Relational verification using product programs

César Kunz[1,2]

(César Kunz holds a Juan de la Cierva Fellowship, MICINN, Spain)

Joint work with Gilles Barthe[1] and Juan Manuel Crespo[1]

[1]IMDEA Software, Spain [2]Universidad Politécnica de Madrid

First PROMETIDOS Summer School, UCM, Madrid

Relational Program Reasoning

Captures the fact that two programs behave similarly or that the same program behaves similarly in two different executions.

- Compiler correctness
- Program continuity
- Non-interference

Relational Program Reasoning

Captures the fact that two programs behave similarly or that the same program behaves similarly in two different executions.

- Compiler correctness
- Program continuity
- Non-interference

Relational Program Reasoning

Captures the fact that two programs behave similarly or that the same program behaves similarly in two different executions.

- Compiler correctness
- Program continuity
- Non-interference

Relational Program Reasoning

Captures the fact that two programs behave similarly or that the same program behaves similarly in two different executions.

- Compiler correctness
- Program continuity
- Non-interference

Translation validation

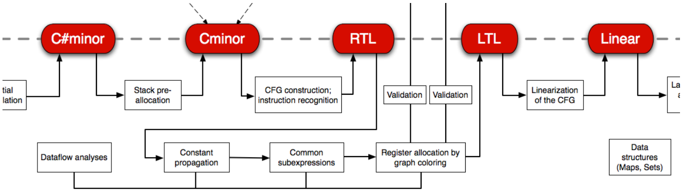


image fragment taken from the Compcert webpage.

Several techniques to provide trust over compiled code:

- Translation validation
- Certified compilation
- Certificate translation

Definition

Validating a program transformation consists on establishing that, assuming correct the result of the analysis, the original program and the transformed program are observationally equivalent.

Translation validation

For each compilation step: $c_i \longrightarrow c_{i+1}$

$$\begin{array}{llccc} \textbf{Execution} & c_i : & s_1 & \rightsquigarrow & s_1' \\ \textbf{Execution} & c_{i+1} : & s_2 & \rightsquigarrow & s_2' \end{array}$$

if the initial states s_1 and s_2 are observationally equivalent, then the final states s_1' and s_2' are observationally equivalent:

$$\forall x. s_1 x = s_2 x \Longrightarrow \forall x. s_1' x = s_2' x$$

Translation validation

For each compilation step: $c_i \longrightarrow c_{i+1}$

$$\textbf{Execution } c_i : \quad s_1 \rightsquigarrow s_1'$$

$$\textbf{Execution } c_{i+1} : \quad s_2 \rightsquigarrow s_2'$$

if the initial states s_1 and s_2 are observationally equivalent, then the final states s_1' and s_2' are observationally equivalent:

$$\forall x. s_1 x = s_2 x \Longrightarrow \forall x. s_1' x = s_2' x$$

Translation validation

Example: are these programs equivalent?

```
i := 0;                i := 0;
                       j := C;
while (i < N) do       while (i < N) do
   j := i*B+C;            x += j;
   x += j;                j += B;
   i++                    i++
```

lots of work on translation validation

- Pnueli, Zuck et al. Translation validation
- Sorin Lerner et al. Rhodium tool
- Benton's RHL

Translation validation

Example: are these programs equivalent?

```
$i := 0$;

while ($i < N$) do
    $j := i * B + C$;
    $x$ += $j$;
    $i$++
```

```
$i := 0$;
$j := C$;
while ($i < N$) do
    $x$ += $j$;
    $j$ += $B$;
    $i$++
```

lots of work on translation validation

- Pnueli, Zuck et al. Translation validation
- Sorin Lerner et al. Rhodium tool
- Benton's RHL

Program continuity

Analysis of program robustness under small input variations

[Gulwani et al.]

Given a program c:

$$\textbf{Execution } c : \quad s_1 \quad \rightsquigarrow \quad s_1'$$

$$\textbf{Execution } c : \quad s_2 \quad \rightsquigarrow \quad s_2'$$

if s_1 and s_2 differ in infinitesimal values, then s_1' and s_2' differ in infinitesimal values. For any ϵ, there is δ s.t.:

$$\forall x. \ \mid s_1 x - s_2 x \mid < \delta \Longrightarrow \forall x. \ \mid s_1' x - s_2' x \mid < \epsilon$$

Program continuity

Analysis of program robustness under small input variations

[Gulwani et al.]

Given a program c:

$$\textbf{Execution } c : \quad s_1 \quad \rightsquigarrow \quad s_1'$$

$$\textbf{Execution } c : \quad s_2 \quad \rightsquigarrow \quad s_2'$$

if s_1 and s_2 differ in infinitesimal values, then s_1' and s_2' differ in infinitesimal values. For any ϵ, there is δ s.t.:

$$\forall x. \ \mid s_1 x - s_2 x \mid < \delta \Longrightarrow \forall x. \ \mid s_1' x - s_2' x \mid < \epsilon$$

Program continuity

Example: is bubblesort continuous?

```
$i := 0$;
$j := N$;
while $(i \leq N)$ do
    $j := N$;
    while $(j > i)$ do
        if $(a[j-1] > a[j])$ then
            $x := a[j]$;
            $a[j] := a[j-1]$;
            $a[j-1] := x$
        $j := j-1$;
    $i := 1+i$
```

Non-interference

Basic idea

Equal low level inputs produce the same low outputs, regardless of the high level inputs.

Given a program c:

$$\textbf{Execution } c: \quad s_1 \quad \rightsquigarrow \quad s_1'$$

$$\textbf{Execution } c: \quad s_2 \quad \rightsquigarrow \quad s_2'$$

if s1 and s2 coincide in the public portion of the state, then s1' and s2' coincide in the public portion of the state.

$$\forall x \in L.\ s_1 x = s_2 x \Longrightarrow \forall x \in L.\ s_1' x = s_2' x$$

Non-interference

Basic idea

Equal low level inputs produce the same low outputs, regardless of the high level inputs.

Given a program c:

$$\textbf{Execution } c: \quad s_1 \quad \rightsquigarrow \quad s_1'$$

$$\textbf{Execution } c: \quad s_2 \quad \rightsquigarrow \quad s_2'$$

if s1 and s2 coincide in the public portion of the state, then s1' and s2' coincide in the public portion of the state.

$$\forall x \in L.\ s_1 x = s_2 x \Longrightarrow \forall x \in L.\ s_1' x = s_2' x$$

If it looks like we are talking about the same thing is in fact because we are.

Hyperproperties

All these relational properties are particular instances of Clarkson and Schneider *Hyperproperties*: Journal of Computer Security (2010)

Property: a set of execution traces

Hyperproperty: a set of sets of execution traces
(or a set of trace properties)
(or a property of sets of traces)

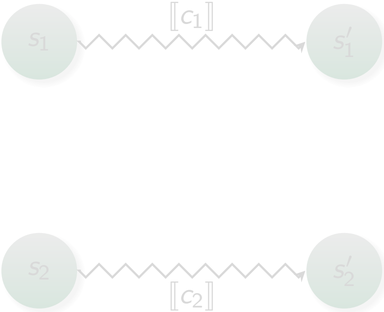
In particular k-safety properties: the bad thing never involves more than k traces.

program equivalence, program continuity and non-interference are 2-safety properties.

RELATIONAL JUDGEMENTS

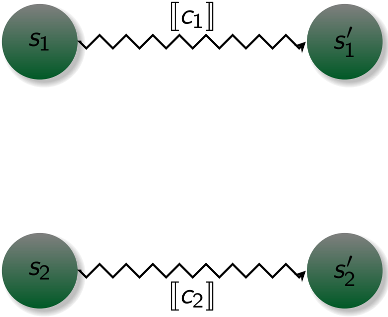
Relational judgement

$$\vDash \{\varphi\} c_1 \sim c_2 \{\psi\}$$



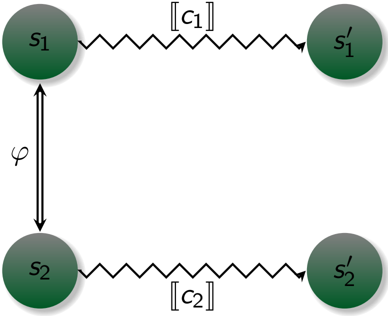
Relational judgement

$$\vDash \{\varphi\} c_1 \sim c_2 \{\psi\}$$



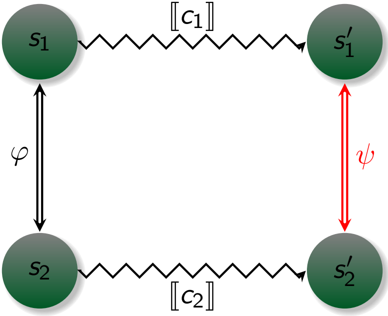
Relational judgement

$$\vDash \{\varphi\} c_1 \sim c_2 \{\psi\}$$



Relational judgement

$$\vDash \{\varphi\} c_1 \sim c_2 \{\psi\}$$



Give me a method to verify these relational judgements, and I'll have the power to prove the 2-safety properties we've been mentioning before.

Program transformations

$$\varphi \doteq \bigwedge_{x_i \in \mathbf{FV}(c_1)} x_i = x_i' \qquad \psi \doteq \bigwedge_{x_i \in \mathbf{FV}(c_1)} x_i = x_i'$$

Program continuity

$$\varphi \doteq \bigwedge_{x_i \in \mathbf{FV}(c_1)} |x_i - x_i'| < \delta \qquad \psi \doteq \bigwedge_{x_i \in \mathbf{FV}(c_1)} |x_i - x_i'| < \epsilon$$

Non-interference

$$\varphi \doteq \bigwedge_{x_i \in L} x_i = x_i' \qquad \psi \doteq \bigwedge_{x_i \in L} x_i = x_i'$$

A variety of logical methods have been proposed independently for each particular problem, as they were no connection between them.

Self-composition.

Logical verification of non-interference.

- typing and static analyses reject many secure programs
- logical frameworks extend the expressivity of declassification

Partial release (Sabelfeld and Myers)

$$\{(h \geq k)_{\langle 1\rangle} = (h \geq k)_{\langle 2\rangle} \wedge l_{\langle 1\rangle} = l_{\langle 2\rangle} \wedge k_{\langle 1\rangle} = k_{\langle 2\rangle}\}$$

$$\text{if } h \geq k \text{ then } h := h - 1; l := l + k; \text{ then skip}$$

$$\{l_{\langle 1\rangle} = l_{\langle 2\rangle} \wedge k_{\langle 1\rangle} = k_{\langle 2\rangle}\}$$

Self-composition.

Darvas et al. and Barthe et al. realized that non-interference can be reduced to a safety property:

- Darvas, Hähnle and Sands: *A theorem proving approach to analysis of secure information flow.* WITS'03
- Barthe, D'Argenio and Rezk: *Secure Information Flow by Self-composition.* CSFW'04

$$\models \{\varphi\}\, P; P' \,\{\psi\} \qquad \Longrightarrow \qquad \models \{\varphi\} P \sim P' \{\psi\}$$

Ok for straight line code, but cumbersome otherwise.

Nick Benton's Relational Hoare Logic (RHL)

RHL

logical characterization and verification of static analyses and program optimizations

- verification of non-interference
- program analysis and verification
- non-relational Hoare Logic verification

The problem, for Benton:

> *commonly, analysis are intensionally expressed, whereas program transformations are enabled by more extensional interpretations.*

Benton's RHL

"At first sight, this may seem frighteningly simple-minded, but it actually works rather nicely."—$N.Benton$

$$\frac{\vdash C \sim C' : \Phi \wedge (B\langle 1\rangle \wedge B'\langle 2\rangle) \Rightarrow \Phi \wedge (B\langle 1\rangle = B'\langle 2\rangle)}{\vdash \texttt{while}\ B\ \texttt{do}\ C \sim \texttt{while}\ B'\ \texttt{do}\ C' : \Phi \wedge (B\langle 1\rangle = B'\langle 2\rangle) \Rightarrow \Phi \wedge \texttt{not}(B\langle 1\rangle \vee B'\langle 2\rangle)}$$

$$\frac{\vdash C \sim C' : \Phi \wedge (B\langle 1\rangle \wedge B'\langle 2\rangle) \Rightarrow \Phi' \quad \vdash D \sim D' : \Phi \wedge \texttt{not}(B\langle 1\rangle \vee B'\langle 2\rangle) \Rightarrow \Phi'}{\vdash \texttt{if}\ B\ \texttt{then}\ C\ \texttt{else}\ D \sim \texttt{if}\ B'\ \texttt{then}\ C'\ \texttt{else}\ D' : \Phi \wedge (B\langle 1\rangle = B'\langle 2\rangle) \Rightarrow \Phi'}$$

$$\vdash X \texttt{ := } E \sim Y \texttt{ := } E' : \Phi[E\langle 1\rangle/X\langle 1\rangle, E'\langle 2\rangle/Y\langle 2\rangle] \Rightarrow \Phi$$

$$\frac{\vdash C \sim C' : \Phi \Rightarrow \Phi' \quad \vdash D \sim D' : \Phi' \Rightarrow \Phi''}{\vdash C\ \texttt{;}\ D \sim C'\ \texttt{;}\ D' : \Phi \Rightarrow \Phi''}$$

Benton's RHL

Succeeds to verify some code optimizations and secure information flow.

Verified slicing example

$$\{\mathsf{true}\}$$

```
I := 1;                  I := 1;
S := 0;
P := 1;                  P := 1
while I<N do (    ==>    while I<N do (
 S := S+I;
 P := P*I;                P := P*I;
 I := I+1;)               I := I+1;)
```

$$\{I_{\langle 1\rangle} = I_{\langle 2\rangle} \wedge P_{\langle 1\rangle} = P_{\langle 2\rangle}\}$$

with loop inv. $\{I_{\langle 1\rangle} = I_{\langle 2\rangle} \wedge P_{\langle 1\rangle} = P_{\langle 2\rangle}\}$

Benton's RHL

But RHL fails to verify a simple example when the number of iterations does not coincide:

Consider e.g. the following basic example:

```
i := 0;                    j := 1;
while (i ≤ N) do           while (j ≤ N) do
    x += i;                    y += j;
    i++                        j++
```

Moreover, as opposed to traditional verification, this relational method has no tool support

Other syntactic methods for deriving program equivalences

- Hongseok Yang's relational separation logic
  - it confines reasoning to structurally equivalent programs
  - not implemented (except Crespo's Coq formalization).

- Santiago Zanella et. al. probabilistic relational Hoare logic.
  - focused on (but not limited to) probabilistic programs
  - extension in progress, but far from a realistic programming language

Other syntactic methods for deriving program equivalences

- Hongseok Yang's relational separation logic
  - it confines reasoning to structurally equivalent programs
  - not implemented (except Crespo's Coq formalization).

- Santiago Zanella et. al. probabilistic relational Hoare logic.
  - focused on (but not limited to) probabilistic programs
  - extension in progress, but far from a realistic programming language

Combine the best of self-composition and RHL, and enable the use of existing tools.

Program products

Main Idea

Given two programs c_1 and c_2 and post and pre relations φ and ψ define a program $c_1 \times c_2$ and pre and postconditions $\bar{\varphi}$ and $\bar{\psi}$ such that:

$$\text{if} \models \{\bar{\varphi}\}\, c_1 \times c_2\, \{\bar{\psi}\} \text{ then} \models \{\varphi\} c_1 \sim c_2 \{\psi\}$$

Not really a new idea!

Self-composition does essentially this, by setting $c_1 \times c_2 = c_1; c_2$.

- \+ Does not require programs to be structurally equivalent.
- \+ (Relatively) Complete.
- − Impractical: not really amenable to program verification (lacks the synchronized loop invariants of RHL).

Program products

Main Idea

Given two programs c_1 and c_2 and post and pre relations φ and ψ define a program $c_1 \times c_2$ and pre and postconditions $\bar{\varphi}$ and $\bar{\psi}$ such that:

$$\text{if} \models \{\bar{\varphi}\}\, c_1 \times c_2\, \{\bar{\psi}\} \text{ then} \models \{\varphi\} c_1 \sim c_2 \{\psi\}$$

Not really a new idea!

Self-composition does essentially this, by setting $c_1 \times c_2 = c_1; c_2$.

- \+ Does not require programs to be structurally equivalent.
- \+ (Relatively) Complete.
- − Impractical: not really amenable to program verification (lacks the synchronized loop invariants of RHL).

Program products

Main Idea

Given two programs c_1 and c_2 and post and pre relations φ and ψ define a program $c_1 \times c_2$ and pre and postconditions $\bar{\varphi}$ and $\bar{\psi}$ such that:

$$\text{if} \vDash \{\bar{\varphi}\}\, c_1 \times c_2\, \{\bar{\psi}\} \text{ then} \vDash \{\varphi\} c_1 \sim c_2 \{\psi\}$$

Not really a new idea!

Self-composition does essentially this, by setting $c_1 \times c_2 = c_1; c_2$.

- \+ Does not require programs to be structurally equivalent.
- \+ (Relatively) Complete.
- − Impractical: not really amenable to program verification (lacks the synchronized loop invariants of RHL).

Program products

Main Idea

Given two programs c_1 and c_2 and post and pre relations φ and ψ define a program $c_1 \times c_2$ and pre and postconditions $\bar{\varphi}$ and $\bar{\psi}$ such that:

$$\text{if} \vDash \{\bar{\varphi}\}\, c_1 \times c_2\, \{\bar{\psi}\} \text{ then} \vDash \{\varphi\} c_1 \sim c_2 \{\psi\}$$

Not really a new idea!

Self-composition does essentially this, by setting $c_1 \times c_2 = c_1; c_2$.

+ Does not require programs to be structurally equivalent.
+ (Relatively) Complete.
– Impractical: not really amenable to program verification (lacks the synchronized loop invariants of RHL).

Consider again the following basic example:

```
$i := 0$;
while ($i \leq N$) do
    $x$ += $i$;
    $i$++
```

```
$j := 1$;
while ($j \leq N$) do
    $y$ += $j$;
    $j$++
```

Self-composition as product construction:

```
$i := 0$;
while ($i \leq N$) do
    $x$ += $i$;
    $i$++
```

×

```
$j := 1$;
while ($j \leq N$) do
    $y$ += $j$;
    $j$++
```

⟶

```
$i := 0$;
while ($i \leq N$) do
    $x$ += $i$;
    $i$++;
$j := 1$;
while ($j \leq N$) do
    $y$ += $j$;
    $j$++
```

The verification of the code on the right requires providing a couple of quadratic loop invariants.

Self-composition as product construction:

```
$i := 0$;
while ($i \leq N$) do
    $x$ += $i$;
    $i$++
```

×

```
$j := 1$;
while ($j \leq N$) do
    $y$ += $j$;
    $j$++
```

⟶

```
$i := 0$;
while ($i \leq N$) do
    $x$ += $i$;
    $i$++;
$j := 1$;
while ($j \leq N$) do
    $y$ += $j$;
    $j$++
```

The verification of the code on the right requires providing a couple of quadratic loop invariants.

Self-composition as product construction:

```
$i := 0$;
while ($i \le N$) do
    $x$ += $i$;
    $i$++
```

×

```
$j := 1$;
while ($j \le N$) do
    $y$ += $j$;
    $j$++
```

⟶

```
$i := 0$;
while ($i \le N$) do
    $x$ += $i$;
    $i$++;
$j := 1$;
while ($j \le N$) do
    $y$ += $j$;
    $j$++
```

The verification of the code on the right requires providing a couple of quadratic loop invariants.

A more clever product construction:

```
$i := 0$;
while ($i \le N$) do
    $x$ += $i$;
    $i$++
```

×

```
$j := 1$;
while ($j \le N$) do
    $y$ += $j$;
    $j$++
```

⟶

```
$i := 0$;
assert($i \le N$);
$x$ += $i$;  $i$++;
$j := 1$;
assert($i \le N \Leftrightarrow j \le N$);
while ($i \le N$) do
    $y$ += $j$;  $j$++;
    $x$ += $i$;  $i$++;
    assert($i \le N \Leftrightarrow j \le N$);
```

- The verification of the code on the right only needs the invariant $x = y \wedge i = j$.
- Relational verification is reduced to derivation of a product program followed by standard verification.

A more clever product construction:

```
$i := 0$;
while ($i \leq N$) do
    $x$ += $i$;
    $i$++
```

×

```
$j := 1$;
while ($j \leq N$) do
    $y$ += $j$;
    $j$++
```

$\longrightarrow$

```
$i := 0$;
assert($i \leq N$);
$x$ += $i$;  $i$++;
$j := 1$;
assert($i \leq N \Leftrightarrow j \leq N$);
while ($i \leq N$) do
    $y$ += $j$;  $j$++;
    $x$ += $i$;  $i$++;
    assert($i \leq N \Leftrightarrow j \leq N$);
```

- The verification of the code on the right only needs the invariant $x = y \wedge i = j$.
- Relational verification is reduced to derivation of a product program followed by standard verification.

Contribution:

- Syntactic method for constructing products that soundly abstract the behaviour of its constituents.
- Relational loop invariants become greatly simpler.
- A wide variety of application, e.g., correctness of advanced loop optimisations. Most of them verified with the Why verification framework, using SMT solvers and the Coq proof assistant.

Product construction rules

$$\frac{}{c_1 \times c_2 \to c_1; c_2} \qquad \frac{c_1 \times c_2 \to c \qquad c_1' \times c_2' \to c'}{c_1; c_1' \times c_2; c_2' \to c; c'}$$

$$\frac{c_1 \times c_2 \to c}{\mathsf{while}\ b_1\ \mathsf{do}\ c_1 \times \mathsf{while}\ b_2\ \mathsf{do}\ c_2 \to \mathsf{assert}(b_1 \Leftrightarrow b_2); \mathsf{while}\ b_1\ \mathsf{do}\ (c; \mathsf{assert}(b_1 \Leftrightarrow b_2))}$$

$$\frac{c_1 \times c_2 \to c \qquad c_1' \times c_2' \to c'}{\mathsf{if}\ b_1\ \mathsf{then}\ c_1\ \mathsf{else}\ c_1' \times \mathsf{if}\ b_2\ \mathsf{then}\ c_2\ \mathsf{else}\ c_2' \to \mathsf{assert}(b_1 = b_2); \mathsf{if}\ b_1\ \mathsf{then}\ c\ \mathsf{else}\ c'}$$

$$\frac{c_1 \succcurlyeq c_1' \qquad c_2 \succcurlyeq c_2' \qquad c_1' \times c_2' \to c}{c_1 \times c_2 \to c}$$

$$\ldots$$

Product construction rules

$$\frac{}{c_1 \times c_2 \rightarrow c_1; c_2} \qquad \frac{c_1 \times c_2 \rightarrow c \qquad c_1' \times c_2' \rightarrow c'}{c_1; c_1' \times c_2; c_2' \rightarrow c; c'}$$

$$\frac{c_1 \times c_2 \rightarrow c}{\text{while } b_1 \text{ do } c_1 \times \text{while } b_2 \text{ do } c_2 \rightarrow \text{assert}(b_1 \Leftrightarrow b_2); \text{while } b_1 \text{ do } (c; \text{assert}(b_1 \Leftrightarrow b_2))}$$

$$\frac{c_1 \times c_2 \rightarrow c \qquad c_1' \times c_2' \rightarrow c'}{\text{if } b_1 \text{ then } c_1 \text{ else } c_1' \times \text{if } b_2 \text{ then } c_2 \text{ else } c_2' \rightarrow \text{assert}(b_1 = b_2); \text{if } b_1 \text{ then } c \text{ else } c'}$$

$$\frac{c_1 \succcurlyeq c_1' \qquad c_2 \succcurlyeq c_2' \qquad c_1' \times c_2' \rightarrow c}{c_1 \times c_2 \rightarrow c}$$

. . .

Syntactic Reduction Rules

$$\frac{}{\text{if } b \text{ then } c_1 \text{ else } c_2 \succcurlyeq \text{assert}(b \Leftrightarrow \neg b'); \text{if } b' \text{ then } c_2 \text{ else } c_1}$$

$$\frac{}{\text{while } b \text{ do } c \succcurlyeq \text{assert}(b); c; \text{while } b \text{ do } c}$$

$$\frac{c_1 \succcurlyeq c_1' \qquad c_2 \succcurlyeq c_2'}{\text{if } b \text{ then } c_1 \text{ else } c_2 \succcurlyeq \text{if } b \text{ then } c_1' \text{ else } c_2'}$$

$$\frac{c \succcurlyeq c'}{\text{while } b \text{ do } c \succcurlyeq \text{while } b \text{ do } c'} \qquad \frac{c \succcurlyeq c' \qquad c' \succcurlyeq c''}{c \succcurlyeq c''} \qquad \frac{}{c \succcurlyeq c} \qquad \frac{c_1 \succcurlyeq c_1' \qquad c_2 \succcurlyeq c_2'}{c_1; c_2 \succcurlyeq c_1'; c_2'}$$

. . .

Main Results

- (Embedding of relational Hoare logic into standard one). For all derivations in relational hoare logic $\vdash \{\varphi\} c_1 \sim c_2 \{\psi\}$ there exist c such that $c_1 \times c_2 \rightarrow c$ and $\vdash \{\varphi\}\, c\, \{\psi\}$.
- (Soundness of the method). For all statements c_1 and c_2 and pre and post-relations φ and ψ, if $c_1 \times c_2 \rightarrow c$, $\models \{\varphi\}\, c\, \{\psi\}$ and φ is strong enough to ensure that c does not get stuck in any assert then $\models \{\varphi\} c_1 \sim c_2 \{\psi\}$.

THERE'S NO MAGIC HERE. PRODUCT CONSTRUCTION STILL DEMANDS A LOT OF EFFORT.

bubblesort continuity

Source code:

```
$i := 0$;
while ($i < N$) do
    $j := N-1$;
    while ($j > i$) do
        if ($a[j-1] > a[j]$) then
            $swap(a, j, j-1)$;
        $j--$
    $i++$
```

Program product (simplified):

```
$i := 0$; $i' := 0$;
while ($i < N$) do
    $j := N-1$; $j' := N-1$;
    while ($j > i$) do
        if ($a[j-1] > a[j]$) then
            $swap(a, j, j-1)$;
        if ($a'[j'-1] > a'[j']$) then
            $swap(a', j', j'-1)$;
        $j--$; $j'--$
    $i++$; $i'++$
```

$$\vDash \{\forall i.\ |a[i]-a'[i]| < \epsilon\}\ \mathbf{P}\ \{\forall i.\ |a[i]-a'[i]| < \epsilon\}$$

bubblesort continuity

Source code:

```
$i := 0$;
while ($i < N$) do
    $j := N-1$;
    while ($j > i$) do
        if ($a[j-1] > a[j]$) then
            $swap(a, j, j-1)$;
        $j--$
    $i++$
```

Program product (simplified):

```
$i := 0$;  $i' := 0$;
while ($i < N$) do
    $j := N-1$;  $j' := N-1$;
    while ($j > i$) do
        if ($a[j-1] > a[j]$) then
            $swap(a, j, j-1)$;
        if ($a'[j'-1] > a'[j']$) then
            $swap(a', j', j'-1)$;
        $j--$;  $j'--$
    $i++$;  $i'++$
```

$$\models \{\forall i.\ |a[i]-a'[i]| < \epsilon\}\ \mathbf{P}\ \{\forall i.\ \ |a[i]-a'[i]| < \epsilon\}$$

Non-interference

```
$i := 0$;
while $(i < N)$ do
    if $(ps[i].JoinInd)$ then
        $j := 0$;
        while $(j < M)$ do
            if $(ps[i].PID = es[j].EID)$ then
                $tab[i].employee := es[j]$;
                $tab[i].payroll := ps[i]$;
            $j$++;
    $i$++;
```

Pre

$es = es' \land \forall i : 0 \leq i < N : ps[i].PID = ps'[i].PID \land$
$ps[i].JoinInd = ps'[i].JoinInd \land (ps[i].JoinInd \Rightarrow ps[i].salary = ps'[i].salary)$

Post

$\forall i : 0 \leq i < N : ps[i].JoinInd \Rightarrow tab[i] = tab'[i]$

Non-interference

Program product verification:

```
$\{Pre : es = es' \wedge \forall i : 0 \leq i < N : ps[i].PID = ps'[i].PID \wedge$
    $ps[i].JoinInd = ps'[i].JoinInd \wedge (ps[i].JoinInd \Rightarrow ps[i].salary = ps'[i].salary)\}$
  $i := 0$; $i' := 0$; assert($i < N \Leftrightarrow i' < N$);
  while ($i < N$) do
    assert($ps[i].JoinInd \Leftrightarrow ps'[i'].JoinInd$);
    if ($ps[i].JoinInd$) then
      $j := 0$; $j' := 0$; assert($j < M \Leftrightarrow j' < M$);
      while ($j < M$) do
        assert($ps[i].PID = es[j].EID \Leftrightarrow ps'[i'].PID = es'[j'].EID$);
        if ($ps[i].PID = es[j].EID$) then
          $tab[i].employee := es[j]$; $tab'[i'].employee := es'[j]$;
          $tab[i].payroll := ps[i]$; $tab'[i'].payroll := ps'[i]$;
        $j{+}{+}$; $j'{+}{+}$; assert($i < N \Leftrightarrow i' < N$);
    $i{+}{+}$; $i'{+}{+}$; assert($i < N \Leftrightarrow i' < N$);
$\{Post : \forall i : 0 \leq i < N : ps[i].JoinInd \Rightarrow tab[i] = tab'[i]\}$
```

loop pipelining

Source program:

```
$i := 0$;
while ($i < N$) do
    $a[i]$++;  $b[i]$ += $a[i]$;
    $c[i]$ += $b[i]$;  $i$++
```

Transformed program:

```
$j := 0$;
$\bar{a}[0]$++;  $\bar{b}[0]$ += $\bar{a}[0]$;
$\bar{a}[1]$++;
while ($j < N-2$) do
    $\bar{a}[j+2]$++;
    $\bar{b}[j+1]$ += $\bar{a}[j+1]$;
    $\bar{c}[j]$ += $\bar{b}[j]$;  $j$++
$\bar{c}[j]$ += $\bar{b}[j]$;
$\bar{b}[j+1]$ += $\bar{a}[j+1]$;
$\bar{c}[j+1]$ += $\bar{b}[j+1]$
```

Product program:

```
$\{a = \bar{a} \wedge b = \bar{b} \wedge c = \bar{c}\}$
  $i := 0$;
  $j := 0$;
  assert($i < N$);
  $a[i]$++;  $b[i]$ += $a[i]$;
  $c[i]$ += $b[i]$;  $i$++
  $\bar{a}[0]$++;  $\bar{b}[0]$ += $\bar{a}[0]$;
  assert($i < N$);
  $a[i]$++;  $b[i]$ += $a[i]$;
  $c[i]$ += $b[i]$;  $i$++
  $\bar{a}[1]$++;
  assert($i < N \Leftrightarrow j < N-2$);
  while ($i < N$) do
      $a[i]$++;  $b[i]$ += $a[i]$;
      $c[i]$ += $b[i]$;  $i$++
      $\bar{a}[j+2]$++;
      $\bar{b}[j+1]$ += $\bar{a}[j+1]$;
      $\bar{c}[j]$ += $\bar{b}[j]$;  $j$++
      assert($i < N \Leftrightarrow j < N-2$);
  $\bar{c}[j]$ += $\bar{b}[j]$;
  $\bar{b}[j+1]$ += $\bar{a}[j+1]$;
  $\bar{c}[j+1]$ += $\bar{b}[j+1]$
$\{a = \bar{a} \wedge b = \bar{b} \wedge c = \bar{c}\}$
```

loop pipelining

Source program:

```
$i := 0$;
while $(i < N)$ do
    $a[i]$++;  $b[i]$ += $a[i]$;
    $c[i]$ += $b[i]$;  $i$++
```

Transformed program:

```
$j := 0$;
$\bar{a}[0]$++;  $\bar{b}[0]$ += $\bar{a}[0]$;
$\bar{a}[1]$++;
while $(j < N-2)$ do
    $\bar{a}[j+2]$++;
    $\bar{b}[j+1]$ += $\bar{a}[j+1]$;
    $\bar{c}[j]$ += $\bar{b}[j]$;  $j$++
$\bar{c}[j]$ += $\bar{b}[j]$;
$\bar{b}[j+1]$ += $\bar{a}[j+1]$;
$\bar{c}[j+1]$ += $\bar{b}[j+1]$
```

Product program:

```
$\{a = \bar{a} \wedge b = \bar{b} \wedge c = \bar{c}\}$
  $i := 0$;
  $j := 0$;
  assert$(i < N)$;
  $a[i]$++;  $b[i]$ += $a[i]$;
  $c[i]$ += $b[i]$;  $i$++
  $\bar{a}[0]$++;  $\bar{b}[0]$ += $\bar{a}[0]$;
  assert$(i < N)$;
  $a[i]$++;  $b[i]$ += $a[i]$;
  $c[i]$ += $b[i]$;  $i$++
  $\bar{a}[1]$++;
  assert$(i < N \Leftrightarrow j < N-2)$;
  while $(i < N)$ do
      $a[i]$++;  $b[i]$ += $a[i]$;
      $c[i]$ += $b[i]$;  $i$++
      $\bar{a}[j+2]$++;
      $\bar{b}[j+1]$ += $\bar{a}[j+1]$;
      $\bar{c}[j]$ += $\bar{b}[j]$;  $j$++
      assert$(i < N \Leftrightarrow j < N-2)$;
  $\bar{c}[j]$ += $\bar{b}[j]$;
  $\bar{b}[j+1]$ += $\bar{a}[j+1]$;
  $\bar{c}[j+1]$ += $\bar{b}[j+1]$
$\{a = \bar{a} \wedge b = \bar{b} \wedge c = \bar{c}\}$
```

static-caching (Annie Liu)

Source program:

```
$i_1 := 0$;
while ($i_1 \leq N - M$) do
  $s[i_1] := 0$;  $k_1 := 0$;
  while ($k_1 \leq M - 1$) do
    $l_1 := 0$;
    while ($l_1 \leq L - 1$) do
      $s[i_1]$+=$a[i_1 + k_1, l_1]$;  $l_1$++;
    $k_1$++;
  $i_1$++
```

Transformed program:

```
$t[0] := 0$;  $k_2 := 0$;
while ($k_2 \leq M - 1$) do
  $b[k_2] := 0$;  $l_2 := 0$;
  while ($l_2 \leq L - 1$) do
    $b[k_2]$ += $a[k_2, l_2]$;  $l_2$++;
  $t[0]$ += $b[k_2]$;  $k_2$++;
$i_2 := 1$;
while ($i_2 \leq N - M$) do
  $b[i_2 + M - 1] := 0$;  $l_2 := 0$;
  while ($l_2 \leq L - 1$) do
    $b[i_2 + M - 1]$+=$a[i_2 + M - 1, l_2]$;  $l_2$++;
  $z := b[i_2 + M - 1] - b[i_2 - 1]$;
  $t[i_2] := t[i_2 - 1] + z$;  $i_2$++
```

static-caching (Annie Liu)

```
{true}
$i_1 := 0$; assert($i_1 \le N-M$); $s[i_1] := 0$; $k_1 := 0$; $t[0] := 0$; $k_2 := 0$;
assert($k_1 \le M-1 \Leftrightarrow k_2 \le M-1$);
while ($k_1 \le M-1$) {$Inv_1$} do
    $l_1 := 0$; $b[k_2] := 0$; $l_2 := 0$; assert($l_1 \le L-1 \Leftrightarrow l_2 \le L-1$);
    while ($l_1 \le L-1$) {$Inv_2$} do
        $s[i_1]$+=$a[i_1+k_1, l_1]$; $l_1$++; $b[k_2]$ += $a[k_2, l_2]$; $l_2$++;
        assert($l_1 \le L-1 \Leftrightarrow l_2 \le L-1$);
    $k_1$++; $t[0]$ += $b[k_2]$; $k_2$++; assert($k_1 \le M-1 \Leftrightarrow k_2 \le M-1$);
$i_1$++; $i_2 := 1$; assert($i_1 \le N-M \Leftrightarrow i_2 \le N-M$);
while ($i_1 \le N-M$) {$Inv_3$} do
    $b[i_2+M-1] := 0$; $l_2 := 0$;
    while ($l_2 \le L-1$) {$Inv_4$} do
        $b[i_2+M-1]$+=$a[i_2+M-1, l_2]$; $l_2$++;
    $z := b[i_2+M-1] - b[i_2-1]$; $t[i_2] := t[i_2-1]+z$; $i_2$++;
    $s[i_1] := 0$; $k_1 := 0$;
    while ($k_1 \le M-1$) {$Inv_5$} do
        $l_1 := 0$;
        while ($l_1 \le L-1$) {$Inv_6$} do
            $s[i_1]$+=$a[i_1+k_1, l_1]$; $l_1$++;
        $k_1$++;
    $i_1$++
    assert($i_1 \le N-M \Leftrightarrow i_2 \le N-M$);
{$\forall\, i \in [0, N-M].\ s[i] = t[i]$}
```

$$Inv_2 \doteq i_1{=}0 \wedge k_1{=}k_2 \wedge l_1{=}l_2 \wedge k_1{\le}M \wedge l_1{\le}L \wedge s[i_1] = t[0]+b[k_1] = \textstyle\sum_{k'=0}^{k_1-1} b[k'] + b[k_1] \wedge$$

$$\forall\, k' \in [0, k_1).\ b[k'] = \textstyle\sum_{l'=0}^{L-1} a[k', l'] \wedge b[k_1] = \textstyle\sum_{l'=0}^{l_1-1} a[k_1, l']$$

$$Inv_3 \doteq i_1 = i_2 \wedge i_1 \le N-M+1 \wedge \forall\, i' \in [0, i_1) \Rightarrow s[i'] = t[i'] = \textstyle\sum_{k'=0}^{M-1} b[k'+i'] \wedge \forall\, i' \in [0, i_1+M-1).\ b[i'] = \textstyle\sum_{l'=0}^{L-1} a[i', l']$$

$$Inv_4 \doteq Inv_3 \wedge k_1 \le M \wedge l_2 \le L \wedge b[i_2+M-1] = \textstyle\sum_{l'=0}^{l_2-1} a[i_2+M-1, l'] \wedge s[i_1] = \textstyle\sum_{k'=0}^{k_1-1} b[k'+i_1]$$

$$Inv_6 \doteq Inv_3 \wedge k_1 \le M \wedge l_1 \le L \wedge b[i_2+M-1] = \textstyle\sum_{l'=0}^{L-1} a[i_2+M-1, l'] \wedge s[i_1] = \textstyle\sum_{k'=0}^{k_1-1} b[k'+i_1] + \textstyle\sum_{l'=0}^{l_1-1} a[i_1+k_1, l']$$

Overview of Why Verification

Optimisation	Proof Obligations	Automatically Discharged
Strength Reduction	9	9
Loop Reversal	13	13
Loop Interchange	41	38
Loop Alignment	53	53
Loop Pipelining	77	77
Static Caching	188	165

Open Research Topics

- Advanced heuristics for the construction of program products.
- Tool for interactive (i.e. tactic based) construction of products.
- Extension of the method for *k*-safety properties.
- Separation on heap manipulating programs (there are solutions to this, as relocatable programs, but never implemented).
- Program separation and Hongseok Yang's Relational Separation logics. (there are discussions with Bart Jacobs towards the *Multi*VeriFast tool)
- Relational program synthesis.

Conclusion

- A technique to reduce relational program verification (2-safety properties) into standard ones.
- This technique is usually constrained to structurally similar programs or requires full-blown functional verification.
- We generalized a marriage of RHL with self-composition coping with non structurally equivalent programs.
- Enables using existing program verification tools for such verification problems
- Requires ingenious effort to construct the program product
- We have illustrated the usefulness of our approach by using it to validate advanced loop optimization.