemma *fdisj-break*:
$fdisj\ a\ (b \mathbin{++} c) = (fdisj\ a\ b \land fdisj\ a\ c)$
$fdisj\ (a \mathbin{++} b)\ c = (fdisj\ b\ c \land fdisj\ a\ c)$
by (*auto simp add*: *fdisj-def2 Int-def*)

lemma *fdisj-minus*:
$fdisj\ a\ b \Longrightarrow fdisj\ (fminus\ a\ x)\ b$
by (*simp add*: *fdisj-def fminus-def*)

lemma *dom-None*:
$[\![A\ x = None;\ dom\ B \subseteq dom\ A]\!] \Longrightarrow B\ x = None$
by *auto*

lemma *dom-Some*:
$[\![A\ x = Some\ y;\ dom\ A \subseteq dom\ B]\!] \Longrightarrow \exists z.\ B\ x = Some\ z$
by *auto*

lemma *dom-fminus*:
$dom\ (fminus\ x\ y) = dom\ x - \{y\}$
by (*auto simp add*: *dom-def fminus-def*)

Deny-guarantee permissions

We do a shallow embedding of deny-guarantee permissions.

types
$'a\ fracDG \quad = (bool,\ 'a)\ permTag$
$('a,'b)\ permDG = 'a \times 'a \Rightarrow 'b\ fracDG$

definition *is-deny* :: $('a::res\text{-}algebra)\ fracDG \Rightarrow bool$ **where**
$is\text{-}deny\ x \equiv (x \neq H\text{-}zero \land x \neq H\text{-}top \land fst\ (Rep\text{-}permTag\ x))$

definition *is-guar* :: $('a::res\text{-}algebra)\ fracDG \Rightarrow bool$ **where**
$is\text{-}guar\ x \equiv (x \neq H\text{-}zero \land x \neq H\text{-}top \land \neg fst\ (Rep\text{-}permTag\ x))$

lemma *is-deny-mon*:
is-deny x $\Longrightarrow$ *x* $\odot$ *y* = *H-top* $\vee$ *is-deny* (*x* $\odot$ *y*)
apply (*case-tac* $\neg$ *H-def x y*, *drule undef-star*, *simp*)
apply (*rule-tac x=x* **in** *Abs-permTag-cases*,
rule-tac x=y **in** *Abs-permTag-cases*)
apply (*clarsimp simp add*: *is-deny-def*
H-def-permTag-def H-star-permTag-def H-top-permTag-def H-zero-permTag-def
Abs-permTag-inverse Abs-permTag-inject permTag-star1 permTag-simps)
apply (*simp split*: *split-if-asm add*: *permTag-star-def permTag-def permTag-def-def H-ac H-canc*)
apply (*case-tac default-value*::*bool*, *simp*, *simp*)+
done

lemma *is-guar-mon*:
is-guar x $\Longrightarrow$ *x* $\odot$ *y* = *H-top* $\vee$ *is-guar* (*x* $\odot$ *y*)
apply (*case-tac* $\neg$ *H-def x y*, *drule undef-star*, *simp*)
apply (*rule-tac x=x* **in** *Abs-permTag-cases*,
rule-tac x=y **in** *Abs-permTag-cases*)
apply (*clarsimp simp add*: *is-guar-def*
H-def-permTag-def H-star-permTag-def H-top-permTag-def H-zero-permTag-def
Abs-permTag-inverse Abs-permTag-inject permTag-star1 permTag-simps)
apply (*simp split*: *split-if-asm add*: *permTag-star-def permTag-def permTag-def-def H-ac H-canc*)
apply (*case-tac default-value*::*bool*, *simp*, *simp*)+
done

lemma *is-guar-antimon*:
is-guar (*x* $\odot$ *y*) $\Longrightarrow$ *x* = *H-zero* $\vee$ *is-guar x*
apply (*case-tac* $\neg$ *H-def x y*, *drule undef-star*, *simp add*: *is-guar-def*)
apply (*rule-tac x=x* **in** *Abs-permTag-cases*,
rule-tac x=y **in** *Abs-permTag-cases*)
apply (*clarsimp simp add*: *is-guar-def*
H-def-permTag-def H-star-permTag-def H-top-permTag-def H-zero-permTag-def
Abs-permTag-inverse Abs-permTag-inject permTag-star1 permTag-simps)
apply (*simp split*: *split-if-asm add*: *permTag-star-def permTag-def permTag-def-def H-ac H-canc*)
done

lemma *is-guar-expand*:
is-guar x $\longleftrightarrow$ (*x* $\neq$ *H-zero* $\wedge$ *x* $\neq$ *H-top* $\wedge$ $\neg$ *is-deny x*)
by (*simp add*: *is-deny-def is-guar-def*) *fast*

lemma *fracDG-cases*:
$[\![$ *x* = *H-zero* $\Longrightarrow$ *P*; *x* = *H-top* $\Longrightarrow$ *P*;
is-deny x $\Longrightarrow$ *P*; *is-guar x* $\Longrightarrow$ *P* $]\!]$ $\Longrightarrow$ *P*
by (*case-tac is-guar x*, *simp*, *simp add*: *is-guar-expand*, *metis*)

lemma *is-guar-star3*:
$[\![$ *is-guar x*; *H-def x y* $]\!]$ $\Longrightarrow$ *y* = *H-zero* $\vee$ *is-guar y*
apply (*rule-tac x=y* **in** *fracDG-cases*, *simp-all add*: *is-guar-expand*)
apply (*simp add*: *H-def-permTag-def permTag-def-def*)

apply (*rule-tac x=x* **in** *Abs-permTag-cases*,
rule-tac x=y **in** *Abs-permTag-cases*)
apply (*clarsimp simp add*: *is-deny-def*
H-def-permTag-def H-star-permTag-def H-top-permTag-def H-zero-permTag-def
Abs-permTag-inverse Abs-permTag-inject permTag-star1 permTag-simps)
apply (*case-tac default-value*::*bool*, *simp*, *simp*)
done

definition *allowed* :: ($'a$,$'b$::*res-algebra*) *permDG* ⇒ ($'a \times 'a$) *set* **where**
allowed k x ≡ (*case x of* (s,s') ⇒ s=s' ∨ k (s,s') = *H-top* ∨ *is-guar* (k (s,s')))

The predicate *allowed k x* holds if and only if the permission k allows the program to do the transition x.

lemma *allowed-refl*:
allowed k (s,s)
by (*simp add*: *allowed-def*)

lemma *allowed-mon*[*elim*]:
⟦ *allowed k x*; *H-def k k'* ⟧ ⟹ *allowed* ($k \odot k'$) x
by (*simp split*: *prod.splits add*: *allowed-def H-star-fun-def*)
((*erule disjE*, *simp*)+, *drule is-guar-mon*, *fast*)

definition *interfered* :: ($'a$,$'b$::*res-algebra*) *permDG* ⇒ ($'a \times 'a$) *set* **where**
interfered k x ≡ (*case x of* (s,s') ⇒ s = s' ∨ k (s,s') = *H-zero* ∨ *is-guar* (k (s,s')))

The predicate *interfered k x* holds if and only if the permission k allows the environment to interfere by doing the transition x.

lemma *interfered-refl*:
interfered k (s,s)
by (*simp add*: *interfered-def*)

lemma *interfered-antimon*[*elim*]:
⟦ *interfered* ($k \odot k'$) x; *H-def k k'* ⟧ ⟹ *interfered k x*
by (*simp split*: *prod.splits add*: *interfered-def H-star-fun-def H-ac H-canc*)
((*erule disjE*, *simp*)+, *drule is-guar-antimon*, *simp*)

lemma *allowed-interfered*:
⟦ *H-def k k'*; *allowed k x* ⟧ ⟹ *interfered k' x*
apply (*simp split*: *prod.splits*
add: *allowed-def interfered-def H-star-fun-def H-def-fun-def H-ac H-canc*)
apply (*rename-tac m n*, *drule-tac a=m* **and** *b=n* **in** *all2D*)
apply ((*erule disjE*, *simp*)+, *drule* (*1*) *is-guar-star3*, *fast*)
done

Deny-guarantee assertions

Below is a deep embedding of deny-guarantee assertions. The deep embedding seems unavoidable because of the recursion in the domain.

datatype ($'a$,$'b$) *lt-assn* =

a-True
| *a-Bool 'a set*
| *a-Perm ('a,'b) permDG set*
| *a-Thread 'a ⇒ nat ('a,'b) lt-assn*
| *a-Star ('a,'b) lt-assn ('a,'b) lt-assn* (**infixr** ∗∗ *160*)

types *('a,'b) lt-queue = nat ⇀ ('a,'b) lt-assn*

We proceed to define the semantics of DG assertions.

primrec
lt-sat :: *('a,'b::res-algebra) lt-assn ⇒*
('a × ('a,'b) permDG × ('a,'b) lt-queue) set
where
lt-sat (a-True) x = True
| *lt-sat (a-Bool B) x = (case x of (s,k,tq) ⇒ B s ∧ k=H-zero ∧ tq=empty)*
| *lt-sat (a-Perm K) x = (case x of (s,k,tq) ⇒ K k ∧ tq=empty)*
| *lt-sat (a-Thread f A) x = (case x of (s,k,tq) ⇒ k=H-zero ∧ tq = fsingl (f s) A)*
| *lt-sat (a-Star P Q) x = (case x of (s,k,tq) ⇒ ∃k1 k2 tq1 tq2.*
lt-sat P (s,k1,tq1) ∧ lt-sat Q (s,k2,tq2)
∧ H-def k1 k2 ∧ k = k1 ⊙ k2
∧ fdisj tq1 tq2 ∧ tq = tq1 ++ tq2)

An assertion describes a three-element tuple consisting of a state, a permission and a thread queue.

lemma *Star-cong[cong]*:
⟦*lt-sat P = lt-sat P'*; *lt-sat Q = lt-sat Q'*⟧ ⟹
lt-sat (P ∗∗ Q) = lt-sat (P' ∗∗ Q')
by (*rule ext*, *clarsimp*)

lemma *Star-assoc*:
lt-sat ((P ∗∗ Q) ∗∗ R) = lt-sat (P ∗∗ (Q ∗∗ R))
apply (*rule ext*, *clarsimp simp add*: *H-ac fdisj-break*)
apply (*safe*, *simp-all add*: *fdisj-break H-ac*)
apply (*frule-tac* [*1−2*] *def-starD2*)
apply (*assumption* | *rule exI conjI* | *simp add*: *fdisj-break H-ac*)+
done

lemma *Star-comm*:
lt-sat (Q ∗∗ P) = lt-sat (P ∗∗ Q)
apply (*rule ext*, *clarsimp*, *safe*, *simp-all add*: *H-ac*)
apply (*assumption* | *rule exI conjI* |
simp add: *fdisj-def2 map-add-comm Int-commute H-ac*)+
done

lemma *Star-left-comm*:
lt-sat (P ∗∗ Q ∗∗ R) = lt-sat (Q ∗∗ P ∗∗ R)
apply (*subst* (2) *Star-assoc* [*THEN sym*])
apply (*simp add*: *Star-comm* [*of Q P*])
apply (*simp only*: *Star-assoc*)

done

lemmas *Star-ac* = *Star-assoc Star-comm Star-left-comm*

lemma *Star-mon*:
  $[\![$*lt-sat P* ⊆ *lt-sat P′*; *lt-sat Q* ⊆ *lt-sat Q′*$]\!]$ ⟹
   lt-sat (*P* ∗∗ *Q*) ⊆ *lt-sat* (*P′* ∗∗ *Q′*)
apply (*clarsimp simp add*: *le-fun-def le-bool-def*)
apply ((*erule allE*)+, (*erule* (*1*) *impE*)+)
apply *fast*
done

definition
  lt-sat-eq :: (*′a*,*′b*::*res-algebra*) *lt-assn* ⇒ (*′a*,*′b*) *lt-assn* ⇒ *bool*
  (**infix** ≃ *100*)
where
  P ≃ *Q* ⟷ *lt-sat P* = *lt-sat Q*

instantiation *lt-assn* :: (*type*, *res-algebra*) *ord*
begin
  definition *x* ≤ *y* ≡ *lt-sat x* ⊆ *lt-sat y*
  definition *x* < *y* ≡ *lt-sat x* ⊂ *lt-sat y*
  instance ..
end

Now we prove some trivial transitivity lemmas, which assist in writing Isar transitivity proofs.

lemma *lt-assn-trans*[*trans*]:
  (*x*::(*′a*,*′b*::*res-algebra*) *lt-assn*) ≃ *y* ⟹ *y* ≃ *z* ⟹ *x* ≤ *z*
  (*x*::(*′a*,*′b*::*res-algebra*) *lt-assn*) ≃ *y* ⟹ *y* ≤ *z* ⟹ *x* ≤ *z*
  (*x*::(*′a*,*′b*::*res-algebra*) *lt-assn*) ≤ *y* ⟹ *y* ≃ *z* ⟹ *x* ≤ *z*
  (*x*::(*′a*,*′b*::*res-algebra*) *lt-assn*) ≤ *y* ⟹ *y* ≤ *z* ⟹ *x* ≤ *z*
by (*simp-all add*: *less-eq-lt-assn-def lt-sat-eq-def*)

Precise assertions:

definition *precise* :: (*′a*,*′b*::*res-algebra*) *lt-assn* ⇒ *bool* **where**
  precise P ≡ (∀ *s k tq k1 tq1 k1′ tq1′ k2 tq2 k2′ tq2′*.
    lt-sat P (*s*,*k1*,*tq1*) ∧ *lt-sat P* (*s*,*k2*,*tq2*)
    ∧ *H-def k1 k1′* ∧ *k* = *k1* ⊙ *k1′* ∧ *fdisj tq1 tq1′* ∧ *tq* = *tq1* ++ *tq1′*
    ∧ *H-def k2 k2′* ∧ *k* = *k2* ⊙ *k2′* ∧ *fdisj tq2 tq2′* ∧ *tq* = *tq2* ++ *tq2′*
    ⟶ *k1* = *k2* ∧ *tq1* = *tq2*)

typedef (*′a*,*′b*) *prec-assn* = {*P*::(*′a*,*′b*::*res-algebra*) *lt-assn*. *precise P*}
by (*rule-tac x*=*a-Bool bot* **in** *exI*, *simp add*: *precise-def*)

lemma *precD*:
   $[\![$*lt-sat* (*Rep-prec-assn p*) (*s*, *k1*, *tq1*); *lt-sat* (*Rep-prec-assn p*) (*s*, *k2*, *tq2*);
    H-def k1 k1′; *fdisj tq1 tq1′*; *H-def k2 k2′*; *fdisj tq2 tq2′*;
    k1 ⊙ *k1′* = *k2* ⊙ *k2′*; *tq1* ++ *tq1′* = *tq2* ++ *tq2′*$]\!]$ ⟹
   k1 = *k2* ∧ *k1′* = *k2′* ∧ *tq1* = *tq2* ∧ *tq1′* = *tq2′*

apply (*rule-tac x=p* **in** *Abs-prec-assn-cases*,
clarsimp simp add: *Abs-prec-assn-inverse prec-assn-def*)
apply (*simp* (*no-asm-use*) *add*: *precise-def*)
apply ((*erule allE*)+, *erule impE*, (*erule conjI*)+, (*rule exI*, *erule conjI*, *simp*)+)
apply (*clarsimp simp add*: *H-ac H-canc*)
apply (*simp split*: *option.splits add*: *expand-fun-eq map-add-def fdisj-def*)
apply (*rule allI*, *rename-tac m*, (*erule-tac x=m* **in** *allE*)+)
apply (*case-tac tq1′ m*, *simp-all*)
apply (*case-tac tq2′ m*, *simp-all*)+
done

definition *stable* :: (*′a*,*′b*::*res-algebra*) *lt-assn* ⇒ *bool* **where**
stable p ≡ (∀ *s s′ k tq*. *lt-sat p* (*s*,*k*,*tq*) ∧ *interfered k* (*s*,*s′*) ⟶ *lt-sat p* (*s′*,*k*,*tq*))

An assertion p is stable if and only if it is preserved under any interference it allows.

lemma *stableE*:
⟦*stable p*; *lt-sat p* (*s*,*k*,*tq*); *interfered k* (*s*,*s′*)⟧ ⟹ *lt-sat p* (*s′*,*k*,*tq*)
by (*unfold stable-def*) (*fast*)

lemma *stableE2*:
⟦*lt-sat p* (*s*,*k*,*tq*); *stable p*; *interfered k* (*s*,*s′*)⟧ ⟹ *lt-sat p* (*s′*,*k*,*tq*)
by (*unfold stable-def*) (*fast*)

lemma *relpow-simps2*:
R ^ *Suc n* = *R* ^ *n O R*
by (*induct n*, *simp*, *simp add*: *O-assoc* [*THEN sym*])

declare *relpow.simps*(*2*) [*simp del*]

lemma *vv-simps*[*simp*]:
Id (*s*,*s′*) = (*s′*=*s*)
(*R′ O R*) (*s*,*s′*) = (∃*t*. *R* (*s*,*t*) ∧ *R′* (*t*,*s′*))
by (*simp-all add*: *Id-def Collect-def rel-comp-def mem-def*)

lemma *stableEc*:
⟦*lt-sat p* (*s*,*k*,*tq*); *stable p*; (*interfered k* ^ *n*) (*s*,*s′*)⟧ ⟹ *lt-sat p* (*s′*,*k*,*tq*)
apply (*induct n arbitrary*: *s*, *simp*)
apply (*clarsimp simp add*: *relpow-simps2*, *drule* (*2*) *stableE2*, *simp*)
done

lemma *stable-star*:
⟦*stable P*; *stable Q*⟧ ⟹ *stable* (*P* ∗∗ *Q*)
apply (*simp* (*no-asm-use*) *add*: *stable-def*, *clarify*)
apply (*frule* (*1*) *interfered-antimon*)
apply ((*erule allE*)+, *erule impE*, *erule* (*1*) *conjI*, *erule impE*, *erule conjI*)
apply (*rule interfered-antimon*, *subst star-comm*, *assumption*, *simp add*: *H-ac*)
apply *fast*
done

types
var = *nat*
state = *var* ⇒ *nat*

definition
nsupp :: *var* ⇒ (*state*,'*a*::*res-algebra*) *lt-assn* ⇒ *bool*
where
nsupp v P ≡ (∀ *s k tq n*. *lt-sat P* (*s*,*k*,*tq*) ⟷ *lt-sat P* (*s*(*v*:=*n*),*k*,*tq*))

The predicate *nsupp v P* holds if and only if the variable *v* is not in the support of *P*. The support of an assertion *P* is the semantic counterpart of the set of its free variables.

lemma *nsuppD2*:
⟦*lt-sat P* (*s*,*k*,*tq*); *nsupp v P* ⟧⟹ *lt-sat P* (*s*(*v*:=*n*),*k*,*tq*)
by (*simp add*: *nsupp-def*)

Commands

datatype '*a cmd* =
c-Skip
| *c-Seq* ('*a*::*res-algebra*) *cmd* '*a cmd* (**infixl** ;; *100*)
| *c-Choice* '*a cmd* '*a cmd*
| *c-Loop* '*a cmd*
| *c-Atom* *state* ⇒ *state* ⇒ *bool*
| *c-Fork* *var* (*state*,'*a*) *prec-assn* (*state*,'*a*) *lt-assn* '*a cmd*
| *c-Join* *state* ⇒ *nat*

Commands include the empty command, sequential composition, non-deterministic choice, looping, atomic state updates, fork and join. Fork commands are annotated with the precondition and the postcondition of the new thread.

primrec
lt-Prim :: ('*a*::*res-algebra*) *cmd* ⇒ *bool*
where
lt-Prim c-Skip = *False*
| *lt-Prim* (*C*;; *D*) = *False*
| *lt-Prim* (*c-Choice C D*) = *False*
| *lt-Prim* (*c-Loop C*) = *False*
| *lt-Prim* (*c-Atom A*) = *True*
| *lt-Prim* (*c-Fork v P Q C*) = *True*
| *lt-Prim* (*c-Join n*) = *True*

The predicate *lt-Prim C* is true if and only if *C* is a primitive command (evaluates in a single step).

D.3.2 The local operational semantics

We say that a local configuration is safe if and only if it does not abort in a single step. This corresponds to the absence of a direct transition to **abort** in the paper's local operational semantics.

primrec

lt-safe :: ('a::res-algebra) cmd ⇒ state ⇒ (state,'a) permDG ⇒
(state,'a) lt-queue ⇒ bool
where
lt-safe c-Skip s k tq = True
| lt-safe (C1;; C2) s k tq = lt-safe C1 s k tq
| lt-safe (c-Choice C1 C2) s k tq = True
| lt-safe (c-Loop C) s k tq = True
| lt-safe (c-Atom A) s k tq = (∀ s'. A s s' ⟶ allowed k (s,s'))
| lt-safe (c-Fork v P Q C) s k tq =
(∃ k0 k1 tq0 tq1. lt-sat (Rep-prec-assn P) (s,k0,tq0)
∧ H-def k0 k1 ∧ k0 ⊙ k1 = k
∧ fdisj tq0 tq1 ∧ tq0 ++ tq1 = tq
∧ (∀ n. allowed k1 (s, s(v:=n))))
| lt-safe (c-Join E) s k tq = (∃ q. tq (E s) = Some q)

The local operational semantics uses the following three kinds of labels.

datatype 'a lt-lab =
lab-None
| lab-Fork nat ('a::res-algebra) cmd (state,'a) permDG
(state,'a) lt-queue (state,'a) lt-assn
| lab-Join nat (state,'a) permDG (state,'a) lt-queue

And finally, here is the main transition relation:

inductive
lt-sem :: ('a::res-algebra) cmd ⇒ state ⇒ (state,'a) permDG ⇒ (state,'a) lt-queue
⇒ 'a lt-lab ⇒ 'a cmd ⇒ state ⇒ (state,'a) permDG ⇒ (state,'a) lt-queue
⇒ bool
where
skip-seqI[intro!]: lt-sem (c-Skip;; C) s k tq lab-None C s k tq
| seqI[intro!]: lt-sem C s k tq lab C1 s' k' tq'
⟹ lt-sem (C;; C') s k tq lab (C1;; C') s' k' tq'
| choice-lI[intro!]: lt-sem (c-Choice C1 C2) s k tq lab-None C1 s k tq
| choice-rI[intro!]: lt-sem (c-Choice C1 C2) s k tq lab-None C2 s k tq
| loopI[intro!]:
lt-sem (c-Loop C) s k tq lab-None (c-Choice c-Skip (c-Seq C (c-Loop C))) s k tq
| atomI[intro!]: ⟦A s s'; allowed k (s,s') ⟧
⟹ lt-sem (c-Atom A) s k tq lab-None c-Skip s' k tq
| forkI[intro!]:
⟦s' = s(v:=n); lt-sat (Rep-prec-assn P) (s,k0,tq0);
k = k0 ⊙ k1; H-def k0 k1; tq = tq0 ++ tq1; fdisj tq0 tq1;
allowed k1 (s,s'); tq n = None; tq2 = tq1 ++ fsingl n Q ⟧ ⟹
lt-sem (c-Fork v P Q C) s k tq (lab-Fork n C k0 tq0 Q) c-Skip s' k1 tq2
| joinI[intro!]:
⟦E s = n; tq n = Some q; lt-sat q (s,k0,tq0);
H-def k k0; k' = k ⊙ k0;
fdisj tq tq0; tq' = fminus (tq ++ tq0) n ⟧ ⟹
lt-sem (c-Join E) s k tq (lab-Join n k0 tq0) c-Skip s k' tq'

The following two definitions are the closure of the safety function and of the transition relation under reflexivity, transitivity, and environment actions.

primrec
lt-safeS :: *nat* ⇒ (*'a*::*res-algebra*) *cmd* ⇒ *state* ⇒
(*state*,*'a*) *permDG* ⇒ (*state*,*'a*) *lt-queue* ⇒ *bool*
where
lt-safeS 0 = (*λC s k tq*. *True*)
| *lt-safeS* (*Suc n*) = (*λC s k tq*. *lt-safe C s k tq*
∧ (∀ *lab C' s' k' tq'*. *lt-sem C s k tq lab C' s' k' tq'* ⟶ *lt-safeS n C' s' k' tq'*)
∧ (∀ *s'*. *interfered k* (*s*,*s'*) ⟶ *lt-safeS n C s' k tq*))

primrec
lt-semS :: *nat* ⇒(*'a*::*res-algebra*) *cmd* ⇒ *state* ⇒ (*state*,*'a*) *permDG* ⇒ (*state*,*'a*) *lt-queue* ⇒
'a cmd ⇒ *state* ⇒ (*state*,*'a*) *permDG* ⇒ (*state*,*'a*) *lt-queue* ⇒ *bool*
where
lt-semS 0 *C s k tq C' s' k' tq'* = (*C*=*C'* ∧ *s*=*s'* ∧ *k*=*k'* ∧ *tq*=*tq'*)
| *lt-semS* (*Suc n*) *C s k tq C' s' k' tq'* =
(∃ *C'' s'' k'' tq''*. *lt-semS n C'' s'' k'' tq'' C' s' k' tq'*
∧ (∃ *lab*. *lt-sem C s k tq lab C'' s'' k'' tq''*
∨ *interfered k* (*s*,*s''*) ∧ *k''*=*k* ∧ *C''*=*C* ∧ *tq''*=*tq*))

Here is an alternative definition for *lt-safeS* in terms of *lt-semS*.

lemma *lt-safeS-def2*:
lt-safeS n C s k tq =
(∀ *m* < *n*. ∀ *C' s' k' tq'*. *lt-semS m C s k tq C' s' k' tq'*
⟶ *lt-safe C' s' k' tq'*)
apply (*induct n arbitrary*: *C s k tq*, *simp*, *clarsimp*)
apply (*rule iffI*, *clarsimp*)
apply (*case-tac m*, *simp*, *clarsimp*, *fast*)
apply (*intro conjI*, *erule-tac x=0* **in** *allE*, *simp*)
apply (*clarify*, *erule-tac x=Suc m* **in** *allE*, *simp*, *fast*)+
done

Now, we define the meaning of the deny-guarantee Hoare triples.

definition
lt-has-spec :: (*state*,*'a*::*res-algebra*) *lt-assn* ⇒ *'a cmd* ⇒ (*state*,*'a*) *lt-assn* ⇒ *bool*
where
lt-has-spec P C Q ≡ ∀ *s pr tq*. *lt-sat P* (*s*,*pr*,*tq*) ⟶ (
(∀ *n*. *lt-safeS n C s pr tq*)
∧ (∀ *n s' pr' tq'*. *lt-semS n C s pr tq c-Skip s' pr' tq'* ⟶ *lt-sat Q* (*s'*,*pr'*,*tq'*)))

primrec
lt-wa :: (*'a*::*res-algebra*) *cmd* ⇒ *bool*
where
lt-wa c-Skip = *True*
| *lt-wa* (*C* ;; *D*) = (*lt-wa C* ∧ *lt-wa D*)
| *lt-wa* (*c-Choice C D*) = (*lt-wa C* ∧ *lt-wa D*)
| *lt-wa* (*c-Loop C*) = (*lt-wa C*)
| *lt-wa* (*c-Atom A*) = *True*
| *lt-wa* (*c-Fork v P Q C*) = (*lt-has-spec* (*Rep-prec-assn P*) *C Q* ∧ *lt-wa C*)
| *lt-wa* (*c-Join n*) = *True*

A command is well annotated (*lt-wa C*) if and only if all its forked threads satisfy their annotated specifications.

definition
lt-Hoare :: (*state*,′*a*::*res-algebra*) *lt-assn* ⇒ ′*a cmd* ⇒ (*state*,′*a*) *lt-assn* ⇒ *bool*
where
lt-Hoare P C Q ≡ *lt-wa C* ∧ *lt-has-spec P C Q*

The Hoare triple *lt-Hoare P C Q* holds if and only if *C* is well annotated and satisfies it specification (*lt-has-spec P C Q*).

First, we prove some simple properties about the individual transitions.

lemma *lt-sem-Skip*:
¬*lt-sem c-Skip s pr tq lab C′ s′ pr′ tq′*
by (*rule notI*, *erule lt-sem.cases*, *simp-all*)

lemma *lt-semS-Skip*:
lt-semS n c-Skip s k tq C′ s′ k′ tq′
⟷ (*C′*=*c-Skip* ∧ *k′*=*k* ∧ *tq′*=*tq* ∧ (*interfered k* ^ *n*) (*s*,*s′*))
by (*induct n arbitrary*: *s*)
(*auto simp add*: *lt-sem-Skip relpow-simps2*)

lemma *lt-semS-Prim*:
lt-Prim C ⟹
lt-semS n C s k tq C′ s′ k′ tq′
⟷ (*C′*=*C* ∧ *k′*=*k* ∧ *tq′*=*tq* ∧ (*interfered k* ^ *n*) (*s*,*s′*)
∨ *C′*=*c-Skip* ∧
(∃*n1 n2 s1 s2 lab*. *n* = *Suc* (*n1*+*n2*)
∧ (*interfered k* ^ *n1*) (*s*,*s1*)
∧ (*interfered k′* ^ *n2*) (*s2*,*s′*)
∧ *lt-sem C s1 k tq lab c-Skip s2 k′ tq′*))
apply (*induct n arbitrary*: *s*, *simp add*: *eq-ac conj-ac*, *simp*)
apply (*rule iffI*, *elim exE conjE*)
apply (*erule disjE*, *erule exE*, *rule disjI2*)
apply (*erule lt-sem.cases*, *simp-all*)
apply (*clarsimp simp add*: *lt-semS-Skip*,
rule-tac x=0 **in** *exI*, *rule-tac x=n* **in** *exI*, *simp*, *fast*)+
apply (*erule disjE*, *clarsimp simp add*: *relpow-simps2*, *fast*)
apply (*rule disjI2*, *clarsimp*)
apply (*rule-tac x=Suc n1* **in** *exI*, *rule-tac x=n2* **in** *exI*,
simp add: *relpow-simps2*, *fast*)
apply (*erule disjE*, *clarsimp simp add*: *relpow-simps2*, *rename-tac t*)
apply (*erule-tac x=t* **in** *meta-allE*, *simp*, *fast*)
apply (*clarsimp*, *case-tac n1*)
apply (*rule-tac x=c-Skip* **in** *exI*, *simp add*: *lt-semS-Skip*, *fast*)
apply (*clarsimp simp only*: *relpow-simps2 vv-simps*)
apply (*rename-tac m t*, *erule meta-allE*, *drule iffD2*, *rule disjI2*, *simp*)
apply (*rule-tac x=m* **in** *exI*, *rule-tac x=n2* **in** *exI*, *simp*, *case-tac C*, *simp-all*)
apply *fast+*
done

lemma *lt-sem-Guar*:
lt-sem C s k tq lab C′ s′ k′ tq′ ⟹ *allowed k* (*s*,*s′*)
by (*erule lt-sem.induct*, *simp-all add*: *allowed-refl*)
(*drule-tac k′=k0* **in** *allowed-mon*, *simp-all add*: *H-ac fun-upd-def*)

Soundness of the proof rules

We proceed by proving each rule in order. We start with the frame rule, whose proof requires the following monotonicity lemmas.

lemma *lt-safe-mon*: ⟦ *lt-safe C s k tq*; *H-def k k′*; *fdisj tq tq′* ⟧ ⟹
lt-safe C s (*k* ⊙ *k′*) (*tq* ++ *tq′*)
apply (*subgoal-tac lt-safe C s k tq* ⟶ *?Q*, *erule* (*1*) *mp*)
apply (*induct C*, *simp-all*)
apply *fast*
apply (*clarsimp*, *rename-tac k0 k1 tq0 tq1*)
apply (*rule exI*, *rule-tac x=k1* ⊙ *k′* **in** *exI*, *rule exI*, *erule conjI*)
apply (*simp add*: *H-ac H-canc*, *rule conjI*, *fast*)
apply (*rule-tac x=tq1++tq′* **in** *exI*, *clarsimp simp add*: *fdisj-def*)
apply (*rule conjI*, *metis*)
apply (*rule allI*, *erule allE*, *erule allowed-mon*, *fast*)
apply *clarsimp*
apply (*simp split*: *option.splits add*: *map-add-def*)
done

lemma *lt-sem-mon* [*rule-format*]:
lt-sem C s k0 tq0 lab C′ s′ k2 tq2 ⟹
H-def k k′ ⟶ *fdisj tq tq′* ⟶
k0 = *k* ⊙ *k′* ⟶ *tq0* = *tq* ++ *tq′* ⟶
lt-safe C s k tq ⟶
(∃*k1 tq1*. *lt-sem C s k tq lab C′ s′ k1 tq1*
∧ *H-def k1 k′* ∧ *k1* ⊙ *k′* = *k2*
∧ *fdisj tq1 tq′* ∧ *tq1* ++ *tq′* = *tq2*)
apply (*erule lt-sem.induct*)
apply *fast*
apply (*clarsimp*, *rule exI*, *rule exI*, *rule conjI*, *fast*, *simp*)
apply (*fast*, *fast*, *fast*)
apply (*clarsimp*, *rule exI*, *rule exI*, *rule conjI*, *fast*, *simp*)
— Fork
apply *clarsimp*
apply (*frule def-starD1*)
apply (*clarsimp simp add*: *H-ac H-canc fun-upd-def* [*THEN symmetric*])
apply (*drule-tac tq2′=tq1a* ++ *tq′* **in** *precD*, *assumption+*, *simp-all*)
apply (*simp add*: *fdisj-def*, *metis*)
apply *clarsimp*
apply (*rule exI*, *rule exI*, *rule conjI*, *fast*, *simp*)
apply (*simp split*: *option.splits add*: *map-add-def expand-fun-eq fdisj-def fsingl-def*)
— Join
apply *clarsimp*

apply (*rule-tac x=k0* ⊙ *k* **in** *exI*)
apply (*rule-tac x=fminus* (*tq* ++ *tq0*) (*E s*) **in** *exI*, *simp add*: *H-ac*)
apply (*frule def-starD2*, *clarsimp simp add*: *H-ac H-canc*)
apply (*erule disjE*, *simp add*: *fdisj-def* [*of tq*], *erule-tac x=E s* **in** *allE*, *simp*)
apply (*rule conjI*, *rule joinI*, *simp*, *assumption*+, (*simp add*: *H-ac*)+)
apply (*simp add*: *fdisj-def*, *metis*, *simp*)
apply (*simp add*: *fdisj-def fminus-def expand-fun-eq*)
apply (*rule conjI*, *metis*)
apply (*clarsimp split*: *option.splits simp add*: *map-add-def*)
apply ((*erule-tac x=x* **in** *allE*)+, (*erule disjE* | *simp*)+)
done

lemma *lt-safeS-mon* [*rule-format*]:
  ⟦ *lt-safeS n C s k tq* ; *H-def k k′*; *fdisj tq tq′* ⟧ ⟹
    lt-safeS n C s (*k* ⊙ *k′*) (*tq* ++ *tq′*)
apply (*subgoal-tac lt-safeS n C s k tq* ⟶ *?Q*, *erule* (*1*) *mp*,
    thin-tac lt-safeS n C s k tq)
apply (*induct n arbitrary*: *C s k tq*, *simp*, *clarsimp*)
apply (*rule conjI*, *erule* (2) *lt-safe-mon*)
apply (*rule conjI*, *clarsimp*)
apply (*drule-tac k=k* **and** *k′=k′* **and** *tq=tq* **and** *tq′=tq′* **in** *lt-sem-mon*,
    simp-all, *clarsimp*)
apply *clarsimp*
apply (*erule meta-allE*)+
apply (*erule* (*1*) *meta-impE*)+
apply (*erule mp*, *erule all-impD*)
apply (*erule* (*1*) *interfered-antimon*)
done

lemma *lt-semS-Frame* [*rule-format*]:
  stable F ⟹
   ∀ *C s k1 tq1*. *lt-sat F* (*s*, *k2*, *tq2*) ⟶
     H-def k1 k2 ⟶ *fdisj tq1 tq2* ⟶ *lt-safeS n C s k1 tq1* ⟶
     lt-semS n C s (*k1* ⊙ *k2*) (*tq1* ++ *tq2*) *c-Skip s′ k′ tq′* ⟶
     (∀ *m s′ k′ tq′*. *lt-semS m C s k1 tq1 c-Skip s′ k′ tq′* ⟶ *lt-sat Q* (*s′*, *k′*, *tq′*))
     ⟶ (∃ *k1 k2 tq1*.
         lt-sat Q (*s′*, *k1*, *tq1*) ∧
         (∃ *tq2*. *lt-sat F* (*s′*, *k2*, *tq2*) ∧ *H-def k1 k2* ∧ *k′* = *k1* ⊙ *k2*
             ∧ *fdisj tq1 tq2* ∧ *tq′* = *tq1* ++ *tq2*))

apply (*insert nat.induct* [*of* λ*n*.
     ∀ *C s k1 tq1*.
       lt-sat F (*s*, *k2*, *tq2*) ⟶
       H-def k1 k2 ⟶
       fdisj tq1 tq2 ⟶
       lt-safeS n C s k1 tq1 ⟶
       lt-semS n C s (*k1* ⊙ *k2*) (*tq1* ++ *tq2*) *c-Skip s′ k′ tq′* ⟶
       (∀ *m s′ k′ tq′*. *lt-semS m C s k1 tq1 c-Skip s′ k′ tq′* ⟶ *lt-sat Q* (*s′*, *k′*, *tq′*)) ⟶
       (∃ *k1 k2 tq1*.

lt-sat Q (*s′*, *k1*, *tq1*) ∧
(∃*tq2*. *lt-sat F* (*s′*, *k2*, *tq2*) ∧ *H-def k1 k2* ∧ *k′* = *k1* ⊙ *k2* ∧ *fdisj tq1 tq2* ∧ *tq′* = *tq1* ++ *tq2*)) *n*],
erule meta-impE, *erule-tac*[2] *meta-mp*)
apply (*clarsimp*)
apply (*erule-tac x=0* **in** *allE*, *simp*)
apply ((*rule exI*)+, *erule conjI*, *rule exI*, *erule conjI*, *simp*)
apply *clarsimp*
apply (*erule disjE*)
— Program transition
apply *clarsimp*
apply (*frule-tac k=k1* **and** *k′=k2* **and** *tq=tq1* **and** *tq′=tq2* **in** *lt-sem-mon*, *simp+*)
apply *clarsimp*
apply (*drule* (*1*) *stableE2*, *rule-tac k=k1* **in** *allowed-interfered*, *assumption*,
erule lt-sem-Guar)
apply (*drule* (*1*) *all2-impD*)
apply (*drule-tac a=k1a* **in** *all-impD*, *assumption*)
apply (*drule-tac a=tq1a* **in** *all-impD*, *assumption*)
apply (*drule* (*1*) *all5-impD*, *erule* (*2*) *imp3D*)
apply (*rule allI*, *rename-tac m*, *clarify*, *erule-tac a=Suc m* **in** *all4-impD*, *simp*)
apply ((*rule exI*)+, *erule conjI*, *rule disjI1*, *erule exI*)
— Environment transition
apply *clarsimp*
apply (*drule all-impD*, *erule* (*1*) *interfered-antimon*)
apply (*drule-tac s′=s′′* **in** *stableE2*, *assumption*, *rule-tac k′=k1* **in** *interfered-antimon*,
simp add: *H-ac*, *simp add*: *H-ac*)
apply (*drule* (*1*) *all2-impD*, *drule* (*1*) *all-impD*, *drule* (*1*) *all-impD*,
erule (*2*) *imp3D*)
apply (*rule allI*, *rename-tac m*, *clarify*, *erule-tac a=Suc m* **in** *all4-impD*, *simp*)
apply ((*rule exI*)+, *erule conjI*, *rule disjI2*, *simp*, *erule* (*1*) *interfered-antimon*)
done

theorem *Rule-Frame*:
⟦ *lt-Hoare P C Q*; *stable F* ⟧ ⟹ *lt-Hoare* (*a-Star P F*) *C* (*a-Star Q F*)
apply (*clarsimp simp add*: *lt-Hoare-def lt-has-spec-def*)
apply (*drule* (*1*) *all3-impD*, *clarsimp*)
apply (*rule conjI*, *rule allI*, *erule allE*, *erule* (*2*) *lt-safeS-mon*)
apply (*clarify*, *erule* (*3*) *lt-semS-Frame*, *erule* (*2*) *allE*, *erule* (*1*) *all4-impD*)
done

The standard Hoare logic consequence rule follows easily from the definitions:

theorem *Rule-Conseq*:
⟦ *lt-Hoare P C Q*; *lt-sat P′* ≤ *lt-sat P*; *lt-sat Q* ≤ *lt-sat Q′* ⟧
⟹ *lt-Hoare P′ C Q′*
apply (*clarsimp simp add*: *lt-Hoare-def lt-has-spec-def*)
apply (*drule all3-impD*, *erule* (*1*) *predicate1D*, *clarsimp*)
apply (*drule* (*1*) *all4-impD*, *erule* (*1*) *predicate1D*)
done

The next lemma is convenient for doing Isar transitivity proofs.

lemma *Rule-Conseq-trans*[*trans*]:
$P \simeq P' \Longrightarrow$ *lt-Hoare* P' C $Q \Longrightarrow$ *lt-Hoare* P C Q
$P \leq P' \Longrightarrow$ *lt-Hoare* P' C $Q \Longrightarrow$ *lt-Hoare* P C Q
lt-Hoare P C $Q' \Longrightarrow Q' \simeq Q \Longrightarrow$ *lt-Hoare* P C Q
lt-Hoare P C $Q' \Longrightarrow Q' \leq Q \Longrightarrow$ *lt-Hoare* P C Q
by (*auto elim*: *Rule-Conseq simp add*: *lt-sat-eq-def less-eq-lt-assn-def*)

Now, the proofs for Skip, atomic commands, Fork, and Join.

theorem *Rule-Skip*: *stable* $P \Longrightarrow$ *lt-Hoare P c-Skip P*
apply (*clarsimp simp add*: *lt-has-spec-def lt-Hoare-def lt-semS-Skip lt-safeS-def2*)
apply (*erule* (2) *stableEc*)
done

theorem *Rule-Atom*: $[\![$*stable P*; *stable Q*;
$\forall s\ k\ tq\ s'.$ *lt-sat* P $(s,k,tq) \longrightarrow A\ s\ s' \longrightarrow$ (*lt-sat* Q $(s',k,tq) \wedge$ *allowed* k $(s,s'))]\!] \Longrightarrow$
lt-Hoare P (*c-Atom A*) *Q*
apply (*clarsimp simp add*: *lt-Hoare-def lt-has-spec-def lt-safeS-def2 lt-semS-Prim*)
apply (*rule conjI*)
apply (*clarsimp*, *drule* (2) *stableEc*, *drule* (*1*) *all3-impD*)
apply (*drule* (*1*) *all-impD*, *simp*)
apply (*clarsimp*, *drule* (2) *stableEc*, *drule* (*1*) *all3-impD*)
apply (*erule lt-sem.cases*, *simp-all*, *clarsimp*)
apply (*drule* (*1*) *all-impD*, *clarify*)
apply (*erule* (2) *stableEc*)
done

theorem *Rule-Fork*:
$[\![$ *lt-Hoare* (*Rep-prec-assn P0*) *C Q0*;
$\forall s\ k\ tq\ n.$ *lt-sat* F $(s,k,tq) \longrightarrow$ *allowed* k $(s,s(v{:=}n))$;
nsupp v F;
lt-sat (*a-Star* (*a-Thread* $(\lambda s.\ s\ v)$ *Q0*) *F*) $\leq$ *lt-sat Q*;
stable (*Rep-prec-assn P0*); *stable F*; *stable Q* $]\!] \Longrightarrow$
lt-Hoare (*a-Star* (*Rep-prec-assn P0*) *F*) (*c-Fork v P0 Q0 C*) *Q*
apply (*simp add*: *lt-Hoare-def*, *thin-tac lt-wa ?a* $\wedge$ *?b*)
apply (*clarsimp simp del*: *lt-sat.simps simp add*: *lt-has-spec-def lt-safeS-def2 lt-semS-Prim*)
apply (*rule conjI*, *clarify*)
apply (*drule* (*3*) *stableEc* [*OF - stable-star*], *clarsimp*)
apply ((*rule exI* | *erule conjI* | *simp*)+)
apply (*erule all2-impD*, *erule exI*)
apply *clarify*
apply (*drule* (*3*) *stableEc* [*OF - stable-star*], *clarsimp*)
apply (*erule lt-sem.cases*, *simp-all*, *clarsimp*)
apply (*drule* (*7*) *precD*, *clarsimp*)
apply (*rule stableEc*, *simp-all*)
apply (*erule predicate1D*, *simp add*: *H-ac H-canc*)
apply (*rule exI*, *rule conjI*, *erule* (*1*) *nsuppD2*)
apply (*simp add*: *fdisj-def fsingl-def map-add-comm dom-def*)
done

theorem *Rule-Join*:
⟦ *stable* (*a-Star P* (*a-Thread E Q*)); *stable* (*a-Star P Q*) ⟧ ⟹
lt-Hoare (*a-Star P* (*a-Thread E Q*)) (*c-Join E*) (*a-Star P Q*)
apply (*clarsimp simp del*: *lt-sat.simps simp add*: *lt-Hoare-def lt-has-spec-def lt-safeS-def2 lt-semS-Prim*)
apply (*rule conjI*, *clarify*)
apply (*drule* (*2*) *stableEc*, *clarsimp*)
apply (*simp split*: *option.splits add*: *map-add-def fdisj-def fsingl-def*)
apply (*clarify*, *drule* (*2*) *stableEc*)
apply (*erule lt-sem.cases*, *simp-all del*: *lt-sat.simps*, *clarify*)
apply (*rule stableEc*, *simp-all*)
apply (*clarsimp simp add*: *fsingl-def fminus-def fdisj-def map-add-def*)
apply ((*rule exI* | *erule conjI*)+, *simp split*: *option.splits*)
apply (*simp split*: *option.splits split-if-asm add*: *expand-fun-eq*)
apply (*metis*)
done

Proving the sequential composition rule requires the following lemmas about the operational semantics:

lemma *lt-safeS-Seq* [*rule-format*]:
⟦ *lt-safeS n C1 s k tq* ⟧
⟹ (∀*s′ k′ tq′*. (∃*n*. *lt-semS n C1 s k tq c-Skip s′ k′ tq′*) ⟶ (∀*n*. *lt-safeS n C2 s′ k′ tq′*))
⟶ *lt-safeS n* (*C1* ;; *C2*) *s k tq*
apply (*induct n arbitrary*: *C1 s k tq*, *simp*, *clarsimp*)
apply (*rule conjI*, *clarsimp*)
— Program transition
apply (*erule lt-sem.cases*, *simp-all*)
apply (*drule all3-impD*, *rule-tac x=0* **in** *exI*, *simp*, (*rule conjI* | *simp*)+)
apply *clarsimp*
apply (*drule* (*1*) *all5-impD*)
apply ((*erule meta-allE*)+, *erule* (*1*) *meta-impE*, *erule mp*)
apply ((*rule allI*)+, *rule impI*, *erule exE*, *rename-tac m*, *erule all3-impD*)
apply (*rule-tac x=Suc m* **in** *exI*, *simp*)
apply ((*rule exI*)+, *erule conjI*, *rule disjI1*, *erule exI*)
— Environment transition
apply (*clarify*, *drule* (*1*) *all-impD*)
apply ((*erule meta-allE*)+, *erule* (*1*) *meta-impE*, *erule mp*)
apply ((*rule allI*)+, *rule impI*, *erule exE*, *rename-tac m*, *erule all3-impD*)
apply (*rule-tac x=Suc m* **in** *exI*, *simp*)
apply ((*rule exI*)+, *erule conjI*, *rule disjI2*, *simp*)
done

lemma *lt-semS-Seq*: *lt-semS n* (*C1* ;; *C2*) *s k tq c-Skip s″ k″ tq″* ⟹
∃*s′ k′ tq′ n1 n2*. *n1*+*n2* ≤ *n*
∧ *lt-semS n1 C1 s k tq c-Skip s′ k′ tq′*
∧ *lt-semS n2 C2 s′ k′ tq′ c-Skip s″ k″ tq″*
apply (*induct n arbitrary*: *C1 s k tq*, *simp*, *clarsimp*)
apply (*erule disjE*, *clarify*)
— Program action
apply (*erule lt-sem.cases*, *simp-all*)

```
 apply (rule exI, rule exI, rule exI,
       rule-tac x=0 in exI, rule-tac x=n in exI, (rule conjI |simp)+)
 apply clarsimp
 apply ((erule meta-allE)+, erule (1) meta-impE, clarsimp, rename-tac m1 m2)
 apply (rule exI, rule exI, rule exI,
       rule-tac x=Suc m1 in exI, rule-tac x=m2 in exI, simp)
 apply (rule conjI) prefer 2 apply (assumption)
 apply ((rule exI)+, erule conjI, rule disjI1, erule exI)
— Environment action
apply ((erule meta-allE)+, erule (1) meta-impE, clarsimp, rename-tac m1 m2)
apply (rule exI, rule exI, rule exI,
      rule-tac x=Suc m1 in exI, rule-tac x=m2 in exI, simp)
apply (rule conjI) prefer 2 apply (assumption)
apply ((rule exI)+, erule conjI, rule disjI2, simp)
done

theorem Rule-Seq [trans]:
  ⟦lt-Hoare P C1 Q; lt-Hoare Q C2 R⟧ ⟹ lt-Hoare P (C1 ;; C2) R
apply (clarsimp simp add: lt-Hoare-def lt-has-spec-def)
apply (drule (1) all3-impD, clarify)
apply (rule conjI, rule allI)
 apply (rule lt-safeS-Seq, erule (1) allE)
 apply (erule exE, (erule allE)+, erule (1) impE, erule (1) impE, erule conjE, erule (1) allE)
apply clarify
apply (drule lt-semS-Seq, clarsimp)
apply ((erule allE)+, erule (1) impE, erule (1) impE, erule conjE, (erule allE)+, erule (2) impE)
done
```

The rule for non-deterministic choice also requires two helper lemmas:

```
lemma lt-safeS-Choice:
  ⟦ lt-safeS n C1 s k tq; lt-safeS n C2 s k tq ⟧ ⟹ lt-safeS n (c-Choice C1 C2) s k tq
apply (induct n arbitrary: s, simp, clarsimp)
apply (erule lt-sem.cases, simp-all)
apply (clarsimp, erule all-impD, rule interfered-refl)+
done

lemma lt-semS-Choice:
  ⟦ lt-semS n (c-Choice C1 C2) s k tq c-Skip s′ k′ tq′ ⟧ ⟹
  lt-semS n C1 s k tq c-Skip s′ k′ tq′ ∨ lt-semS n C2 s k tq c-Skip s′ k′ tq′
apply (induct n arbitrary: s, simp, clarsimp)
apply (erule disjE, clarsimp, erule lt-sem.cases, simp-all)
  apply ((rule exI)+, erule conjI, simp add: interfered-refl)
 apply (drule (1) all4-impD, simp add: interfered-refl)
apply (erule meta-allE, erule (1) meta-impE, clarsimp)
apply (thin-tac lt-semS n (c-Choice C1 C2) ?s ?k ?tq c-Skip ?s′ ?k′ ?tq′)
apply (erule disjE)
 apply ((rule exI)+, erule conjI, simp)
apply (drule (1) all4-impD, simp)
done
```

theorem *Rule-Choice*:
⟦*lt-Hoare P C1 Q*; *lt-Hoare P C2 Q*⟧ ⟹ *lt-Hoare P* (*c-Choice C1 C2*) *Q*
apply (*clarsimp simp add*: *lt-has-spec-def lt-Hoare-def*)
apply (*drule* (*1*) *all3-impD*, *erule conjE*)+
apply (*rule conjI*, *rule allI*, *rule lt-safeS-Choice*, (*erule* (*1*) *allE*)+)
apply (*clarify*, *drule lt-semS-Choice*, *erule disjE*, (*drule* (*2*) *all4-impD*)+)
done

D.3.3 Global operational semantics

types
′a gt-queue = *nat* ⇀ (*′a*::*res-algebra*) *cmd* × (*state*,*′a*) *permDG* × (*state*,*′a*) *lt-queue*
′a gt-conf = *state* × (*′a*::*res-algebra*) *gt-queue*

Global thread queues map thread identifiers to commands, permissions and their local thread queues. The last two components are logical and are not present in the machine semantics.

A global configuration consists of the state and a global thread queue.

definition
gt-safe :: (*′a*::*res-algebra*) *gt-conf* ⇒ *bool*
where
gt-safe x ≡ (*case x of* (*s*,*d*) ⇒ ∀ *tid C k tq*. *d tid* = *Some* (*C*,*k*,*tq*) ⟶
lt-safe C s k tq
∧ (∀ *tid0 k3 tq3 C′ s′ k′ tq′*. *lt-sem C s k tq* (*lab-Join tid0 k3 tq3*) *C′ s′ k′ tq′*
⟶ (*case* (*d tid0*) *of*
None ⇒ *False*
| *Some* (*C0′*,*k0*,*tq0*) ⇒ *C0′*=*c-Skip* ⟶
(∃*C′ s′ k′ tq′*. *lt-sem C s k tq* (*lab-Join tid0 k0 tq0*) *C′ s′ k′ tq′*))))

A global configuration is safe if and only if it does not abort in a single step.

inductive
gt-sem :: (*′a*::*res-algebra*) *gt-conf* ⇒ *′a gt-conf* ⇒ *bool*
where
gt-noneI[*elim*!]:
⟦ *lt-sem C s k tq lab-None C′ s′ k′ tq′*;
d tid = *Some* (*C*,*k*,*tq*); *d′* = *d*(*tid*:= *Some* (*C′*,*k′*,*tq′*)) ⟧
⟹ *gt-sem* (*s*,*d*) (*s′*,*d′*)
| *gt-forkI*[*elim*!]:
⟦ *lt-sem C s k tq* (*lab-Fork tid2 C2 k2 tq2 Q2*) *C′ s′ k′ tq′*;
d tid = *Some* (*C*,*k*,*tq*); *d tid2* = *None*;
d′ = (*d*(*tid*:=*Some*(*C′*,*k′*,*tq′*)))(*tid2* := *Some*(*C2*,*k2*,*tq2*)) ⟧
⟹ *gt-sem* (*s*,*d*) (*s′*,*d′*)
| *gt-joinI*[*elim*!]:
⟦ *lt-sem C s k tq* (*lab-Join tid2 k2 tq2*) *C′ s′ k′ tq′*; *tid* ≠ *tid2*;
d tid = *Some* (*C*,*k*,*tq*); *d tid2* = *Some* (*c-Skip*,*k2*,*tq2*);
d′ = (*d*(*tid*:=*Some*(*C′*,*k′*,*tq′*)))(*tid2*:=*None*) ⟧
⟹ *gt-sem* (*s*,*d*) (*s′*,*d′*)

definition *combine-perms* :: *bool* × *′a*::*res-algebra* ⇒ *bool* × *′a* ⇒ *bool* × *′a* **where**

combine-perms x y ≡ (*fst x* ∧ *fst y* ∧ *H-def* (*snd x*) (*snd y*), *snd x* ⊙ *snd y*)

definition
combine-queues :: (*state*,′*a*::*res-algebra*) *lt-queue option* ⇒
(*state*,′*a*) *lt-queue option* ⇒
(*state*,′*a*) *lt-queue option*
where
combine-queues x y ≡
(*case x of None* ⇒ *None* | *Some x* ⇒
(*case y of None* ⇒ *None* | *Some y* ⇒
if fdisj x y then Some (*x*++*y*) *else None*))

definition *perms* :: (′*a*::*res-algebra*) *gt-queue* ⇒ *bool* × (*state*,′*a*) *permDG* **where**
perms A ≡ *fold combine-perms*
(λ*x*. (*True*, *case A x of None* ⇒ *H-zero* | *Some* (-,*x*,-) ⇒ *x*))
(*True*, *H-zero*)
(*dom A*)

definition *queues* :: (′*a*::*res-algebra*) *gt-queue* ⇒ (*state*,′*a*) *lt-queue option* **where**
queues A ≡ *fold combine-queues*
(λ*x*. *case A x of None* ⇒ *None* | *Some* (-,-,*x*) ⇒ *Some x*)
(*Some empty*)
(*dom A*)

definition *gt-wf* :: (′*a*::*res-algebra*) *gt-conf* ⇒ *bool* **where**
gt-wf x ≡ (*case x of* (*s*,*d*) ⇒
finite (*dom d*) ∧ *fst* (*perms d*) ∧ (*case queues d of None* ⇒ *False* | *Some Gamma* ⇒
dom Gamma ⊆ *dom d* ∧
(∀ *tid C k tq*. *d tid* = *Some* (*C*,*k*,*tq*) ⟶
lt-wa C ∧ (∀ *n*. *lt-safeS n C s k tq*)
∧ (∀ *Q n s*′ *k*′ *tq*′. *Gamma tid* = *Some Q* ⟶
lt-semS n C s k tq c-Skip s′ *k*′ *tq*′ ⟶ *lt-sat Q* (*s*′,*k*′,*tq*′)))))

A global configuration is well formed if and only if it is safe for an arbitrary number of transitions and all threads satisfy their expected postconditions upon termination.

lemma (**in** *comm-monoid-mult*) *vfold-insert*:
⟦*finite A*⟧ ⟹ *fold* (*op* ∗) *g z* (*insert x A*) = *g x* ∗ (*fold* (*op* ∗) *g z* (*A* − {*x*}))
by (*subst fold-insert* [*THEN sym*], *simp-all*)

interpretation *comb-p*: *comm-monoid-mult* [(*True*,*H-zero*) *combine-perms*]
by (*unfold-locales*) (*auto simp add*: *combine-perms-def H-ac*)

interpretation *comb-q*: *comm-monoid-mult* [*Some empty combine-queues*]
by (*unfold-locales*)
(*auto split*: *option.splits simp add*: *combine-queues-def fdisj-def2 map-add-comm*)

lemma *pq-simps1*[*simp*]:
perms (*fsingl tid* (*C*,*k*,*tq*)) = (*True*,*k*)
queues (*fsingl tid* (*C*,*k*,*tq*)) = *Some tq*

by (*simp-all add*: *perms-def queues-def fsingl-def dom-def*)

lemma *pq-simps2*[*simp*]:
 finite (*dom d*) $\Longrightarrow$
 perms (*d*(*tid*:=*Some*(*C*,*k*,*tq*))) = *combine-perms* (*True*,*k*) (*perms* (*d*(*tid*:=*None*)))
 finite (*dom d*) $\Longrightarrow$
 queues (*d*(*tid*:=*Some*(*C*,*k*,*tq*))) = *combine-queues* (*Some tq*) (*queues* (*d*(*tid*:=*None*)))
 apply (*simp add*: *perms-def comb-p.vfold-insert*)
 apply (*rule comb-p.fold-cong* [*THEN arg-cong*], *simp*, *simp*)
apply (*simp add*: *queues-def comb-q.vfold-insert*)
apply (*rule comb-q.fold-cong* [*THEN arg-cong*], *simp*, *simp*)
done

lemma *pq-expand*:
 [[*finite* (*dom d*); *d tid* = *Some* (*C*,*k*,*tq*)]] $\Longrightarrow$
 perms d = *combine-perms* (*True*,*k*) (*perms* (*d*(*tid*:=*None*)))
 [[*finite* (*dom d*); *d tid* = *Some* (*C*,*k*,*tq*)]] $\Longrightarrow$
 queues d = *combine-queues* (*Some tq*) (*queues* (*d*(*tid*:=*None*)))
 apply (*rule-tac t*=*d* **and** *s*=*d*(*tid* := *Some*(*C*,*k*,*tq*)) **in** *subst*, *erule-tac* [2] *pq-simps2*)
 apply (*simp add*: *fun-upd-def expand-fun-eq*)
apply (*rule-tac t*=*d* **and** *s*=*d*(*tid* := *Some*(*C*,*k*,*tq*)) **in** *subst*, *erule-tac* [2] *pq-simps2*)
apply (*simp add*: *fun-upd-def expand-fun-eq*)
done

lemma *pq-expand2*:
 [[*finite* (*dom d*); *d tid* = *Some* (*C*,*k*,*tq*); *tid* ≠ *tid*′; *d tid*′ = *Some* (*C*′,*k*′,*tq*′)]] $\Longrightarrow$
 perms d = *combine-perms* (*True*,*k*) (*combine-perms* (*True*,*k*′) (*perms* (*d*(*tid*:=*None*,*tid*′:=*None*))))
 [[*finite* (*dom d*); *d tid* = *Some* (*C*,*k*,*tq*); *tid* ≠ *tid*′; *d tid*′ = *Some* (*C*′,*k*′,*tq*′)]] $\Longrightarrow$
 queues d = *combine-queues* (*Some tq*) (*combine-queues* (*Some tq*′) (*queues* (*d*(*tid*:=*None*,*tid*′:=*None*))))
 apply (*rule-tac t*=*d* **and** *s*=*d*(*tid* := *Some*(*C*,*k*,*tq*)) **in** *subst*,
 simp add: *fun-upd-def expand-fun-eq*)
 apply (*rule-tac t*=*d* **and** *s*=*d*(*tid*′ := *Some*(*C*′,*k*′,*tq*′)) **in** *subst*,
 simp add: *fun-upd-def expand-fun-eq*)
 apply (*subst pq-simps2*, *simp*, *simp*)
 apply (*subst fun-upd-twist*, *simp*)
 apply (*subst pq-simps2*, *simp*, *simp*)
apply (*rule-tac t*=*d* **and** *s*=*d*(*tid* := *Some*(*C*,*k*,*tq*)) **in** *subst*,
 simp add: *fun-upd-def expand-fun-eq*)
apply (*rule-tac t*=*d* **and** *s*=*d*(*tid*′ := *Some*(*C*′,*k*′,*tq*′)) **in** *subst*,
 simp add: *fun-upd-def expand-fun-eq*)
apply (*subst pq-simps2*, *simp*, *simp*)
apply (*subst fun-upd-twist*, *simp*)
apply (*subst pq-simps2*, *simp*, *simp*)
done

lemma *gt-wfI*:
 [[*lt-Hoare P C Q*; *lt-sat P* (*s*,*k*,*empty*)]]
 $\Longrightarrow$ *gt-wf* (*s*, *fsingl tid* (*C*,*k*,*empty*))
apply (*simp add*: *gt-wf-def lt-Hoare-def lt-has-spec-def*)

apply (*simp add*: *dom-def fsingl-def*)
done

lemma *lt-sem-None* [*rule-format*]:
$lt\text{-}sem\ C\ s\ k\ tq\ lab\ C'\ s'\ k'\ tq' \Longrightarrow lab{=}lab\text{-}None \longrightarrow k'{=}k \wedge tq'{=}tq$
by (*erule lt-sem.induct*, *simp-all*)

lemma *combine-def*:
$[\![\ fst\ (combine\text{-}perms\ (True,k)\ (perms\ d));\ finite\ (dom\ d);$
$\exists tid\ C'\ tq'.\ d\ tid = Some\ (C',k',tq')\ ]\!] \Longrightarrow H\text{-}def\ k\ k'$
apply (*clarsimp simp add*: *pq-expand*(*1*) [**where** *d=d*])
apply (*auto simp add*: *combine-perms-def*)
done

lemma *lt-sem-wa*[*rule-format*]:
$lt\text{-}sem\ C\ s\ k\ tq\ lab\ C'\ s'\ k'\ tq'$
$\Longrightarrow lt\text{-}wa\ C$
$\longrightarrow (lt\text{-}wa\ C' \wedge (\forall tid0\ C0\ k0\ tq0\ Q0.\ lab = lab\text{-}Fork\ tid0\ C0\ k0\ tq0\ Q0 \longrightarrow lt\text{-}wa\ C0))$
by (*erule lt-sem.induct*, *simp-all* (*no-asm*), *fast*)

lemma *lt-sem-Fork*[*rule-format*]:
$lt\text{-}sem\ C\ s\ k\ tq\ lab\ C'\ s'\ k'\ tq' \Longrightarrow$
$lab = lab\text{-}Fork\ tid0\ C0\ k0\ tq0\ Q0 \longrightarrow$
$(\exists v.\ s'{=}s(v{:=}tid0) \wedge allowed\ k'\ (s,s')) \wedge H\text{-}def\ k0\ k' \wedge k{=}k0 \odot k'$
$\wedge fdisj\ tq0\ tq' \wedge tq\ tid0 = None \wedge tq{+}{+}fsingl\ tid0\ Q0{=}tq0{+}{+}tq'$
apply (*erule lt-sem.induct*, *simp-all*, *clarsimp*)
apply (*rule conjI*, *simp add*: *expand-fun-eq*, *metis*)
apply (*simp add*: *fdisj-def2 fsingl-def dom-def Int-def fun-upd-def*)
done

lemma *lt-sem-Join*[*rule-format*]:
$lt\text{-}sem\ C\ s\ k\ tq\ lab\ C'\ s'\ k'\ tq' \Longrightarrow$
$lab = lab\text{-}Join\ tid0\ k0\ tq0 \longrightarrow$
$s'{=}s \wedge H\text{-}def\ k\ k0 \wedge k'{=}k \odot k0 \wedge fdisj\ tq\ tq0 \wedge tq\ tid0 \neq None \wedge tq'{=}fminus\ (tq{+}{+}tq0)\ tid0$
by (*erule lt-sem.induct*, *simp-all*, *clarsimp*)

lemma *lt-sem-Join2*[*rule-format*]:
$lt\text{-}sem\ C\ s\ k\ tq\ lab\ C'\ s'\ k'\ tq' \Longrightarrow$
$lab = lab\text{-}Join\ tid0\ k0\ tq0 \longrightarrow tq\ tid0 = Some\ q \longrightarrow$
$(\forall k0'\ tq0'.\ H\text{-}def\ k\ k0' \wedge fdisj\ tq\ tq0' \wedge lt\text{-}sat\ q\ (s,k0',tq0') \longrightarrow$
$(\exists k'\ tq'.\ lt\text{-}sem\ C\ s\ k\ tq\ (lab\text{-}Join\ tid0\ k0'\ tq0')\ C'\ s'\ k'\ tq'))$
apply (*erule lt-sem.induct*, *simp-all*)
apply (*clarsimp*, *drule all2-impD*, *erule conjI*, *erule* (*1*) *conjI*, *clarify*)
apply (*rule exI*, *rule exI*, *erule seqI*)
apply *fast*
done

lemma *lt-wa-Fork*:
$[\![lt\text{-}wa\ C;\ lt\text{-}sem\ C\ s\ (k2 \odot k')\ tq\ (lab\text{-}Fork\ tid2\ C2\ k2\ tq2\ Q2)\ C'\ (s(v := tid2))\ k'\ tq'\ ]\!]$

⟹ (∀*n*. *lt-safeS n C2* (*s*(*v* := *tid2*)) *k2 tq2*) ∧
(∀*n s′ k′ tq′*. *lt-semS n C2* (*s*(*v* := *tid2*)) *k2 tq2 c-Skip s′ k′ tq′*
⟶ *lt-sat Q2* (*s′*, *k′*, *tq′*))
apply (*induct C arbitrary*: *C′*, *simp-all add*: *lt-sem-Skip*)
apply (*erule-tac* [*1*−*6*] *lt-sem.cases*, *simp-all*)
— Fork
apply (*clarsimp simp add*: *lt-has-spec-def*)
apply (*drule* (*1*) *all3-impD*, *clarify*)
apply (*rule conjI*, *clarify*, *rename-tac m*, *erule-tac x=Suc m* **in** *allE*, *clarsimp*)
apply (*erule all-impD*, *rule-tac k=k′* **in** *allowed-interfered*,
simp add: *H-ac*, *assumption*)
apply (*rule allI*, *rename-tac m*, *clarify*, *erule-tac a=Suc m* **in** *all4-impD*, *simp*)
apply ((*rule exI*)+, *erule conjI*, *rule disjI2*, *simp*,
rule-tac k=k′ **in** *allowed-interfered*, *simp add*: *H-ac*, *assumption*)
done

Next, we prove the two main lemmas for proving the soundness of the global semantics: (1) Well-formed configurations do not fail in one step. (2) Reduction preserved well-formedness.

lemma *gt-wf-safe*:
gt-wf (*s*,*d*) ⟹ *gt-safe* (*s*,*d*)
apply (*clarsimp split*: *option.split-asm simp add*: *gt-wf-def gt-safe-def*)
apply (*rule conjI*)
apply (*drule* (*1*) *all4-impD*, *clarsimp*)
apply (*erule-tac x=Suc 1* **in** *allE*, *simp*)
apply (*clarify*)
apply (*frule lt-sem-Join*, *fast*, *clarsimp split*: *option.splits*)
apply (*drule-tac a=tid0* **in** *allD*)
apply (*rule conjI*)
— First prove that *d tid0* ≠ *None*.
apply (*simp split*: *option.splits split-if-asm add*: *pq-expand combine-queues-def*)
apply (*drule-tac A=tq* **and** *B=a* **in** *dom-Some*)
apply (*erule-tac t=a* **in** *subst*, *simp add*: *Un-upper2*, *erule exE*)
apply (*drule* (*1*) *dom-Some*, *simp*)
— Main case
apply *clarsimp*
apply (*subgoal-tac tid* ≠ *tid0*)
prefer *2* **apply** (*erule contrapos-pn*, *simp add*: *lt-sem-Skip*)
apply (*drule-tac lt-sem-Join2*, *fast*, *assumption*)
prefer *2* **apply** *fast*
apply (*frule* (*3*) *pq-expand2*(*1*), *frule* (*3*) *pq-expand2*(*2*))
apply (*clarsimp split*: *option.splits split-if-asm*
simp add: *combine-perms-def combine-queues-def H-ac fdisj-break*)
apply (*rule conjI*, *erule def-starE*)
apply (*drule all-impD*)
apply (*simp split*: *option.split-asm add*: *map-add-def fdisj-def*)
apply ((*erule-tac x=tid0* **in** *allE*)+, (*simp* | *erule disjE*)+)
apply (*erule-tac a=0* **in** *all4-impD*, *simp*)
done

lemma *gt-wf-step*:
  ⟦ *gt-wf* (*s*,*d*); *gt-sem* (*s*,*d*) (*s*′,*d*′) ⟧ ⟹ *gt-wf* (*s*′,*d*′)
apply (*erule gt-sem.cases*)
— Simple transition
 apply (*clarsimp split*: *option.splits simp add*: *gt-wf-def*)
 apply (*simp add*: *pq-expand* [**where** *d*=*d*])
 apply (*frule lt-sem-None*, *simp*, *clarsimp*)
 apply (*rule conjI*, *erule subset-insertI2*)
 apply (*clarsimp*, *rename-tac tidX*, *erule-tac x*=*tidX* **in** *allE*)
 apply (*rule conjI*, *clarsimp*)
 — Case *tidX* = *tid*
  apply (*rule conjI*, *frule* (*1*) *lt-sem-wa*, *erule* (*1*) *conjE*)
  apply (*rule conjI*, *rule allI*, *rename-tac m*, *erule-tac x*=*Suc m* **in** *allE*, *simp*)
  apply (*clarify*, *drule* (*1*) *all-impD*)
  apply (*erule-tac a*=*Suc n* **in** *all4-impD*, *simp*)
  apply ((*rule exI*)+, *erule conjI*, *rule disjI1*, *erule exI*)
 — Case *tidX* ≠ *tid*
 apply (*clarify*, *drule* (*1*) *all3-impD*, *clarsimp*)
 apply (*rule conjI*, *clarsimp*)
  apply (*erule-tac x*=*Suc n* **in** *allE*, *clarsimp*, *erule all-impD*)
  apply (*rule allowed-interfered*, *erule-tac* [2] *lt-sem-Guar*)
  apply (*erule combine-def*, *simp*, *simp*, *fast*)
 apply (*clarify*, *drule* (*1*) *all-impD*)
 apply (*erule-tac a*=*Suc n* **in** *all4-impD*, *clarsimp*)
 apply ((*rule exI*)+, *erule conjI*, *rule disjI2*, *simp*)
 apply (*rule allowed-interfered*, *erule-tac* [2] *lt-sem-Guar*)
 apply (*erule combine-def*, *simp*, *simp*, *fast*)
— Fork
apply (*clarsimp split*: *option.splits simp add*: *gt-wf-def*)
apply (*subst* (*1 2 3*) *fun-upd-twist*, *erule contrapos-pn*, *simp*)
apply (*simp add*: *pq-expand* [**where** *d*=*d*])
apply (*subgoal-tac d*(*tid2*:=*None*) = *d*, *erule-tac* [2] *fun-upd-idem*, *clarsimp*)
apply (*frule lt-sem-Fork*, *fast*, *clarsimp*)
apply (*clarsimp split*: *option.split-asm split-if-asm*
    simp add: *combine-queues-def H-ac*, *rename-tac TQ*)
apply (*subgoal-tac fdisj* (*tq2* ++ *tq*′) *TQ*, *simp add*: *fdisj-break*)
 prefer *2*
 apply (*erule-tac t*=*tq2* ++ *tq*′ **in** *subst*, *simp add*: *fdisj-break fdisj-def fsingl-def*)
 apply (*erule* (*1*) *dom-None*)
apply (*clarsimp simp add*: *combine-perms-def H-ac*)
apply (*frule def-starD1*, *simp add*: *H-ac*)
apply (*rule conjI*, *rule subset-insertI2*, *erule subset-insertI2*)
apply (*subgoal-tac dom* (*tq2* ++ *tq*′) ⊆ *insert tid2* (*insert tid* (*dom d*)), *simp*)
 prefer *2*
 apply (*erule-tac t*=*tq2* ++ *tq*′ **in** *subst*, *simp add*: *fsingl-def*)
 apply (*rule conjI*, *simp add*: *dom-def*, *rule subset-insertI2*, *erule subset-insertI2*)
apply (*subgoal-tac lt-wa C*) **prefer** *2* **apply** *fast*
apply (*rule allI*, *rename-tac tidX*, *erule-tac x*=*tidX* **in** *allE*)
apply (*case-tac tidX* = *tid*, *clarsimp*)

— Case *tidX* = *tid*
apply (*rule conjI*, *clarsimp*, *clarsimp*)
apply (*rule conjI*, *frule* (*1*) *lt-sem-wa*, *erule* (*1*) *conjE*)
apply (*rule conjI*, *rule allI*, *rename-tac m*, *erule-tac x=Suc m* **in** *allE*, *simp*)
apply (*rule allI*, *rule impI*, *rule allI*, *rename-tac m*, *clarify*)
apply (*drule all-impD*) **prefer** *2*
apply (*erule-tac a=Suc m* **in** *all4-impD*, *clarsimp*)
apply ((*rule exI*)+, *erule conjI*, *rule disjI1*, *erule exI*)
apply (*simp* (*no-asm*) *add*: *map-add-def*)
apply (*rule-tac t=tq tid* **and** *s*=(*tq2* ++ *tq′*) *tid* **in** *subst*)
apply (*erule-tac t=tq2* ++ *tq′* **in** *subst*, *simp add*: *map-add-def fsingl-def*)
apply (*erule disjE*, *simp*, *simp*)
— Case *tidX* = *tid2*
apply (*simp*, *case-tac tidX* = *tid2*, *clarsimp*)
apply (*rule conjI*, *drule* (*1*) *lt-sem-wa*, *clarsimp*)
apply (*subgoal-tac* (*tq2* ++ *tq′* ++ *TQ*) *tid2* = *Some Q2*, *simp*)
apply (*erule* (*1*) *lt-wa-Fork*)
apply (*erule-tac t=tq2* ++ *tq′* **in** *subst*)
apply (*simp* (*no-asm*) *split*: *option.splits add*: *map-add-def fsingl-def*)
apply (*drule* (*1*) *dom-None*, *simp*)
— Third case
apply (*clarsimp*)
apply (*erule-tac t=tq2* ++ *tq′* **in** *subst*)
apply (*rule conjI*, *clarsimp simp add*: *fun-upd-def* [*THEN symmetric*])
apply (*erule-tac x=Suc n* **in** *allE*, *clarsimp*, *erule all-impD*)
apply (*erule allowed-interfered* [*rotated*])
apply (*rule combine-def*, *simp add*: *combine-perms-def*, *erule* (*1*) *conjI*, *simp*)
apply (*rule-tac x=tidX* **in** *exI*, *simp*)
apply (*rule allI*, *rename-tac Z*, *rule impI*, *drule-tac a=Z* **in** *all-impD*)
apply (*simp split*: *option.split-asm add*: *map-add-def fsingl-def*)
apply (*rule allI*, *rename-tac m*, *clarify*)
apply (*erule-tac a=Suc m* **in** *all4-impD*,
clarsimp simp add: *fun-upd-def* [*THEN symmetric*])
apply ((*rule exI*)+, *erule conjI*, *rule disjI2*, *simp*, *erule allowed-interfered* [*rotated*])
apply (*rule combine-def*, *simp add*: *combine-perms-def*, *erule* (*1*) *conjI*, *simp*)
apply (*rule-tac x=tidX* **in** *exI*, *simp*)
— Join
apply (*clarsimp split*: *option.splits simp add*: *gt-wf-def*)
apply (*subst* (*1 2 3*) *fun-upd-twist*, *assumption*)
apply (*simp add*: *pq-expand*)
apply (*subgoal-tac d* = *d*(*tid2*:=*Some* (*c-Skip*,*k2*,*tq2*)),
erule-tac [2] *fun-upd-idem* [*THEN sym*])
apply (*frule-tac P=λd*. *combine-queues* (*Some tq*) (*queues* (*d*(*tid* := *None*))) = *Some a* **in** *subst*, *assumption*)
apply (*drule-tac P=λd*. *fst* (*combine-perms* (*True*, *k*) (*perms* (*d*(*tid* := *None*)))) **in** *subst*, *assumption*)
apply (*subst* (*asm*) (*1 2*) *fun-upd-twist*, *erule-tac* [*1*−*2*] *not-sym*, *simp*)
apply (*subst* (*1 2 3*) *fun-upd-twist*, *erule-tac not-sym*)
apply (*clarsimp split*: *option.split-asm split-if-asm*
simp add: *combine-queues-def combine-perms-def*, *rename-tac TQ*)

apply (*frule lt-sem-Join*, *fast*, *clarsimp*)
apply (*rule conjI*) **prefer** *2* **apply** (*rule fdisj-minus*, *simp add*: *fdisj-break*)
apply (*clarsimp simp add*: *H-ac fdisj-break dom-fminus*)
apply (*rule conjI*)
 apply (*rule-tac s=dom TQ* − {*tid2*} **and** *t=dom TQ* **in** *subst*)
  apply (*rule Diff-triv*, *simp add*: *fdisj-def dom-def*)
  apply (*erule-tac x=tid2* **in** *allE*, *erule disjE*, *simp*, *assumption*)
 apply (*rule Diff-mono*, *erule subset-insertI2*, *rule subset-refl*)
apply (*rule conjI*)
 apply (*rule Diff-mono*, *rule subset-insertI2*, *erule* (*1*) *Un-least*, *rule subset-refl*)
apply (*rule allI*, *rename-tac tidX*, *erule-tac x=tidX* **in** *allE*)
apply (*subgoal-tac*
    tidX ≠ *tid2* ⟶
    (*fminus* (*tq* ++ *tq2*) *tid2* ++ *TQ*) *tidX* = (*tq* ++ *tq2* ++ *TQ*) *tidX*)
 prefer *2* **apply** (*simp* (*no-asm*) *add*: *map-add-def fminus-def*)
apply (*case-tac tidX* = *tid*, *clarsimp*)
 — Case *tidX* = *tid*
 apply (*rule conjI*, *frule* (*1*) *lt-sem-wa*, *erule* (*1*) *conjE*)
 apply (*rule conjI*, *rule allI*, *rename-tac m*, *erule-tac x=Suc m* **in** *allE*, *simp*)
 apply (*rule allI*, *rule impI*, *drule* (*1*) *all-impD*)
 apply (*rule allI*, *rename-tac m*, *clarify*, *erule-tac a=Suc m* **in** *all4-impD*, *simp*)
 apply ((*rule exI*)+, *erule conjI*, *rule disjI1*, *erule exI*)
 — Case *tidX* ≠ *tid*
apply (*clarsimp*)
apply (*erule disjE*, *simp*, *clarsimp*)
apply (*drule-tac a=Q* **in** *all-impD*, *simp split*: *option.splits add*: *map-add-def*)
apply (*erule-tac a=Suc n* **in** *all4-impD*, *simp*)
apply ((*rule exI*)+, *erule conjI*, *rule disjI2*, *simp add*: *interfered-refl*)
done

end

D.4 Machine semantics and proof construct erasure

theory *DGMachine*
imports *Main Heaps VHelper DGLogic*
begin

D.4.1 Definitions

First, we define machine commands (which do not have any annotations) and the machine semantics.

datatype *m-cmd* =
  mc-Skip
 | *mc-Seq* *m-cmd m-cmd*
 | *mc-Choice m-cmd m-cmd*
 | *mc-Loop* *m-cmd*
 | *mc-Atom* *state* ⇒ *state* ⇒ *bool*

```
 | mc-Fork   var m-cmd
 | mc-Join   state ⇒ nat

datatype m-lab =
   labm-None
 | labm-Fork nat m-cmd
 | labm-Join nat

inductive
  lm-sem :: m-cmd ⇒ state ⇒ m-lab ⇒ m-cmd ⇒ state ⇒ bool
where
  lm-skip-seqI[intro!]: lm-sem (mc-Seq mc-Skip C) s labm-None C s
| lm-seqI[intro!]:
   lm-sem C s lab C1 s' ⟹ lm-sem (mc-Seq C C') s lab (mc-Seq C1 C') s'
| lm-choice-lI[intro!]: lm-sem (mc-Choice C1 C2) s labm-None C1 s
| lm-choice-rI[intro!]: lm-sem (mc-Choice C1 C2) s labm-None C2 s
| lm-loopI[intro!]:
   lm-sem (mc-Loop C) s labm-None (mc-Choice mc-Skip (mc-Seq C (mc-Loop C))) s
| lm-atomI[intro!]:     A s s' ⟹ lm-sem (mc-Atom A) s labm-None mc-Skip s'
| lm-forkI[intro!]:     s' = s(v:=n) ⟹ lm-sem (mc-Fork v C) s (labm-Fork n C) mc-Skip s'
| lm-joinI[intro!]:     E s = n ⟹ lm-sem (mc-Join E) s (labm-Join n) mc-Skip s


definition
  gm-safe :: state × (nat ⇀ m-cmd) ⇒ bool
where
  gm-safe x ≡ (case x of (s,d) ⇒ ∀ tid C tid' C' s'.
     d tid = Some C ∧ lm-sem C s (labm-Join tid') C' s'
   ⟶ tid' ≠ tid ∧ d tid' ≠ None)

inductive
  gm-sem :: state × (nat ⇀ m-cmd) ⇒ state × (nat ⇀ m-cmd) ⇒ bool
where
  gm-noneI[elim!]:
    ⟦ lm-sem C s labm-None C' s'; d tid = Some C; d' = d(tid:= Some C') ⟧
    ⟹ gm-sem (s,d) (s',d')
| gm-forkI[elim!]:
    ⟦ lm-sem C s (labm-Fork tid2 C2) C' s';
      d tid = Some C; d tid2 = None; d' = d(tid ↦ C', tid2 ↦ C2) ⟧
    ⟹ gm-sem (s,d) (s',d')
| gm-joinI[elim!]:
    ⟦ lm-sem C s (labm-Join tid2) C' s'; tid ≠ tid2;
      d tid = Some C; d tid2 = Some mc-Skip;
      d' = d(tid:=Some C', tid2:=None) ⟧
    ⟹ gm-sem (s,d) (s',d')
```

Definition of erasure on commands, labels, and configurations.

```
primrec
  EC :: ('a::res-algebra) cmd ⇒ m-cmd
```

where
  EC c-Skip = *mc-Skip*
| *EC* (*c-Seq C D*) = *mc-Seq* (*EC C*) (*EC D*)
| *EC* (*c-Choice C D*) = *mc-Choice* (*EC C*) (*EC D*)
| *EC* (*c-Loop C*) = *mc-Loop* (*EC C*)
| *EC* (*c-Atom A*) = *mc-Atom A*
| *EC* (*c-Fork v P Q C*) = *mc-Fork v* (*EC C*)
| *EC* (*c-Join n*) = *mc-Join n*

lemma *EC-inv*:
$(EC\ X = mc\text{-}Skip) \longleftrightarrow (X = c\text{-}Skip)$
$(EC\ X = mc\text{-}Seq\ C\ D) \longleftrightarrow (\exists Y\ Z.\ X = c\text{-}Seq\ Y\ Z \wedge EC\ Y = C \wedge EC\ Z = D)$
$(EC\ X = mc\text{-}Choice\ C\ D) \longleftrightarrow (\exists Y\ Z.\ X = c\text{-}Choice\ Y\ Z \wedge EC\ Y = C \wedge EC\ Z = D)$
$(EC\ X = mc\text{-}Loop\ C) \longleftrightarrow (\exists Y.\ X = c\text{-}Loop\ Y \wedge EC\ Y = C)$
$(EC\ X = mc\text{-}Atom\ A) \longleftrightarrow (X = c\text{-}Atom\ A)$
$(EC\ X = mc\text{-}Fork\ v\ C) \longleftrightarrow (\exists P\ Q\ Y.\ X = c\text{-}Fork\ v\ P\ Q\ Y \wedge EC\ Y = C)$
$(EC\ X = mc\text{-}Join\ n) \longleftrightarrow (X = c\text{-}Join\ n)$
by (*induct X*, *simp-all*)

primrec
  EL :: (*'a*::*res-algebra*) *lt-lab* ⇒ *m-lab*
where
  EL lab-None = *labm-None*
| *EL* (*lab-Fork tid C k tq Q*) = *labm-Fork tid* (*EC C*)
| *EL* (*lab-Join tid k tq*) = *labm-Join tid*

definition
  ECfg :: *state* × (*'a*::*res-algebra*) *gt-queue* ⇒ *state* × (*nat* ⇀ *m-cmd*)
where
  ECfg x ≡ (*case x of* (*s*,*d*) ⇒
    (*s*, (*λtid. case d tid of None* ⇒ *None* | *Some* (*C*,*k*,*tq*) ⇒ *Some* (*EC C*))))

D.4.2 Theorems

We prove two theorems about the erasure: soundness and completeness. Soundness says that for every annotated transition there is a corresponding machine transition. Completeness says that for every well formed state and every machine transition from that state there is a corresponding annotated transition.

lemma *Local-soundness*:
$lt\text{-}sem\ C\ s\ k\ tq\ lab\ C'\ s'\ k'\ tq' \Longrightarrow lm\text{-}sem\ (EC\ C)\ s\ (EL\ lab)\ (EC\ C')\ s'$
by (*erule lt-sem.induct*, *auto simp add*: *fun-upd-def*)

theorem *Global-soundness*:
$gt\text{-}sem\ x\ y \Longrightarrow gm\text{-}sem\ (ECfg\ x)\ (ECfg\ y)$
apply (*erule gt-sem.cases*, *drule-tac* [*1*−*3*] *Local-soundness*, *simp-all add*: *ECfg-def*)
apply (*erule-tac tid*=*tid* **in** *gm-noneI*, *simp-all add*: *expand-fun-eq*)
apply (*erule-tac tid*=*tid* **in** *gm-forkI*, *simp-all add*: *expand-fun-eq*)
apply (*erule-tac tid*=*tid* **in** *gm-joinI*, *simp-all add*: *expand-fun-eq*)

done

lemma *Local-completeness*[*rule-format*]:
⟦ *lm-sem CM s lab CM′ s′* ⟧ ⟹ *lab* = *labm-None* ⟶ (∀ *C k tq*.
EC C = *CM* ⟶ *lt-safe C s k tq* ⟶
(∃ *C′ k′ tq′*. *EC C′* = *CM′* ∧ *lt-sem C s k tq lab-None C′ s′ k′ tq′*))
apply (*erule lm-sem.induct*, *simp-all add*: *EC-inv*)
apply (*clarsimp*, *fast*)+
done

lemma *Local-fork-completeness*[*rule-format*]:
⟦ *lm-sem CM s lab CM′ s′* ⟧ ⟹ *lab* = *labm-Fork tid0 CM0* ⟶ (∀ *C k tq*.
EC C = *CM* ⟶ *lt-safe C s k tq* ⟶ *tq tid0* = *None* ⟶
(∃ *C′ k′ tq′ C0 k0 tq0 Q0*. *EC C′* = *CM′* ∧ *EC C0* = *CM0*
∧ *lt-sem C s k tq* (*lab-Fork tid0 C0 k0 tq0 Q0*) *C′ s′ k′ tq′*))
apply (*erule lm-sem.induct*, *simp-all add*: *EC-inv*)
apply (*clarsimp simp add*: *fun-upd-def* [*THEN symmetric*], *fast*)+
done

lemma *Local-join-completeness*[*rule-format*]:
⟦ *lm-sem CM s lab CM′ s′* ⟧ ⟹ *lab* = *labm-Join tid0* ⟶ (∀ *C k tq*.
EC C = *CM* ⟶ *lt-safe C s k tq* ⟶
(∃ *Q0*. *tq tid0* = *Some Q0*) ∧
(∀ *Q0 k0 tq0*. *tq tid0* = *Some Q0* ⟶ *lt-sat Q0* (*s*,*k0*,*tq0*) ⟶
H-def k k0 ⟶ *fdisj tq tq0* ⟶
(∃ *C′ k′ tq′*. *EC C′* = *CM′*
∧ *lt-sem C s k tq* (*lab-Join tid0 k0 tq0*) *C′ s′ k′ tq′*)))
apply (*erule lm-sem.induct*, *simp-all add*: *EC-inv*)
apply (*clarsimp simp add*: *fun-upd-def* [*THEN symmetric*], *fast*)+
done

theorem *Global-completeness*[*rule-format*]:
⟦ *gm-sem X Y* ⟧ ⟹ ∀ *x*. *ECfg x* = *X* ⟶ *gt-wf x* ⟶
(∃ *y*. *gt-sem x y* ∧ *ECfg y* = *Y*)
apply (*erule gm-sem.cases*, *simp-all add*: *ECfg-def gt-wf-def*)
— Simple transition
apply (*clarsimp split*: *option.splits split-if-asm*)
apply (*drule Local-completeness*, *simp-all*)
apply (*drule* (*1*) *all4-impD*, *clarify*, *erule-tac x=1* **in** *allE*, *simp*)
apply (*clarsimp simp add*: *expand-fun-eq*)
apply (*rule exI*, *rule conjI*, *erule gt-noneI*, *simp*+)
— Fork
apply (*clarsimp split*: *option.splits*)
apply (*drule Local-fork-completeness*, (*rule conjI* | *simp*)+)
apply (*drule* (*1*) *all4-impD*, *clarify*, *erule-tac x=1* **in** *allE*, *simp*)
apply (*drule* (*1*) *dom-None*)
apply (*clarsimp split*: *option.splits split-if-asm*
simp add: *pq-expand combine-queues-def*)
apply (*clarsimp simp add*: *expand-fun-eq*)

apply (*rule exI*, *rule conjI*, *erule gt-forkI*, *simp+*)
apply (*clarsimp simp add*: *expand-fun-eq*)
— Join
apply (*clarsimp split*: *option.splits simp add*: *EC-inv*)
apply (*drule Local-join-completeness*, *simp+*)
apply (*drule-tac a=tid* **in** *all4-impD*, *assumption*, *clarify*,
erule-tac x=1 **in** *allE*, *simp*, *clarsimp*)
apply (*drule all2-impD*)
apply (*drule-tac a=tid2* **in** *all4-impD*, *assumption*, *clarify*)
apply (*drule-tac a=Q0* **in** *all-impD*)
apply (*clarsimp split*: *option.splits split-if-asm*
simp add: *pq-expand combine-queues-def fdisj-def map-add-def*)
apply (*erule-tac x=tid2* **in** *allE*, *erule disjE*, *simp*, *simp*)
apply (*erule-tac a=0* **in** *all4-impD*, (*rule conjI* | *simp*)+)
apply (*drule imp2D*)
apply (*frule* (*3*) *pq-expand2*(*1*), *clarsimp simp add*: *combine-perms-def*, *erule def-starE*)
apply (*frule* (*3*) *pq-expand2*(*2*), *clarsimp split*: *option.splits split-if-asm*
simp add: *combine-queues-def fdisj-break*)
apply (*clarsimp simp add*: *expand-fun-eq*)
apply (*rule exI*, *rule conjI*, *erule-tac tid=tid* **in** *gt-joinI*, *simp+*)
done

end

D.5 Deny-guarantee examples

theory *DGExamples*
imports *Main VHelper DGLogic*
begin

D.5.1 Useful derived rules

lemma *stable-cong*[*cong*]:
lt-sat x = *lt-sat y* ⟹ *stable x* = *stable y*
by (*simp add*: *stable-def*)

We define the notion of having full permission for a given program variable.

definition
fullperm :: *nat* ⇒ (*state*,′*a*::*res-algebra*) *permDG set*
where
fullperm v k ≡ (∀ *s s*′. *if s v* ≠ *s*′ *v then*
if ∀ *w*. *w*≠*v* ⟶ *s w* = *s*′ *w then k*(*s*,*s*′) = *H-top else is-deny* (*k*(*s*,*s*′))
else k(*s*,*s*′) = *H-zero*)

definition
a-Full :: *nat* ⇒ (*state*,′*a*::*res-algebra*) *lt-assn*
where
a-Full v ≡ *a-Perm* (*fullperm v*)

definition

a-FThread :: *nat* ⇒ (*state*,′*a*::*res-algebra*) *lt-assn* ⇒ (*state*,′*a*) *lt-assn*
where
a-FThread v Q ≡ *a-Full v* ∗∗ *a-Thread* (*λs*. *s v*) *Q*

lemma *fullperm-undefD*:
⟦*fullperm v k*; *fullperm v k*′⟧ ⟹ ¬ *H-def k k*′
apply (*simp add*: *fullperm-def H-def-fun-def*)
apply (*drule-tac a*=*s*(*v*:=*0*) **and** *b*=*s*(*v*:=*1*) **in** *all2D*)+
apply (*rule-tac x*=*s*(*v*:=*0*) **in** *exI*, *rule-tac x*=*s*(*v*:=*1*) **in** *exI*)
apply *clarsimp*
done

lemma *nsupp-Star*:
⟦ *nsupp v P*; *nsupp v Q* ⟧ ⟹ *nsupp v* (*P* ∗∗ *Q*)
by (*simp add*: *nsupp-def*)

lemma *stable-Perm*[*simp*]: *stable* (*a-Perm k*) **by** (*simp add*: *stable-def*)
lemma *stable-Full*[*simp*]: *stable* (*a-Full v*) **by** (*simp add*: *a-Full-def*)
lemma *nsupp-Full*[*simp*]: *nsupp w* (*a-Full v*) **by** (*simp add*: *a-Full-def nsupp-def*)

lemma *stable-Star2*:
⟦ *stable x*; *stable y*; *lt-sat* (*x* ∗∗ *y*) = *lt-sat z*⟧ ⟹ *stable z*
apply (*simp add*: *stable-def*, *erule subst*, *simp* (*no-asm-use*), *clarify*)
apply ((*erule allE*)+, *erule impE*, *erule conjI*, *erule* (*1*) *interfered-antimon*)
apply (*erule impE*, *erule conjI*, *rule interfered-antimon*, *subst star-comm*, *assumption*,
simp add: *H-ac*)
apply *fast*
done

lemma *stable-FThread*[*simp*]:
stable (*a-FThread v Q*)
apply (*clarsimp simp add*: *a-FThread-def stable-def a-Full-def interfered-def fullperm-def*)
apply (*drule-tac a*=*s* **and** *b*=*s*′ **in** *all2D*, *clarsimp simp add*: *is-guar-def is-deny-def*)
apply ((*erule disjE*, *clarsimp*)+, *metis*, *clarsimp*)
done

definition
c-PFork :: *var* ⇒ (*state*, ′*a*::*res-algebra*) *lt-assn* ⇒ (*state*, ′*a*) *lt-assn*
⇒ ′*a cmd* ⇒ ′*a cmd*
where
c-PFork v P Q C ≡ *c-Fork v* (*Abs-prec-assn P*) *Q C*

lemma *Rule-Fork2*:
⟦*lt-Hoare P0 C Q0*; *precise P0*; *nsupp v F*;
lt-sat (*a-FThread v Q0* ∗∗ *F*) ⊆ *lt-sat Q*;
stable P0; *stable F*; *stable Q*⟧
⟹ *lt-Hoare* (*a-Full v* ∗∗ *P0* ∗∗ *F*) (*c-PFork v P0 Q0 C*) *Q*
apply (*rule Rule-Conseq*, *rule-tac* [*3*] *order-refl*)
apply (*simp add*: *c-PFork-def*)

apply (*rule-tac F=a-Full v* ∗∗ *F* **in** *Rule-Fork*)
apply (*simp-all add*: *Abs-prec-assn-inverse prec-assn-def nsupp-Star*)
 apply (*clarsimp simp add*: *a-Full-def fullperm-def*)
 apply (*drule-tac a=s* **and** *b=s*(*v*:=*n*) **in** *all2D*)
 apply (*rule allowed-mon*, *clarsimp simp add*: *allowed-def*, *assumption*)
apply (*simp-all add*: *a-FThread-def Star-ac stable-star*)
done

lemma *Rule-Join2*:
 ⟦ *lt-sat* (*a-Full v* ∗∗ *F* ∗∗ *Q′*) ⊆ *lt-sat Q*;
  stable F; *stable Q′*⟧ ⟹
 lt-Hoare (*a-FThread v Q′* ∗∗ *F*) (*c-Join* (*λs*. *s v*)) *Q*
apply (*rule Rule-Conseq*)
apply (*rule-tac P=F* ∗∗ *a-Full v* **and** *Q=Q′* **in** *Rule-Join*)
apply (*simp-all add*: *Star-assoc*, *fold a-FThread-def*)
apply (*simp-all add*: *stable-star Star-ac*)
done

definition
 c-Assign :: *nat* ⇒ (*state* ⇒ *nat*) ⇒ (*′a*::*res-algebra*) *cmd*
where
 c-Assign v E ≡ *c-Atom* (*λs s′*. *s′* = *s*(*v*:=*E s*))

lemma *Rule-Assign*:
 ⟦*stable P*; *stable Q*;
  ∀ *s k tq*. *lt-sat P* (*s*, *k*, *tq*) ⟶ *lt-sat Q* (*s*(*v*:= *E s*), *k*, *tq*) ∧ *allowed k* (*s*, *s*(*v*:=*E s*)) ⟧
 ⟹ *lt-Hoare P* (*c-Assign v E*) *Q*
by (*simp add*: *c-Assign-def*, *erule* (*1*) *Rule-Atom*, *clarsimp*)

lemma *Rule-FalsePre*:
 ⟦¬ (∃*s k tq*. *lt-sat P* (*s*,*k*,*tq*)); *lt-wa C*⟧ ⟹ *lt-Hoare P C Q*
by (*simp add*: *lt-Hoare-def lt-has-spec-def*)

D.5.2 Parallel composition

Parallel composition can be encoded as forking a thread and then joining with it.

constdefs
 Parallel :: *var* ⇒ (*state*,*′a*::*res-algebra*) *lt-assn* ⇒ (*state*,*′a*) *lt-assn* ⇒
   ′a cmd ⇒ *′a cmd* ⇒ *′a cmd*
 Parallel v P1 Q1 C1 C2 ≡ *c-PFork v P1 Q1 C1* ;; *C2* ;; *c-Join* (*λs*. *s v*)

Using the proof rules for fork, join, and sequential composition, we can derive the following simpler proof rule for parallel composition.

lemma *Rule-Parallel*:
assumes *A1*: *lt-Hoare P1 C1 Q1*
and *A2*: *lt-Hoare P2 C2 Q2*
and *N*: *nsupp v P2 precise P1*
and *S*: *stable P1 stable Q1 stable P2 stable Q2*
shows *lt-Hoare*

```
    (P1 ** P2 ** a-Full v)
    (Parallel v P1 Q1 C1 C2)
    (Q1 ** Q2 ** a-Full v)
proof -
  have P1 ** P2 ** a-Full v ≃ a-Full v ** P1 ** P2
    by (simp add: lt-sat-eq-def Star-ac)
  also from A1 N S
  have lt-Hoare ... (c-PFork v P1 Q1 C1) (P2 ** a-FThread v Q1)
    by (erule-tac Rule-Fork2, (simp add: stable-star Star-ac)+)
  also from A2
  have lt-Hoare ... C2 (Q2 ** a-FThread v Q1)  by (rule Rule-Frame, simp)
  also have ... ≃ a-FThread v Q1 ** Q2         by (simp add: lt-sat-eq-def Star-ac)
  also from S
  have lt-Hoare ... (c-Join (λs. s v)) (Q1 ** Q2 ** a-Full v)
    by (rule-tac Rule-Join2, simp-all add: Star-ac)
  finally show ?thesis by (simp add: Parallel-def)
qed
```

D.5.3 An example proof

This section shows the proof of an earlier version of the paper's example.

Here is a half guarantee permission:

```
definition halfG :: (bool,myfrac) permTag where
  halfG ≡ Abs-permTag(False,H-half H-top)

lemma halfG0[simp]:
  Rep-permTag halfG = (False, H-half H-top)
apply (simp add: is-guar-def halfG-def)
apply (rule Abs-permTag-inverse)
apply (simp add: permTag-def H-half-def H-top-myfrac-def H-zero-myfrac-def
       zero-rat one-rat mult-rat le-rat eq-rat)
done

lemma halfG1[simp]:
  halfG ≠ H-zero
  halfG ≠ H-top
  is-guar halfG
  ¬is-deny halfG
apply (simp-all add: is-guar-def is-deny-def H-zero-permTag-def H-top-permTag-def)
apply (simp-all add: halfG-def Abs-permTag-inject permTag-def
       H-half-def H-top-myfrac-def H-zero-myfrac-def
       zero-rat one-rat mult-rat le-rat eq-rat)
done

lemma halfG2[simp]:
  H-def halfG halfG
  halfG ⊙ halfG = H-top
apply (simp-all add: H-star-permTag-def H-def-permTag-def
```

permTag-def-def permTag-star-def H-top-permTag-def)
apply (*simp add*: *H-half-def H-top-myfrac-def H-zero-myfrac-def*
zero-rat one-rat eq-rat le-rat mult-rat)
done

lemma *halfG3*[*simp*]:
halfG ⊙ (*halfG* ⊙ *y*) = *H-top* ⊙ *y*
by (*subst star-assoc* [*THEN sym*], *simp*)

constdefs
pmG :: *nat* ⇒ *nat set* ⇒ (*bool*,*myfrac*) *permTag* ⇒ (*state*,*myfrac*) *permDG set*
pmG v N j k ≡ (∀ *s s′*. *k*(*s*,*s′*) = (*if s* ≠ *s′* ∧ *s′ v* ∈ *N then j else H-zero*))

The predicate *pmG v N j k* says that *k* gives us permission *j* to access write a value in *N* to *v*.

lemma *precise-pmG*:
precise (*a-Perm* (*pmG x N j*))
apply (*clarsimp simp add*: *precise-def pmG-def*)
apply (*rule ext*, *clarsimp*, *metis*)
done

lemma *pmG-def2*:
pmG v N j k = (*k* = (λ(*s*,*s′*). *if s* ≠ *s′* ∧ *s′ v* ∈ *N then j else H-zero*))
by (*simp add*: *pmG-def expand-fun-eq*)

lemma *pmG-star1*: *H-def a b* ⟹
lt-sat (*a-Perm* (*pmG x N* (*a* ⊙ *b*)))
= *lt-sat* (*a-Perm* (*pmG x N a*) ∗∗ *a-Perm* (*pmG x N b*))
apply (*rule ext*, *clarsimp split*: *prod.splits cong*: *conj-cong simp add*: *pmG-def2 H-star-fun-def*)
apply (*simp add*: *H-def-fun-def fdisj-def expand-fun-eq*)
done

lemma *pmG-star2*: *M* ∩ *N* = {} ⟹
lt-sat (*a-Perm* (*pmG x* (*M* ∪ *N*) *a*))
= *lt-sat* (*a-Perm* (*pmG x M a*) ∗∗ *a-Perm* (*pmG x N a*))
apply (*rule ext*, *clarsimp split*: *prod.splits cong*: *conj-cong simp add*: *pmG-def2 H-star-fun-def*)
apply (*simp add*: *H-def-fun-def fdisj-def*)
apply (*rule iffI*, *rule conjI*, *fast*, *simp*, *rule ext*, *clarsimp*, *fast*)
apply (*clarify*, *simp* (*no-asm*) *add*: *expand-fun-eq*, *fast*)
done

lemma *nsupp-pmG*[*simp*]:
nsupp w (*a-Perm* (*pmG v M a*))
by (*simp add*: *pmG-def nsupp-def*)

lemma *nsupp-FThread*[*simp*]:
w ≠ *v* ⟹ *nsupp w* (*a-FThread v Q*)
by (*simp add*: *a-FThread-def*)
(*rule nsupp-Star*, *simp*, *simp add*: *nsupp-def*)

lemma *hack1*[*simp*]:
insert (*Suc 0*) (*UNIV* − {*Suc 0*, *2*}) = *UNIV* − {*2*}
apply (*rule ext*, *rename-tac x*)
apply (*simp only*: *Un-def Collect-def fun-diff-def*
bool-diff-def mem-def insert-def UNIV-def empty-def)
apply (*case-tac x*, *simp-all*, *rename-tac y*, *case-tac y*, *simp-all*)
done

And finally, here is the example proof:

lemma *Example1*:
lt-Hoare
(*a-Full t1* ∗∗ *a-Full t2* ∗∗ *a-Perm* (*pmG x UNIV H-top*))
(*c-PFork t1* (*a-Perm* (*pmG x* {*1*} *halfG*)) (*a-Perm* (*pmG x* {*1*} *halfG*))
(*c-Assign x* (λ*s*. *1*)) ;;
c-PFork t2 (*a-Perm* (*pmG x* {2} *halfG*)) (*a-Perm* (*pmG x* {2} *halfG*))
(*c-Assign x* (λ*s*. *2*)) ;;
c-Join (λ*s*. *s t1*) ;;
c-Assign x (λ*s*. *2*) ;;
c-Join (λ*s*. *s t2*))
(*a-Full t1* ∗∗ *a-Full t2* ∗∗ *a-Perm* (*pmG x UNIV H-top*) ∗∗ *a-Bool* (λ*s*. *s x* = *2*))
proof −
let *?G a* = *a-Perm* (*pmG x a halfG*)
let *?D* = *a-Perm* (*pmG x* (*UNIV*−{*1*,*2*}) *H-top*)
let *?C* = *a-Perm* (*pmG x* (*UNIV*−{2}) *H-top*)

have *1*: *lt-Hoare* (*?G* {*1*}) (*c-Assign x* (λ*s*. *1*)) (*?G* {*1*})
by (*rule Rule-Assign*, *simp-all add*: *pmG-def allowed-def*)
have *2*: *lt-Hoare* (*?G* {2}) (*c-Assign x* (λ*s*. *2*)) (*?G* {2})
by (*rule Rule-Assign*, *simp-all add*: *pmG-def allowed-def*)
have *S*: *stable* (*a-Bool* (λ*s*. *s x* = *2*) ∗∗ *?C*)
by (*clarsimp split*: *split-if-asm*
simp add: *pmG-def2 stable-def interfered-def is-guar-def*)
hence *3*: *lt-Hoare* (*?G* {2} ∗∗ *?C*) (*c-Assign x* (λ*s*. *2*))
(*?G* {2} ∗∗ *a-Bool* (λ*s*. *s x* = *2*) ∗∗ *?C*)
apply (*rule-tac Rule-Assign*, *simp-all add*: *stable-star*)
apply (*thin-tac ?P*, *clarsimp simp add*: *pmG-def fdisj-def*)
apply (*drule-tac a*=*s* **and** *b*=*s*(*x*:=*2*) **in** *all2D*)+
by (*rule allowed-mon*, *clarsimp simp add*: *allowed-def*, *assumption*)

show *?thesis*
proof (*cases t1* = *t2*)
assume *t1* = *t2*
with *1 2* **show** *?thesis*
apply (*rule-tac Rule-FalsePre*, *clarsimp simp add*: *a-Full-def*)
apply (*drule def-starD2*, *drule* (*1*) *fullperm-undefD*, *simp add*: *H-ac*)
by (*simp add*: *c-Assign-def c-PFork-def Abs-prec-assn-inverse*
prec-assn-def lt-Hoare-def precise-pmG)
next
assume *X*: *t1* ≠ *t2*

have *a-Full t1* ∗∗ *a-Full t2* ∗∗ *a-Perm* (*pmG x UNIV H-top*) ≃
a-Full t1 ∗∗ *?G* {*1*} ∗∗ *a-Full t2* ∗∗ *?G* {*1*} ∗∗ *?G* {2} ∗∗ *?G* {2} ∗∗ *?D*
by (*simp add*: *lt-sat-eq-def Star-ac*
pmG-star1 [*of halfG halfG*, *simplified*]
pmG-star2 [*of* {2} {*1*}, *simplified*]
pmG-star2 [*of* {*1*,2} (*UNIV*−{*1*,2}), *simplified*])
also from *1* **have**
lt-Hoare . . .
(*c-PFork t1* (*?G* {*1*}) (*?G* {*1*}) (*c-Assign x* (λ*s*. *1*)))
(*a-Full t2* ∗∗ *?G* {2} ∗∗ *a-FThread t1* (*?G* {*1*}) ∗∗ *?G* {*1*} ∗∗ *?G* {2} ∗∗ *?D*)
by (*erule-tac Rule-Fork2*,
simp-all add: *precise-pmG nsupp-Star stable-star Star-ac*)
also from *2 X* **have**
lt-Hoare . . .
(*c-PFork t2* (*?G* {2}) (*?G* {2}) (*c-Assign x* (λ*s*. 2)))
(*a-FThread t1* (*?G* {*1*}) ∗∗ *a-FThread t2* (*?G* {2}) ∗∗ *?G* {*1*} ∗∗ *?G* {2} ∗∗ *?D*)
by (*erule-tac Rule-Fork2*,
simp-all add: *precise-pmG nsupp-Star stable-star Star-ac*)
also have
lt-Hoare . . . (*c-Join* (λ*s*. *s t1*))
(*a-Full t1* ∗∗ *a-FThread t2* (*?G* {2}) ∗∗ *?G* {*1*} ∗∗ *?G* {*1*} ∗∗ *?G* {2} ∗∗ *?D*)
by (*rule-tac Rule-Join2*, *simp-all add*: *Star-ac stable-star*)
also from *3* **have**
lt-Hoare . . . (*c-Assign x* (λ*s*. 2))
(*a-FThread t2* (*?G* {2}) ∗∗ *?G* {2} ∗∗ *a-Full t1* ∗∗ *a-Bool* (λ*s*. *s x* = 2) ∗∗ *?C*)
apply (*rule-tac Rule-Conseq*,
erule-tac F=*a-Full t1* ∗∗ *a-FThread t2* (*?G* {2}) **in** *Rule-Frame*)
by (*simp-all add*: *Star-ac stable-star*
pmG-star1 [*of halfG halfG*, *simplified*]
pmG-star2 [*of* {*1*} (*UNIV*−{*1*,2}), *simplified*])
also from *S* **have**
lt-Hoare . . . (*c-Join* (λ*s*. *s t2*))
(*a-Bool* (λ*s*. *s x* = 2) ∗∗ *?G* {2} ∗∗ *?G* {2} ∗∗ *?C* ∗∗ *a-Full t1* ∗∗ *a-Full t2*)
by (*rule-tac Rule-Join2*, *simp-all add*: *stable-star*, *simp-all add*: *Star-ac*)
also have
. . . ≃ (*a-Full t1* ∗∗ *a-Full t2* ∗∗ *a-Perm* (*pmG x UNIV H-top*) ∗∗ *a-Bool* (λ*s*. *s x* = 2))
by (*simp add*: *lt-sat-eq-def Star-ac*
pmG-star1 [*of halfG halfG*, *simplified*]
pmG-star2 [*of* {2} (*UNIV*−{2}), *simplified*])
finally show *?thesis* **.**
qed
qed

end