Deny-Guarantee Reasoning

Mike Dodds, Xinyu Feng,
Matthew Parkinson & Viktor Vafeiadis

November 3rd, 2008

Deny-guarantee

- Rely-guarantee is the best current approach to reasoning about concurrency.
- However, it only deals with parallel composition, not fork / join.
- With deny-guarantee we can deal naturally with fork / join, by dynamically splitting interference
- Deny-guarantee is a powerful approach applicable beyond fork-join programs.

Deny-guarantee and the heap

This *is* a separation logic talk, in disguise

We depend crucially on the insights of abstract separation logic[1]

However, this isn't a talk about the heap, mostly.

- States have a fixed set of disjoint variables.
- Separate over interference only.

---

[1] See *Local Action and Abstract Separation Logic*, Calcagno, O'Hearn & Yang

Why do we need deny-guarantee?

Fork, join, and parallel composition

We can structure concurrency using parallel composition:

$$C_1 \parallel C_2$$

This executes C_1 and C_2 in parallel.

More natural to use fork and join

$$\text{fork } C_1 \qquad\qquad \text{join } C_1$$

Start C_1 and continue execution of the parent thread. Join C_1 later.

An example using fork and join

```
t1 := fork (x := 1) ;
t2 := fork (x := 2) ;
join t1;
x := 2;
join t2;
```

An example using fork and join

{true}

```
t1 := fork (x := 1) ;
t2 := fork (x := 2) ;
join t1;
x := 2;
join t2;
```

An example using fork and join

$\{\text{true}\}$

t1 := fork (x := 1) ;

t2 := fork (x := 2) ;

join t1;

x := 2;

join t2;

$\{x = 2\}$

A sketch proof

$\{\text{true}\}$

t1 := fork (x := 1) ;

$\{\text{Thread}(t1)\}$

t2 := fork (x := 2) ;

$\{\text{Thread}(t1) \wedge \text{Thread}(t2)\}$

join t1;

$\{\text{Thread}(t2)\}$

x := 2;

$\{\text{Thread}(t2) \wedge x = 2\}$

join t2;

$\{x = 2\}$

Rely-guarantee reasoning

Model concurrent interference as relations.

- Rely: what the environment can do.
- Guarantee: what the program can do.

Rely-guarantee judgements are of the form:

$$R, G \vdash \{P\}\, C \{Q\}$$

…where $R, G \subseteq \text{State} \times \text{State}$.

Parallel composition in RG

Reasoning about parallel composition is easy.

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

Interference is *statically scoped*: the same at the beginning and end of the parallel composition.

Static scoping won't work for fork / join!

Separation and interference

We want to split and join interference.

We already know how to dynamically split and join things.

separation logic!

Separation logic and the parallel rule

Consider the parallel rule in separation logic

$$\frac{\vdash_{\mathrm{SL}} \{P_1\}\, C_1\, \{Q_1\} \qquad \vdash_{\mathrm{SL}} \{P_2\}\, C_2\, \{Q_2\}}{\vdash_{\mathrm{SL}} \{P_1 * P_2\}\, C_1 \| C_2\, \{Q_1 * Q_2\}}$$

Separation allows us to naturally deal with dynamic scoping.

Conclusion: To deal with fork and join we need a star-operator for interference.

Developing deny-guarantee

First attempt

Take inspiration from the parallel rule.

$$\frac{\begin{array}{cc} R_1, G_1 \vdash \{P_1\}\ C_1\ \{Q_1\} & G_1 \subseteq R_2 \\ R_2, G_2 \vdash \{P_2\}\ C_2\ \{Q_2\} & G_2 \subseteq R_1 \end{array}}{R_1 \cap R_2, G_1 \cup G_2 \vdash \{P_1 \wedge P_2\}\ C_1 \| C_2\ \{Q_1 \wedge Q_2\}}$$

Define $*$ using union and intersection.

$$(R_1, G_1) * (R_2, G_2) = \begin{cases} (R_1 \cap R_2, G_1 \cup G_2) & G_1 \subseteq R_2 \ \& \ G_2 \subseteq R_1 \\ \text{undefined} & \text{otherwise} \end{cases}$$

Problem: we need cancellativity

Cancellative: for all x, y and z, if $x * y$ is defined and $x * y = x * z$, then $y = z$.

Non-cancellative operators lose information, breaking soundness for separation logic.

Union and intersection are not cancellative, so our first attempt fails.

Deny, not rely

Intuition: if we increase the context, proofs should be easier.

Rely-guarantee is the other way round!

We define a *deny*, saying what the environment can't do, instead of a rely.

$(D_1 \cup D_2, G_1 \cup G_2)$ is still not cancellative, but it's more uniform.

Second attempt

Disjoint union?

$$(D_1, G_1) * (D_2, G_2) = \begin{cases} (D_1 \uplus D_2, G_1 \uplus G_2) & G_1 \cap D_2 = \emptyset \\ & \& \; G_2 \cap D_1 = \emptyset \\ \text{undefined} & \text{otherwise} \end{cases}$$

Forbids any sharing between the two assertions.

For example, we can't prove:

$\{\text{true}\}$
t1 := fork (x:=2)
x := 2;
$\{x = 2\}$

Permissions and sharing

In concurrent separation logic we share locations using fractional permissions.

$$x \mapsto y \quad \leftrightarrow \quad x \overset{\frac{1}{2}}{\mapsto} y * x \overset{\frac{1}{2}}{\mapsto} y$$

Can give a thread $\frac{1}{2}$-permission on a location.

Associate actions with a fractional permission?

$$\text{State} \times \text{State} \rightarrow \text{Interval}[0, 1]$$

Third (successful) attempt

Define labelled permissions

$$\mathrm{PermDG} \stackrel{\mathbf{def}}{=} (\{\mathrm{guar}\} \times (0,1)) \ \uplus \ (\{\mathrm{deny}\} \times (0,1)) \ \uplus \ \{0\} \ \uplus \ \{1\}$$

Top and bottom elements 1 and 0.

Label fractions in $(0,1)$ with

- 'deny', for deny permissions.
- 'guar', for guarantee permissions.

Deny-guarantee permissions

Associate actions with labelled permissions.

$$pr : \text{State} \times \text{State} \rightarrow \text{PermDG}$$

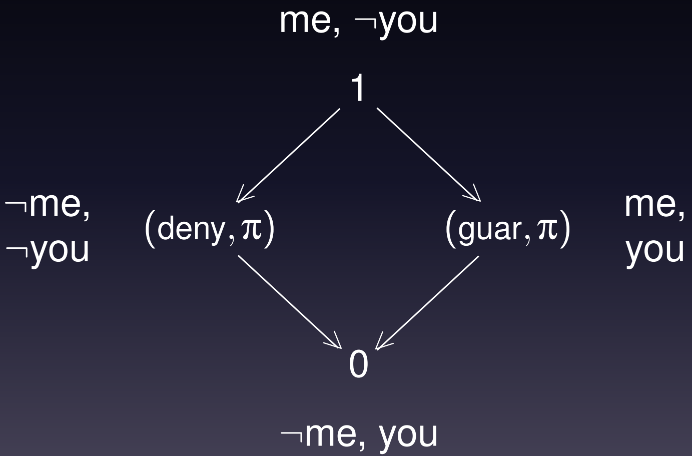
Deny-guarantee permissions can be split and joined in the way that we want.

From a relation to a function

Note that we have moved from sets of actions to a function on actions.

Justification: RG relation is a function from actions to 0 or 1.

Intuition: who can do something?



Adding permissions

0 and 1 behave conventionally.

$$0 \oplus x \overset{\text{def}}{=} x \oplus 0 \overset{\text{def}}{=} x$$

$$1 \oplus x \overset{\text{def}}{=} x \oplus 1 \overset{\text{def}}{=} \text{if } x = 0 \text{ then } 1 \text{ else undef}$$

Can't add a deny to a guar.

$$(\text{deny}, \pi) \oplus (\text{deny}, \pi') \overset{\text{def}}{=} \text{if } \pi + \pi' < 1 \text{ then } (\text{deny}, \pi + \pi') \text{ else if } \pi + \pi' = 1 \text{ then } 1 \text{ else } \bot$$

$$(\text{guar}, \pi) \oplus (\text{guar}, \pi') \overset{\text{def}}{=} \text{if } \pi + \pi' < 1 \text{ then } (\text{guar}, \pi + \pi') \text{ else if } \pi + \pi' = 1 \text{ then } 1 \text{ else } \bot$$

A star for interference

We can define a cancellative star for interference

For any action *a* and pair of permissions *pr* and *pr*, the star is defined so that

$$(pr * pr')(a) \; = \; pr(a) \oplus pr'(a)$$

Extracting rely-guarantee conditions

Extract rely-guarantee conditions from deny-guarantee permission *pr*

$$
\begin{aligned}
[\![pr]\!] \stackrel{\text{def}}{=} (&\{a \mid pr(a) = (\text{guar}, \_) \vee pr(a) = 0\}, \\
&\{a \mid pr(a) = (\text{guar}, \_) \vee pr(a) = 1\})
\end{aligned}
$$

Write $pr.R$ for extracted rely, and $pr.G$ for guarantee.

The logic of interference

Define an assertion language.

$$P, Q ::= B \mid pr \mid \text{false} \mid \text{Thread}(E, P) \mid P \rightarrow Q \mid P * Q \mid \exists X.\, P$$

Thread assertions record the expected post-condition for a running thread.

Judgements in the logic

Define judgements over a state σ, permission *pr*, and thread-queue γ

$$\sigma, pr, \gamma \models P$$

Permissions and states defined as before.

Thread-queue γ: TID $\rightarrow$ Stmts records the post-conditions for threads.

Assertion stability

Stable assertions are invariant under the permitted interference.

stable(P) states that if $\sigma, pr, \gamma \models P$ and $(\sigma, \sigma') \in pr.R$, then $\sigma', pr, \gamma \models P$.

We require that all assertions written in triples are stable.

Reasoning about fork and join

$$\frac{\{P_1\}\ C\ \{P_2\} \quad \text{Thread}(x, P_2) * P_3 \rightarrow P_4}{\{P_1 * P_3\}\ x := \textbf{fork}\ C\ \{P_4\}}\ \text{(fork)}$$

$$\frac{}{\{P * \text{Thread}(E, P')\}\ \textbf{join}\ E\ \{P * P'\}}\ \text{(join)}$$

(simplified from the rules in the paper)

Reasoning about assignment

$$\frac{P \rightarrow [E/x]P' \quad \text{allowed}([x := E], P)}{\{P\}\ x := E\ \{P'\}} \text{ (assn)}$$

Assignments have to be allowed by the permission.

allowed(A, P) where $A \subseteq \text{State} \times \text{State}$ asserts that if $\sigma, pr, \gamma \models P$ and $(\sigma, \sigma') \in A$ then $(\sigma, \sigma') \in pr.G$.

Reasoning with deny-guarantee

Proving the example

$\{\text{true}\}$

t1 := fork (x := 1;)

t2 := fork (x := 2;)

join t1;

x := 2;

join t2;

$\{x = 2\}$

Cutting up interference

Thread starts with permission 1 for every action.

First, define a small syntax for assertions:

$$[x : A \overset{p}{\rightsquigarrow} B] \overset{\textbf{def}}{=} \{((\sigma, \sigma[x \mapsto v]), p) \mid \sigma(x) \in A \wedge v \in B\}$$

Split into

$$[\mathrm{x} : \mathbb{Z} \overset{1}{\rightsquigarrow} \{1, 2\}] * K$$

Here K is permission 1 on all actions not defined in the first conjunct.

Cutting up interference

Split the permission again.

$$
\begin{aligned}
[\mathrm{x}: \mathbb{Z} \overset{1}{\leadsto} \{1,2\}] &\longrightarrow [\mathrm{x}: \mathbb{Z} \overset{1}{\leadsto} 1] * [\mathrm{x}: \mathbb{Z} \overset{1}{\leadsto} 2] \\
&\longrightarrow [\mathrm{x}: \mathbb{Z} \overset{\frac{1}{2}\mathbf{g}}{\leadsto} 1] * [\mathrm{x}: \mathbb{Z} \overset{\frac{1}{2}\mathbf{g}}{\leadsto} 1] \\
&\quad * [\mathrm{x}: \mathbb{Z} \overset{\frac{1}{2}\mathbf{g}}{\leadsto} 2] * [\mathrm{x}: \mathbb{Z} \overset{\frac{1}{2}\mathbf{g}}{\leadsto} 2]
\end{aligned}
$$

Define $G_1 = [\mathrm{x}: \mathbb{Z} \overset{\frac{1}{2}\mathbf{g}}{\leadsto} 1]$, and $G_2 = [\mathrm{x}: \mathbb{Z} \overset{\frac{1}{2}\mathbf{g}}{\leadsto} 2]$

Proving the example

Precondition for the example is as follows:

$\{G_1 * G_1 * G_2 * G_2 * K\}$

```
t1 := fork (x := 1;)
t2 := fork (x := 2;)
join t1;
x := 2;
join t2;
```

where $G_1 = [\mathrm{x} : \mathbb{Z} \overset{\frac{1}{2}\mathbf{g}}{\rightsquigarrow} 1]$ and $G_2 = [\mathrm{x} : \mathbb{Z} \overset{\frac{1}{2}\mathbf{g}}{\rightsquigarrow} 2]$

Proving thread specifications

$$\frac{P \rightarrow [E/x]P' \quad \text{allowed}([x := E], P)}{\{P\}\ x := E\ \{P'\}}\ \text{(assn)}$$

Apply the assignment rule:

$$\{[\mathrm{x} : \mathbb{Z} \overset{\frac{1}{2}\mathbf{g}}{\rightsquigarrow} 1]\}\quad \mathrm{x} := 1; \quad \{[\mathrm{x} : \mathbb{Z} \overset{\frac{1}{2}\mathbf{g}}{\rightsquigarrow} 1]\}$$

With a valid triple for $\mathrm{x} := 1$ we can apply the fork rule in the main program.

Proving the example

$$\frac{\{P_1\}\ C\ \{P_2\} \qquad \text{Thread}(x, P_2) * P_3 \rightarrow P_4}{\{P_1 * P_3\}\ x := \textbf{fork}\ C\ \{P_4\}}\ \text{(fork)}$$

Apply the fork rule:

$\{G_1 * G_1 * G_2 * G_2 * K\}$

t1 := fork (x := 1;)

$\{G_1 * G_2 * G_2 * K * \text{Thread}(\mathrm{t1}, G_1\}$

where $G_1 = [\mathrm{x} : \mathbb{Z} \overset{\frac{1}{2}\mathbf{g}}{\rightsquigarrow} 1]$ and $G_2 = [\mathrm{x} : \mathbb{Z} \overset{\frac{1}{2}\mathbf{g}}{\rightsquigarrow} 2]$

Proving the example

$$\frac{\{P_1\}\ C\ \{P_2\} \quad \mathsf{Thread}(x, P_2) * P_3 \rightarrow P_4}{\{P_1 * P_3\}\ x := \textbf{fork}\ C\ \{P_4\}}\ \text{(fork)}$$

Apply the fork rule again:

$\{G_1 * G_2 * G_2 * K * \mathsf{Thread}(\mathrm{t1}, G_1)\}$
    t2 := fork (x := 2;)
$\{G_1 * G_2 * K * \mathsf{Thread}(\mathrm{t1}, G_1) * \mathsf{Thread}(\mathrm{t2}, G_2)\}$

where $G_1 = [\mathrm{x} : \mathbb{Z} \overset{\frac{1}{2}\mathbf{g}}{\rightsquigarrow} 1]$ and $G_2 = [\mathrm{x} : \mathbb{Z} \overset{\frac{1}{2}\mathbf{g}}{\rightsquigarrow} 2]$

Proving the example

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

Apply the join rule

$$\{G_1 * G_2 * K * \mathsf{Thread}(\mathrm{t1}, G_1) * \mathsf{Thread}(\mathrm{t2}, G_2)\}$$
$$\text{join t1;}$$
$$\{G_1 * G_1 * G_2 * K * \mathsf{Thread}(\mathrm{t2}, G_2)\}$$

where $G_1 = [\mathrm{x} : \mathbb{Z} \overset{\frac{1}{2}\mathbf{g}}{\rightsquigarrow} 1]$ and $G_2 = [\mathrm{x} : \mathbb{Z} \overset{\frac{1}{2}\mathbf{g}}{\rightsquigarrow} 2]$

Proving the example

$$\frac{P \rightarrow [E/x]P' \quad \text{allowed}([x := E], P)}{\{P\}\; x := E\; \{P'\}} \text{ (assn)}$$

Apply the assignment rule

$$\{G_1 * G_1 * G_2 * K * \text{Thread}(\mathrm{t2}, G_2)\}$$
$$\mathrm{x} := 2;$$
$$\{G_1 * G_1 * G_2 * K * \text{Thread}(\mathrm{t2}, G_2) \wedge \mathrm{x} = 2\}$$

where $G_1 = [\mathrm{x} : \mathbb{Z} \overset{\frac{1}{2}\mathbf{g}}{\rightsquigarrow} 1]$ and $G_2 = [\mathrm{x} : \mathbb{Z} \overset{\frac{1}{2}\mathbf{g}}{\rightsquigarrow} 2]$

Proving the example

Recall that we require that every pre- and postcondition is *stable*

The following assertion *pr* is stable

$$\{G_1 * G_1 * G_2 * K * \mathrm{Thread}(\mathrm{t2}, G_2) \wedge \mathrm{x} = 2\}$$

…because $pr.R$ contains only actions of the form $(\sigma, \sigma[x \mapsto 2])$.

Everything else is excluded by permissions G_1, G_2 and K.

Proving the example

$\{G_1 * G_1 * G_2 * G_2 * K\}$

t1 := fork (x := 1;)

$\{G_1 * G_2 * G_2 * K * \mathsf{Thread}(\mathrm{t1}, G_1\}$

t2 := fork (x := 2;)

$\{G_1 * G_2 * K * \mathsf{Thread}(\mathrm{t1}, G_1) * \mathsf{Thread}(\mathrm{t2}, G_2)\}$

join t1;

$\{G_1 * G_1 * G_2 * K * \mathsf{Thread}(\mathrm{t2}, G_2)\}$

x := 2;

$\{G_1 * G_1 * G_2 * K * \mathsf{Thread}(\mathrm{t2}, G_2) \wedge \mathrm{x} = 2\}$

join t2;

$\{G_1 * G_1 * G_2 * G_2 * K \wedge \mathrm{x} = 2\}$

Correctness results

We have defined

- semantics for deny-guarantee judgements
- logical operational semantics
- machine operational semantics

We have proved, by hand and mechanically

- soundness of the proof system w.r.t the logical semantics
- correctness of erasure from logical to machine semantics

Deny-guarantee and rely-guarantee

We can encode rely-guarantee pairs into sets of PermDG permissions

$$[\![R, G]\!] \stackrel{\textbf{def}}{=} \{\langle R, G\rangle_f \mid f \in \text{Actions} \to (M \setminus \{0,1\})\}$$

$$\langle R, G\rangle_f \stackrel{\textbf{def}}{=} \lambda a. \begin{cases} (\text{guar}, f(a)) & a \in R \wedge a \in G \\ 0 & a \in R \wedge a \notin G \\ 1 & a \notin R \wedge a \in G \\ (\text{deny}, f(a)) & a \notin R \wedge a \notin G \end{cases}$$

Translating judgements

Translate rely-guarantee judgements into a set of triples in deny-guarantee

$$[\![R, G \vdash \{P\}\, C\, \{Q\}]\!] \stackrel{\textbf{def}}{=} \{\{P * pr\}\ C\ \{Q * pr\} \mid pr \in [\![R, G]\!]\}$$

Judgements still hold, as we can also translate proofs from rely-guarantee into deny-guarantee.

Proofs still hold in deny-guarantee

We can translate the RG parallel rule

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

…and the RG weakening rule.

$$\frac{[\![R_1, G_1 \vdash \{P\}\,C\,\{Q\}]\!] \quad R_2 \subseteq R_1 \quad G_1 \subseteq G_2}{[\![R_2, G_2 \vdash \{P\}\,C\,\{Q\}]\!]}$$

Further applications

Dealing with the heap

Deny-guarantee is mostly orthogonal to the heap.

Define permissions over heaps, rather than states:

$$\text{Heap} \times \text{Heap} \rightarrow \text{PermDG}$$

Otherwise deny-guarantee reasoning remains the same.

Singleton permissions

Alternative use for permissions in the heap:

$$\mathsf{SingleDG} \stackrel{\textbf{def}}{=} \mathsf{Val} \times \mathsf{Val} \rightarrow \mathsf{PermDG}$$

Define the heap with singleton permissions built in:

$$\mathsf{Heap} : \mathsf{Locs} \rightarrow \mathsf{Vals} \times \mathsf{SingleDG}$$

Permit an update of a location if it is allowed by its permission.

Locks in the Heap

Dynamically-allocated locks are difficult to reason about

Existing solutions use invariants, which prevent compositional reasoning

Deny-guarantee may give us a solution to this

Locks in the Heap

Associate locations with heap permissions?

$$\text{HeapDG} \stackrel{\textbf{def}}{=} \text{Locs} \rightarrow \text{Vals} \times \text{LockPerm}$$
$$\text{LockPerm} \stackrel{\textbf{def}}{=} \text{Vals} \times \text{HeapDG} \times \text{HeapDG} \rightarrow \text{PermDG}$$

Problems:

- Definition not well-founded!
- Self-referring locks.
- Recursive stability checking.

Conclusions

- We can define a cancellative star for interference.
- Deny-guarantee allows us to reason compositionally about fork and join.
- We expect that deny-guarantee will be applicable to other problems, e.g. locks in the heap.
- Meta-conclusion: Separation logic isn't just a logic of heaps.