Deny-Guarantee

(ongoing research)

Mike Dodds

RG + SL

RGSep: use rely-guarantee to extend the power of separation logic.

Deny-guarantee: use separation logic to extend the power of rely-guarantee.

Programming language

C ::= … | x := fork(C) | join(E) | …

Simple program

```
t1 := fork(x:=false);
t2 := fork(x:=true);
join(t1);
x := true;
if (x == false) error;
```

Rely-guarantee?

$$\frac{???}{R, G \vdash \{ P \}\ x := \mathrm{fork}(C)\ \{ Q \}}$$

Beyond rely-guarantee

Move the interference into the state:

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

Beyond rely-guarantee

Move the interference into the state:

$$\{ P \}\ C\ \{ Q \}$$

P and Q define:

- what the state holds; and
- what interference can occur

Fork Rule

$$\frac{\{\ P\ \}\ C\ \{\ Q\ \}}{\{\ P\ \}\ \ t := \mathrm{fork}(\ C\ )\ \ \{\ \mathrm{thr}(t,Q)\ \}}\ .$$

Join Rule

$$\frac{}{\{\ thr(t,Q)\ \}\quad join(\ t\ )\quad \{\ Q\ \}}.$$

Assignment

P stable

Q stable

allowed(P, Q)

$$\frac{P \Rightarrow Q\,[x{=}E]}{\{\,P\,\}\ x := E\ \{\,Q\,\}}\ .$$

Stability

In rely-guarantee, stability is defined with respect to some rely.

In deny-guarantee, we define assertion stability on its own:

$$\text{stable}(P) \quad \Leftrightarrow \quad \sigma \models P \wedge (\sigma,\sigma') \models \text{allowed}(P) \quad \Rightarrow \quad \sigma' \models P$$

How do we keep track of behaviour between threads?

Recall: RG Parallel Rule

$$\frac{\begin{array}{l} G_1 \Rightarrow R_2 \\ G_2 \Rightarrow R_1 \\ R_1, G_1 \vdash \{ P_1 \}\ C_1\ \{ Q_1 \} \\ R_2, G_2 \vdash \{ P_2 \}\ C_2\ \{ Q_2 \} \end{array}}{R_1 \wedge R_2, G_1 \vee G_2 \vdash \{ P_1 \wedge P_2 \}\ C_1 \parallel C_2\ \{Q_1 \wedge Q_2 \}}.$$

Interference as resource

We can't use union and intersection: they lose information.

Interference as resource

We can't use union and intersection: they lose information.

Separation logic to the rescue!

Interference as resource

We can't use union and intersection: they lose information.

Separation logic to the rescue!

DG introduces resources called *permissions*, tracking what behaviour is allowed to each thread.

Permissions

Permissions record interference, for example:

Permissions

Permissions record interference, for example:

$$[\, x{=}0 \rightsquigarrow x{=}1 \,]_{1/2}$$

Permissions

Permissions record interference, for example:

$$[\, x{=}0 \rightsquigarrow x{=}1 \,]_{1/2}$$

$$[\, x{=}n \rightsquigarrow x{=}n{+}1 \,]_{1}$$

Permissions

Permissions record interference, for example:

$[\, x{=}0 \rightsquigarrow x{=}1 \,]_{1/2}$

$[\, x{=}n \rightsquigarrow x{=}n{+}1 \,]_{1}$

$[\, x{=}1 \rightsquigarrow x{=}2 \,]_{1} \;*\; [\, x{=}3 \rightsquigarrow x{=}4 \,]_{1/4}$

Permissions

$[x=\_ \rightsquigarrow x=\text{true}]_{1}$: exclusive permission.

$[x=\_ \rightsquigarrow x=\text{true}]_{f}$ s.t. $f < 1$: non-exclusive permission.

Permission splitting

Permissions can be split:

$[x=\_ \rightsquigarrow x=\text{true}]_{1/2} * [x=\_ \rightsquigarrow x=\text{true}]_{1/2} \quad \Leftrightarrow \quad [x=\_ \rightsquigarrow x=\text{true}]_{1}$

Splitting an exclusive permission gives a non-exclusive permission.

Stability in deny-guarantee

Permissions ensure stability:

For example:

$$[x=0 \rightsquigarrow x=1]_{1/2} \wedge x = 1$$

is stable.

Encoding rely-guarantee

Previously:

$$R = \quad (\ x{=}0 \rightsquigarrow x{=}1\ ) \quad = \quad G$$

$$R, G \vdash \{\ x = 1\ \}\ \mathsf{skip}\ \{\ x = 1\ \}$$

Now write:

$$\{\ [x{=}0 \rightsquigarrow x{=}1]_{1/2} \ \wedge\ x = 1\ \}$$

$$\mathsf{skip}$$

$$\{\ [x{=}0 \rightsquigarrow x{=}1]_{1/2} \ \wedge\ x = 1\ \}$$

Simple program

```
t1 := fork(x:=false);
t2 := fork(x:=true);
join(t1);
x := true;
if (x == false) error;
```

Thread specifications

{ [x=_ ⇝ x=false]$_I$ }

x := false

Thread specifications

$\{\ [x=\_ \rightsquigarrow x=\mathit{false}]_I\ \}$

x := false

$\{\ [x=\_ \rightsquigarrow x=\mathit{false}]_I\ \}$

Thread specifications

{ [x=_ ⇝ x=false]$_1$ }

x := false

{ [x=_ ⇝ x=false]$_1$ }

{ [x=_ ⇝ x=true]$_{1/2}$ }

x := true

{ [x=_ ⇝ x=true]$_{1/2}$ }

Proving the program

{ [x=_⇝x=true]$_I$ * [x=_⇝x=false]$_I$ }

```
t1 := fork(x:=false);

t2 := fork(x:=true);

join(t1);

x := true;

if (x == false) error;
```

Proving the program

{ [x=_ ⇝ x=true]$_I$ * [x=_ ⇝ x=false]$_I$ }

t1 := fork(x:=false);

{ [x=_ ⇝ x=true]$_I$ * thr(t1, [x=_ ⇝ x=false]$_I$) }

t2 := fork(x:=true);

join(t1);

x := true;

if (x == false) error;

Proving the program

$\{ [x=\_ \rightsquigarrow x=true]_1 \ * \ [x=\_ \rightsquigarrow x=false]_1 \}$

t1 := fork(x:=false);

$\{ [x=\_ \rightsquigarrow x=true]_1 \ * \ thr(t1, [x=\_ \rightsquigarrow x=false]_1) \}$

t2 := fork(x:=true);

$\{ [x=\_ \rightsquigarrow x=true]_{1/2} * thr(t2, [x=\_ \rightsquigarrow x=true]_{1/2}) * thr(t1, [x=\_ \rightsquigarrow x=false]_1)\}$

join(t1);

x := true;

if (x == false) error;

Proving the program

$\{ [x=\_ \rightsquigarrow x=true]_1 \ * \ [x=\_ \rightsquigarrow x=false]_1 \}$

t1 := fork(x:=false);

$\{ [x=\_ \rightsquigarrow x=true]_1 \ * \ thr(t1, [x=\_ \rightsquigarrow x=false]_1) \}$

t2 := fork(x:=true);

$\{ [x=\_ \rightsquigarrow x=true]_{1/2} * thr(t2, [x=\_ \rightsquigarrow x=true]_{1/2}) * thr(t1, [x=\_ \rightsquigarrow x=false]_1) \}$

join(t1);

$\{ [x=\_ \rightsquigarrow x=true]_{1/2} * thr(t2, [x=\_ \rightsquigarrow x=true]_{1/2}) * [x=\_ \rightsquigarrow x=false]_1 \}$

x := true;

if (x == false) error;

Proving the program

$\{ [x=\_ \rightsquigarrow x=true]_1 \ * \ [x=\_ \rightsquigarrow x=false]_1 \}$

t1 := fork(x:=false);

$\{ [x=\_ \rightsquigarrow x=true]_1 \ * \ thr(t1, [x=\_ \rightsquigarrow x=false]_1) \}$

t2 := fork(x:=true);

$\{ [x=\_ \rightsquigarrow x=true]_{1/2} * thr(t2, [x=\_ \rightsquigarrow x=true]_{1/2}) * thr(t1, [x=\_ \rightsquigarrow x=false]_1) \}$

join(t1);

$\{ [x=\_ \rightsquigarrow x=true]_{1/2} * thr(t2, [x=\_ \rightsquigarrow x=true]_{1/2}) * [x=\_ \rightsquigarrow x=false]_1 \}$

x := true;

$\{ [x=\_ \rightsquigarrow x=true]_{1/2} * thr(t2, [x=true]_{1/2}) * [x=\_ \rightsquigarrow x=false]_1 \wedge \ x=true \}$

if (x == false) error;

References

- Dodds *et al.* Deny-guarantee Reasoning, ESOP'09.

- Dinsdale-Young *et al.* Concurrent Abstract Predicates, ECOOP'10.