Deny-Guarantee Reasoning

UNIVERSITY OF CAMBRIDGE
Computer Laboratory
Programming, Logic, and Semantics

Intuition

Rely-guarantee is the best approach to reasoning about concurrency.

However, it only deals with parallel composition, not fork and join.

We propose deny-guarantee, a new logic that deals naturally with fork and join by dynamically splitting interference

Fork and Join

Concurrency theorists have mostly dealt with parallel composition.

$$C_1 \parallel C_2$$

However, real programs use *fork* and *join*.

```
fork(C1);                    join(C1);
```

Start a thread C_1 with `fork`, and continue execution. Collect thread with `join`.

Simple fork-join example:

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

Program ensures that x=2 at termination, but this is difficult to prove.

Proving the Example

Suppose we allow interference to be split and joined.

We start with full permission. Full permission on a particular rewrite means no other thread can do it. Then we split it as follows.

$$\text{full} \quad \rightarrow \quad A_1 * A_2 * K$$

Here A_1 gives full permission to update x to 1, A_2 gives the same permission for x to 2, and K is the 'remainder' permission.

We split the full permission A_1 to give permission G_1, a partial permission to write 1 into x.

$$A_1 \quad \rightarrow \quad G_1 * G_1$$

Partial permissions mean other threads *may* be able to do the rewrite.

Then we can prove the program as follows.

$\{G_1 * G_1 * G_2 * G_2 * K\}$
`t1 := fork (x := 1);`
$\{G_1 * G_2 * G_2 * K * \text{Thread}(t1, G_1)\}$
`t2 := fork (x := 2);`
$\{G_1 * G_2 * K * \text{Thread}(t1, G_1) * \text{Thread}(t2, G_2)\}$
`join t1;`
$\{G_1 * G_1 * G_2 * K * \text{Thread}(t2, G_2)\}$
`x := 2;`
$\{G_1 * G_1 * G_2 * K * \text{Thread}(t2, G_2) \wedge x = 2\}$

Post-condition x=2 is stable because $G_1 * G_1 * K$ together give full permission on all actions, *except* writing 2 into x.

That is, the only permitted interference is writing 2 to x.

The Problem with Rely-guarantee

Rely-guarantee models interference as two relations over states.

- A rely R, the interference from the environment
- A guarantee G, the actions permitted for the program

Rely-guarantee rule for parallel composition:

$$\frac{\begin{array}{ll} R_1, G_1 \vdash \{P_1\}\, C_1\, \{Q_1\} & G_1 \subseteq R_2 \\ R_2, G_2 \vdash \{P_2\}\, C_2\, \{Q_2\} & G_2 \subseteq R_1 \end{array}}{R_1 \cap R_2,\ G_1 \cup G_2 \vdash \{P_1 \wedge P_2\}\, C_1 \parallel C_2\, \{Q_1 \wedge Q_2\}} \text{(par-rg)}$$

Note that the interference is *statically scoped* - the same before and after the parallel composition. This can't cope with fork-join!

Deny-guarantee

For deny-guarantee, we split interference *dynamically*.

Deny-guarantee defines unified permissions that combine both the rely and guarantee of

Define a set of permissions PermDG.

$$\text{PermDG} \quad = \quad (\{\text{guar}\} \times (0,1)) \uplus (\{\text{deny}\} \times (0,1)) \uplus \{0\} \uplus \{1\}$$

Permission *pr* map actions in State × State to permissions.

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

Permissions record interference. Given an action a:

- If $pr(a) = (\text{guar}, \pi)$ or 1, program can do action a
- If $pr(a) = (\text{guar}, \pi)$ or 0, environment can do action a
- A deny $pr(a) = (\text{deny}, \pi)$ records that action a cannot occur.

Reasoning About Fork and Join

We can define a separation logic star-operator over a *pr.*

Define a separation logic for programs with fork and join.

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

Assertions define both the state and the permitted interference.

Fork and join rules (simplified).

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

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

Deny-guarantee permissions allow us to prove our example.

People

- **Mike Dodds**, University of Cambridge
- **Xinyu Feng**, Toyota Technology Institute, Chicago
- **Matthew Parkinson**, University of Cambridge
- **Viktor Vafeiadis**, Microsoft Research Cambridge