Automating Relational Program Verification

Hiroshi Unno (University of Tsukuba, Japan)

Relational Program Verification

- Verification of properties that relate **multiple executions** of one or more programs
- Clarkson and Schneider formalized such properties as **sets of sets of program traces** and coined the term **hyperproperties** [CSF 2008]
  - $\boldsymbol{k}$**-safety** is a notable subclass defined as hyperproperties that can be refuted by observing k finite traces (i.e., the bad thing never involves more than k traces)
- An important trend in formal methods with wide applications including **security**

[CSF 2008] Clarkson, Schneider. Hyperproperties.

Example Hyperproperties on Multiple Programs

Variants of Program Equivalence

- Functional (i.e., input-output) equivalence
  - Termination-insensitive: $f =_{DTI} g \triangleq \forall x, y_1, y_2. (f(x) \Downarrow y_1) \wedge (g(x) \Downarrow y_2) \Longrightarrow y_1 = y_2$
  - Termination-sensitive:
    $f =_{DTS} g \triangleq (f =_{DTI} g) \wedge \forall x. \big((f(x) \Uparrow) \Longrightarrow \neg\exists y. (g(x) \Downarrow y)\big) \wedge \forall x. \big((g(x) \Uparrow) \Longrightarrow \neg\exists y. (f(x) \Downarrow y)\big)$
  - Non-det. & Termination-*sensitive*: $f =_{NdTS} g \triangleq \forall x. \{y \mid f(x) \Downarrow y\} = \{y \mid g(x) \Downarrow y\}$
  - Probabilistic & Termination-*sensitive*: $f =_{PrTS} g \triangleq \forall x, y. \Pr[f(x) \Downarrow y] = \Pr[g(x) \Downarrow y]$
- Trace equivalence: $p =_{Tr} q \triangleq Tr(p) = Tr(q)$
- Bisimilarity: $p \sim_{bis} q \triangleq$ there is a strong bisimulation R such that $(p, q) \in R$
- Observational equivalence: $p =_{Obs} q \triangleq \forall C, y. (C[p] \Downarrow y) \Longleftrightarrow (C[q] \Downarrow y)$
  - Captures non-trivial interactions between contexts C and higher-order, object-oriented, and effectful (e.g., non-det., probabilistic, stateful, exception-raising, …) programs
  - In security applications, attackers' capabilities are reflected in the definition of contexts C

An execution of $f(x)$ terminates and returns y_1

$f(x)$ has a diverging execution

The set of finite and infinite execution traces of q

Variants of Program Refinement

Useful to transfer properties and proofs!

- Functional (i.e., input-output) refinement:
  - Termination-insensitive: $f \leq_{TI} g \triangleq \forall x, y. (f(x) \Downarrow y) \Longrightarrow (g(x) \Downarrow y)$
    - If $f \leq_{TI} g$, then $\vDash \{Pre\}\ g\ \{Post\}$ implies $\vDash \{Pre\}\ f\ \{Post\}$
  - Termination-sensitive: $f \leq_{TS} g \triangleq f \leq_{TI} g \land \forall x. (f(x) \Uparrow) \Longrightarrow (g(x) \Uparrow)$
    - If $f \leq_{TS} g$, then $\vDash [Pre]\ g\ [Post]$ implies $\vDash [Pre]\ f\ [Post]$ (i.e., termination is also transferred)
- Trace refinement: $p \leq_{Tr} q \triangleq Tr(p) \subseteq Tr(q)$
  - If $p \leq_{Tr} q$, then trace properties of q can be transferred to p
- Similarity: $p \leq_{sim} q \triangleq$ there is a strong simulation R such that $(p, q) \in R$
  - If $p \leq_{sim} q$, then trace (but branching-time) properties of q can be migrated to p
  - If $p \sim_{bis} q$, then branching-time (but hyper-) properties of q can be migrated to p

Program Refinement as *Generalized* Model Checking

- Program refinement verification $\models p \leq q$ generalizes ordinary model checking $p \models \phi$
  - **A specification of p is given as a program q** instead of a logical formula ϕ
  - q can encode the given ϕ (if the programming language is expressive enough)
  - q can be **a reference implementation** (cf. seL4 Project) or **an abstract model** represented as a highly non-deterministic program
- This motivates me to investigate entailment checking problems $\psi_1 \models \psi_2$ in a first-order fixpoint logic modulo theories we call μCLP [CAV 2017, LICS 2018, POPL 2023]
- I will come back to this point: **relational verification via entailment checking in μCLP**

[CAV 2017] Unno et al. Automating Induction for Solving Horn Clauses.
[LICS 2018] Nanjo et al. A Fixpoint Logic and Dependent Effects for Temporal Property Verification.
[POPL 2023] Unno et al. Modular Primal-Dual Fixpoint Logic Solving for Temporal Verification.

Example Hyperproperties on Single Program

Information Flow Confidentiality & Integrity [JSAC 2003]

- Deterministic programs
  - Termination-insensitive non-interference $\in$ **2-*safety*** $\subset$ ***Hypersafety*** :
    $TINI(f) \triangleq \forall h_1, h_2, x, y_1, y_2. (f(h_1, x) \Downarrow y_1) \wedge (f(h_2, x) \Downarrow y_2) \Longrightarrow y_1 = y_2$
    A well-studied security policy formalizing the absence of information leakage
  - Termination-sensitive non-interference:
    $TSNI(f) \triangleq TINI(f) \wedge \forall h_1, h_2, x. (f(h_1, x) \Uparrow) \Longrightarrow \neg\exists y. (f(h_2, x) \Downarrow y)$
  - Timing attack resilience
- Non-deterministic / concurrent programs
  - Observational determinism: $OD(f) \triangleq \forall h_1, h_2, x, y_1, y_2. (f(h_1, x) \Downarrow y_1) \wedge (f(h_2, x) \Downarrow y_2) \Longrightarrow y_1 = y_2$
  - Possibilistic non-interference
    - TI *Generalized* NI: $TIGNI(f) \triangleq \forall h_1, h_2, x, y. (f(h_1, x) \Downarrow y) \Longrightarrow (f(h_2, x) \Uparrow) \vee (f(h_2, x) \Downarrow y)$
    - TS *Generalized* NI: $TSGNI(f) \triangleq \forall h_1, h_2, x, y. (f(h_1, x) \Downarrow y) \Longrightarrow (f(h_2, x) \Downarrow y)$
  - Bisimulation-based non-interface

[JSAC 2003] Sabelfeld, Myers. Language-based Information-flow Security.

Availability

- Denial of Service (DoS) attack resilience
  - Property that “the *average* response time over all executions is bounded” is *relational* while “the *maximum* response time is bounded” is *non-relational*

Algorithm Analysis

- Robustness and sensitivity [CACM 2012, ICFP 2010]
  - Continuity:

$$Cont(f) \triangleq \begin{array}{c} \forall x_1, x_2. \forall \epsilon > 0. \exists \delta > 0. d(x_1, x_2) < \delta \Longrightarrow \\ \forall y_1, y_2. (f(x_1) \Downarrow y_1) \wedge (f(x_2) \Downarrow y_2) \Longrightarrow d(y_1, y_2) < \epsilon \end{array}$$

  - Lipschitz Continuity:

$$LC(f, c) \triangleq \forall x_1, x_2, y_1, y_2. (f(x_1) \Downarrow y_1) \wedge (f(x_2) \Downarrow y_2) \Longrightarrow \frac{d(y_1, y_2)}{d(x_1, x_2)} < c$$

[CACM 2012] Chaudhuri et al. Continuity and Robustness of Programs.
[ICFP 2010] Reed, Pierce. Distance Makes the Types Grow Stronger.

Specific Use Cases (1/2)

- Regression verification [STVR 2013]
  - Check refinement $T \leq S$ for different versions T, S of programs where T is obtained from S by refactoring, bug fixes, or enhancements
  - Goal is to verify the absence of a **software regression** which is a bug of T introduced by the modifications to S
- Translation validation [TACAS 1998]
  - Check refinement $T \leq S$ for the source S and target T programs obtained by compilation or optimization

[STVR 2013] Godlin, Strichman. Regression verification: proving the equivalence of similar programs.
[TACAS 1998] Pnueli et al. Translation Validation.

Specific Use Cases (2/2)

Verification of an implementation of an **abstract data type** with **algebraic specs.**

- Arithmetic operations with:
  equivalence, associativity, commutativity, distributivity, idempotency, monotonicity invertibility, symmetry, transitivity, …
  (see the right table from [CAV 2017])
- List operations with algebraic specs. like:
  append (take xs n) (drop xs n) = xs
  Try out a web interface of our relational verifier from http://lfp.dip.jp/rcaml/

ID	specification	kind	features	result	time (sec.)
1	$\texttt{mult}\ x\ y + a = \texttt{mult\_acc}\ x\ y\ a$	equiv	P	✓	0.378
2	$\texttt{mult}\ x\ y = \texttt{mult\_acc}\ x\ y\ 0$	equiv	P	✓†	0.803
3	$\texttt{mult}\ (1+x)\ y = y + \texttt{mult}\ x\ y$	equiv	P	✓	0.403
4	$y \geq 0 \Rightarrow \texttt{mult}\ x\ (1+y) = x + \texttt{mult}\ x\ y$	equiv	P	✓	0.426
5	$\texttt{mult}\ x\ y = \texttt{mult}\ y\ x$	comm	P	✓‡	0.389
6	$\texttt{mult}\ (x+y)\ z = \texttt{mult}\ x\ z + \texttt{mult}\ y\ z$	dist	P	✓	1.964
7	$\texttt{mult}\ x\ (y+z) = \texttt{mult}\ x\ y + \texttt{mult}\ x\ z$	dist	P	✓	4.360
8	$\texttt{mult}\ (\texttt{mult}\ x\ y)\ z = \texttt{mult}\ x\ (\texttt{mult}\ y\ z)$	assoc	P	✗	n/a
9	$0 \leq x_1 \leq x_2 \land 0 \leq y_1 \leq y_2 \Rightarrow \texttt{mult}\ x_1\ y_1 \leq \texttt{mult}\ x_2\ y_2$	mono	P	✓	0.416
10	$\texttt{sum}\ x + a = \texttt{sum\_acc}\ x\ a$	equiv		✓	0.576
11	$\texttt{sum}\ x = x + \texttt{sum}\ (x-1)$	equiv		✓	0.452
12	$x \leq y \Rightarrow \texttt{sum}\ x \leq \texttt{sum}\ y$	mono		✓	0.593
13	$x \geq 0 \Rightarrow \texttt{sum}\ x = \texttt{sum\_down}\ 0\ x$	equiv	P	✓	0.444
14	$x < 0 \Rightarrow \texttt{sum}\ x = \texttt{sum\_up}\ x\ 0$	equiv	P	✓	0.530
15	$\texttt{sum\_down}\ x\ y = \texttt{sum\_up}\ x\ y$	equiv	P	✗	n/a
16	$\texttt{sum}\ x = \texttt{apply}\ \texttt{sum}\ x$	equiv	H	✓	0.430
17	$\texttt{mult}\ x\ y = \texttt{apply2}\ \texttt{mult}\ x\ y$	equiv	H, P	✓	0.416
18	$\texttt{repeat}\ x\ (\texttt{add}\ x)\ a\ y = a + \texttt{mult}\ x\ y$	equiv	H, P	✓	0.455
19	$x \leq 101 \Rightarrow \texttt{mc91}\ x = 91$	nonrel	I	✓	0.233
20	$x \geq 0 \land y \geq 0 \Rightarrow \texttt{ack}\ x\ y > y$	nonrel	I	✓	0.316
21	$x \geq 0 \Rightarrow 2 \times \texttt{sum}\ x = x \times (x+1)$	nonrel	N	✓	0.275
22	$\texttt{dyn\_sys}\ 0.\ \not\rightarrow^* \texttt{assert false}$	nonrel	R,N	✓	0.189
23	$\texttt{flip\_mod}\ y\ x = \texttt{flip\_mod}\ y\ (\texttt{flip\_mod}\ y\ x)$	idem	P	✓	13.290
24	$\texttt{noninter}\ h_1\ l_1\ l_2\ l_3 = \texttt{noninter}\ h_2\ l_1\ l_2\ l_3$	nonint	P	✓	1.203
25	$\texttt{try find\_opt}\ p\ l = \texttt{Some}\ (\texttt{find}\ p\ l)\ \texttt{with}$ $\texttt{Not\_Found} \rightarrow \texttt{find\_opt}\ p\ l = \texttt{None}$	equiv	H, E	✓	1.065
26	$\texttt{try mem}\ (\texttt{find}\ ((=)\ x)\ l)\ l\ \texttt{with Not\_Found} \rightarrow \neg(\texttt{mem}\ x\ l)$	equiv	H, E	✓	1.056
27	$\texttt{sum\_list}\ l = \texttt{fold\_left}\ (+)\ 0\ l$	equiv	H	✓	6.148
28	$\texttt{sum\_list}\ l = \texttt{fold\_right}\ (+)\ l\ 0$	equiv	H	✓	0.508
29	$\texttt{sum\_fun randpos}\ n > 0$	equiv	H,D	✓	0.319
30	$\texttt{mult}\ x\ y = \texttt{mult\_Ccode}(x, y)$	equiv	P, C	✓	0.303

† A lemma $P_{\texttt{mult\_acc}}(x, y, a, r) \Rightarrow P_{\texttt{mult\_acc}}(x, y, a - x, r - x)$ is used

‡ A lemma $P_{\texttt{mult}}(x, y, r) \Rightarrow P_{\texttt{mult}}(x - 1, y, r - y)$ is used

Used a machine with Intel(R) Xeon(R) CPU (2.50 GHz, 16 GB of memory).

[CAV 2017] Unno et al. Automating Induction for Solving Horn Clauses.

Goal of This Talk

Stimulate the “immature” research field and, hopefully, find research collaborators by

1. discussing the **challenges in relational verification**,
2. **introducing our automated relational verification methods**, and
3. highlighting the **current limitations as well as future research directions**

Disclaimer: This talk will

- mainly deal with **fully automated** deductive relational verification,
- target **functional properties** and will not address trace or contextual properties, and
- focus on **infinite-state systems that can exhibit effects such as non-termination and non-determinism**, and we will not cover other effects or finite-state systems

Outline

1. Introduction
2. Challenges in Relational Verification
3. Automating Relational Verification
    1. Self-Composition (or Product Programs) [CAV 2021]
    2. Entailment Checking in $\mu\mathbf{CLP}$ [CAV 2017]
4. Current Limitations and Future Directions

[CAV 2021] Unno et al. Constraint-Based Relational Verification.
[CAV 2017] Unno et al. Automating Induction for Solving Horn Clauses.

Outline

1. Introduction
2. **Challenges in Relational Verification**
3. Automating Relational Verification
    1. Self-Composition (or Product Programs) [CAV 2021]
    2. Entailment Checking in μCLP [CAV 2017]
4. Current Limitations and Future Directions

[CAV 2021] Unno et al. Constraint-Based Relational Verification.
[CAV 2017] Unno et al. Automating Induction for Solving Horn Clauses.

Challenges in Relational Verification

- Although each program execution is **independent** (except the input correlation ensured by the precondition), proving relational properties is challenging without reasoning about the **correlation** of **intermediate execution states**
  - Analyzing each execution **separately** does not work well as it necessitates an **exact summarization** of the input-output relation for each execution
  - Thus, an automated verifier is required to simultaneously synthesize
    1. a correlation of intermediate states, namely a **relational invariant** and
    2. a **scheduler** (or **alignment**) that preserves it!

Comparison with Concurrent Programs Verification

- **Executions of programs that run concurrently can be highly dependent** as states can be synchronized with locks or semaphores, and data races can occur due to shared memory
  - Concurrent program is checked to satisfy the specification for all possible interleaving executions under a **demonic** scheduler
- Multiple executions in relational verification are independent
  - It is sufficient to prove that the specification holds in a certain interleaving execution, chosen by an **angelic** scheduler
  - However, to reason about the correlation between states at the time of output, it is **still necessary to effectively synchronize** at other intermediate points as well
  - Therefore, the complexity of the verification is more significantly influenced by the **differences between the programs** than by the complexity of each program individually

Example: Program Equivalence

```
int mult1(int x, int y) {
a: int z=0;
b: while(x>0) { x=x-1; z=z+y }
c: return z;
}
```

```
int mult2(int x, int y) {
d: int z=0;
e: if(x % 2 == 1) { x=x-1; z=z+y }
f: while(x>0) { x=x-2; z=z+2*y }
g: return z;
}
```

Let's align the executions to preserve $z_1 = z_2$ as much as possible

mult1 $(pc_1, x_1, y_1, z_1) : (a, 3,4, ?) \to (b, 3,4,0) \to (b, 2,4,4) \to (b, 1,4,8) \to (b, 0,4,12) \to (c, 0,4,12)$

$z_1 = z_2$ $z_1 = z_2$ $z_1 \neq z_2$ $z_1 = z_2$ $z_1 = z_2$

mult2 $(pc_2, x_2, y_2, z_2) : (d, 3,4, ?) \to (e, 3,4,0) \to (f, 2,4,4) \to (f, 0,4,12) \to (g, 0,4,12)$

But how to infer such predicate $x_1 \% 2 = 1$?

stuttering execution when $x_1 \% 2 = 1$ instead of lock-step one recovers $z_1 = z_2$

A relational invariant preserved by the above alignment:

$y_1 = y_2 \wedge (pc_1 = b \wedge pc_2 = f \wedge x_1 \% 2 = 0 \Rightarrow z_1 = z_2) \wedge (pc_1 = b \wedge pc_2 = f \wedge x_1 \% 2 = 1 \Rightarrow z_1 + y_1 = z_2)$

Example: Termination-Insensitive Non-Interference (TINI) ∈ 2-safety (1/2)

- doubleSquare(h, x) [CAV 2019] computes $2 \cdot x^2$ in two different ways depending on the ***high security*** input h

```
doubleSquare(bool h, int x) {
  int z, y=0;
  if(h) { z=2*x } else { z=x }
  while(z>0) { z--; y=y+x }
  if(!h) { y=2*y }
  return y;
}
```

Can an attacker infer the value of h by observing the ***low security*** input x and the return value y?

No! *TINI*(doubleSquare) holds:

$\forall \boldsymbol{h_1}, \boldsymbol{x_1}, \boldsymbol{y_1}, \boldsymbol{h_2}, \boldsymbol{x_2}, \boldsymbol{y_2}.$
$\texttt{doubleSquare}(\boldsymbol{h_1}, \boldsymbol{x_1}) \Downarrow \boldsymbol{y_1} \wedge$
$\texttt{doubleSquare}(\boldsymbol{h_2}, \boldsymbol{x_2}) \Downarrow \boldsymbol{y_2} \wedge$
$\boldsymbol{x_1} = \boldsymbol{x_2} \Rightarrow \boldsymbol{y_1} = \boldsymbol{y_2}$

[CAV 2019] Shemer et al. Property Directed Self Composition.

Example: Termination-Insensitive Non-Interference (TINI) ∈ 2-safety (2/2)

- The parallel executions of 2 copies of the program under the **partial lock-step scheduler** makes a **safe relational invariant** “complex”

```
doubleSquare(bool h, int x) {
  int z, y=0;
ℓ1: if(h) {z=2*x} else {z=x}
ℓ2: while(z>0) { z--; y=y+x }
ℓ3: if(!h) { y=2*y }
  return y; }
```

Any “simple” but safe and ind. relational invariant preserved by the partial lock-step scheduler?
↳ No! **Safe ind. relational invariant** for the scheduler is **not expressible** in LIA [CAV 2019]

Copy2 has exited the loop. Partial lock-step scheduler waits for Copy1 to exit the loop to synchronize

Executions (h_1, x_1, y_1, z_1) and (h_2, x_2, y_2, z_2) with the partial lock-step scheduler:

$\ell_1(\top,2,0,?) \to_{\ell_2} (\top,2,0,4) \to_{\ell_2} (\top,2,2,3) \to_{\ell_2} (\top,2,4,2) \to_{\ell_2} (\top,2,6,1) \to_{\ell_2} (\top,2,8,0) \to_{\ell_3} (\top,2,8,0)$

$x_1 = x_2$ | $y_1 = y_2$ | $y_1 = y_2$ | $y_1 = y_2$ | $y_1 \neq y_2$ | $y_1 \neq y_2$

$\ell_1(\bot,2,0,?) \to_{\ell_2} (\bot,2,0,2) \to_{\ell_2} (\bot,2,2,1) \to_{\ell_2} (\bot,2,4,0) \to_{\ell_3} (\bot,2,8,0)$

How can we prove that $y_1 = y_2$?

Example: Co-Termination ∈ Hyperliveness (1/2)

```
prog1(int x, int y) { while(x>0) { x=x-y; } }
prog2(int x, int y) { while(x>0) { x=x-2*y; } }
```

Do prog1(x_1, y_1) and prog2(x_2, y_2) agree on termination under the precondition $x_1 = x_2 \land y_1 = y_2$?

Yes! (Symmetric) **co-termination** holds:

$$\forall x_1, y_1, x_2, y_2. (x_1 = x_2 \land y_1 = y_2) \Rightarrow \left(\begin{array}{c} \exists z_1. \mathbf{prog1}(x_1, y_1) \Downarrow z_1 \\ \Rightarrow \neg(\mathbf{prog2}(x_2, y_2) \Uparrow) \end{array} \right) \land \left(\begin{array}{c} \exists z_2. \mathbf{prog2}(x_2, y_2) \Downarrow z_2 \\ \Rightarrow \neg(\mathbf{prog1}(x_1, y_1) \Uparrow) \end{array} \right)$$

One ***symmetric*** co-termination problem boils down to two ***asymmetric*** co-termination problems

Example: Co-Termination ∈ Hyperliveness (2/2)

```
prog1(int x, int y) { while(x>0) { x=x-y; } }
prog2(int x, int y) { while(x>0) { x=x-2*y; } }
```

How can we prove that this residual execution always terminates?

Executions (x_1, y_1) and (x_2, y_2) with the partial lock-step scheduler:

$(4,1) \rightarrow (3,1) \rightarrow (2,1) \rightarrow (1,1) \rightarrow (0,1)$

$(4,1) \rightarrow (2,1) \rightarrow (0,1)$

$x_1 = x_2$
$\wedge\, y_1 = y_2$

prog2 terminated

Example: (TI-/TS-)GNI ∈ ∀∃hyperproperties

- `gniEx`(h, l) ***non-deterministically*** returns a value $x \geq l$ in two different ways depending on the ***high security*** input h

```
gniEx(bool high, int low) {
  if(high) {
    int x = nondet_int();
    if(x >= low) { return x }
    else { while(true) {} }
  } else {
    int x = low;
    while(nondet_bool()){x++}
    return x;
  }
```

Can an attacker infer the value of h by observing the ***low security*** input l and the return value x?

No! *TIGNI*(gniEx) holds:

$$\forall h_1, h_2, l, x_1. (\texttt{gniEx}(h_1, l) \Downarrow x_1) \Rightarrow (\texttt{gniEx}(h_2, l) \Uparrow) \lor \exists x_2. (\texttt{gniEx}(h_2, l) \Downarrow x_2) \land x_1 = x_2$$

Demonic (∀) choice

Angelic (∃) choice

TSGNI(gniEx) also holds:

$$\forall h_1, h_2, l, x_1. (\texttt{gniEx}(h_1, l) \Downarrow x_1) \Rightarrow \exists x_2. (\texttt{gniEx}(h_2, l) \Downarrow x_2) \land x_1 = x_2$$

But how to solve such games between ∀ and ∃?

Other Challenging Examples in the Literature

```
int f(uint n, uint m) {
  int k = 0;
  for(uint i = 0; i < n; ++i) {
    for(uint j = 0; j < m; ++j) {
      k++;
    }
  }
  return k;
}
int g(uint n, uint m) {
  int k = 0;
  for(uint i = 0; i < n; ++i) {
    k += m;
  }
  return k;
}
```

[PLDI 2019]

Figure 12. A difficult problem for equivalence checking via product programs.

```
a: x := 0;                    0: i := 0;
b: while (x < NM) do          1: while (i < N) do
     a[x] := f(x);                  j := 0;
     x++                      2:    while (j < M) do
                                       A[i, j] := f(iM+j); j++;
                                    i++
```

[LFCS 2013]

Fig. 3. Loop tiling example

```
int x = 0;
while ( * ) {
  x++;
}
if ( x < 0 ) {
  error();
}
```

(a) P_0

```
int y = 0;
while ( * ) {
  if ( y == 12 ) {
    y = y + 2;
  } else {
    y++;
  }
}
if ( y < 0 ||
      y == 13 ) {
  error();
}
```

(b) Q_0

```
int z = 0;
while ( * && z < 12 ) {
  z++;
}
if ( z == 12 ) {
  z = z + 2;
}
while ( * && z > 12 ) {
  z++;
}
if ( z < 0 || z == 13 ) {
  error();
}
```

(c) Q_1

[CAV 2016]

Fig. 1. Programs P_0 and Q_0 and the loop-splitting optimization of Q_0.

[PLDI 2019] Churchill et al. Semantic program alignment for equivalence checking.
[LFCS 2013] Barthe et al. Beyond 2-Safety: Asymmetric Product Programs for Relational Program Verification
[CAV 2016] Fedyukovich et al. Property Directed Equivalence via Abstract Simulation

Outline

1. Introduction
2. Challenges in Relational Verification
3. **Automating Relational Verification**
    1. Self-Composition (or Product Programs) [CAV 2021]
    2. Entailment Checking in μ**CLP** [CAV 2017]
4. Current Limitations and Future Directions

[CAV 2021] Unno et al. Constraint-Based Relational Verification.
[CAV 2017] Unno et al. Automating Induction for Solving Horn Clauses.

Self-Composition
(or Product Programs)

Self-Composition (or Product Programs)

- Relational verification amounts to synthesis of **relational invariants** and **schedulers**
- **Self-composition** refers to a range of techniques aimed at synthesizing alignments represented symbolically as programs, automata, logical constraints, and games
  - Syntactic: [CSFW 2004, SAS 2005, PLAS 2006, FM 2011, LFCS 2013, SAS 2016, LPAR 2017]
  - Semantic: [CAV 2019a, CAV 2019b, PLDI 2019, CAV 2021, CAV 2022]

[CSFW 2004] Barthe et al. Secure Information Flow by Self-Composition.
[SAS 2005] Terauchi, Aiken. Secure Information Flow as a Safety Problem.
[PLAS 2006] Unno et al. Combining Type-Based Analysis and Model Checking for Finding Counterexamples against Non-Interference.
[FM 2011] Barthe et al. Relational Verification Using Product Programs.
[LFCS 2013] Barthe et al. Beyond 2-Safety: Asymmetric Product Programs for Relational Program Verification
[SAS 2016] Angelis et al. Relational Verification Through Horn Clause Transformation.
[LPAR 2017] Mordvinov, Fedyukovich. Synchronizing Constrained Horn Clauses.
[CAV 2019a] Farzan, Vandika. Automated Hypersafety Verification.
[CAV 2019b] Shemer et al. Property Directed Self Composition.
[PLDI 2019] Churchill et al. Semantic program alignment for equivalence checking.
[CAV 2021] Unno et al. Constraint-Based Relational Verification.
[CAV 2022] Beutner, Finkbeiner. Software Verification of Hyperproperties Beyond k-Safety.

Our Approach to Semantic Self-Composition [CAV 2021]

- **Soundly** and **completely** encode the simultaneous synthesis problem of **relational invariants** and **fair & semantic schedulers** needed for relational verification (k-safety, co-termination, and GNI) as a **constraint solving problem** of the class, we call **pfwCSP** that extends CHCs with
  1. head-disjunction (used to express scheduler fairness constraints),
  2. well-foundedness constraints (used for synthesizing co-termination witnesses),
  3. functionality constraints (used for synthesizing winning strategies for GNI)
- Generalize semantic self-composition for k-safety [CAV 2019] to GNI and co-termination
- For solving **pfwCSP**, provide a constraint solver **PCSat** based on the template-based CEGIS and an unsat-core based template refinement

[CAV 2019] Shemer et al. Property Directed Self Composition.
[CAV 2021] Unno et al. Constraint-Based Relational Verification.

Semantic Self-Composition for TI-NI

Can choose which program to execute depending on the states

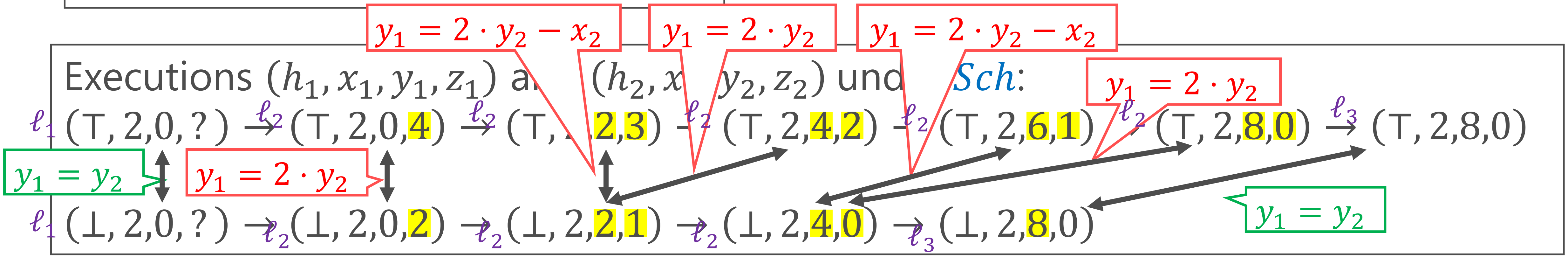
- Find a ***semantic scheduler*** Sch and a ***safe invariant*** of the parallel executions of 2 copies of the program under Sch

```
doubleSquare(bool h,int x) {
  int z, y=0;
ℓ1: if(h) {z=2*x} else {z=x}
ℓ2: while(z>0) { z--; y=y+x }
ℓ3: if(!h) { y=2*y }
  return y; }
```

Found scheduler Sch dictates:
both copies move if $z_1 = 2 \cdot z_2$,
1st copy moves if $z_1 = 2 \cdot z_2 + 1$

Found invariant expressible in LIA:

$$h_1 \wedge \neg h_2 \wedge x_1 = x_2 \wedge \begin{pmatrix} y_1 = 2 \cdot y_2 \wedge z_1 = 2 \cdot z_2 \vee \cdots \wedge \\ y_1 = 2 \cdot y_2 - x_2 \wedge z_1 = 2 \cdot z_2 + 1 \end{pmatrix}$$
$$\vee\, h_1 \wedge h_2 \wedge \cdots \vee \neg h_1 \wedge h_2 \wedge \cdots \vee \neg h_1 \wedge \neg h_2 \wedge \cdots$$



Prog. & Spec. → pfwCSP → SAT or UNSAT

Constraint Generation

Constraint Solving

represents a ***relational invariant*** preserved by a ***semantic scheduler***

the ***relational invariant*** is inductive and implies **TI-NI**

```
doubleSquare
  (bool h, int x) {
```

```
}
```

non-Horn clause that goes beyond **CHCs** !

the scheduler is ***fair***: at least one unfinished program must be scheduled if there is any ***(necessary for the soundness)***

represent the ***semantic scheduler***: if Sch_1 holds, 1st copy is scheduled

- $I(V_1, V_2) \Leftarrow x_1 \ldots \wedge$

$I(x_1, y_1', z_1', V_2) \ldots$

$\wedge \left(z_1 > 0 \wedge \ldots \; z_1 \ldots \right)$

$y_1 = 2 \cdot y_2 \Leftarrow I(V_1, V_2) \ldots \le 0 \wedge z_2 \le 0,$

$\left(Sch_1(V_1, V_2) \vee Sch_2(V_1, V_2) \vee Sch_{1,2}(V_1, V_2) \right)$
$\Leftarrow I(V_1, V_2) \wedge (z_1 > 0 \vee z_2 > 0),$

$z_1 > 0 \Leftarrow I(V_1, V_2) \wedge Sch_1(V_1, V_2) \wedge z_2 > 0,$

- $z_2 > 0 \Leftarrow I(V_1, V_2) \wedge Sch_2(V_1, V_2) \wedge z_1 > 0,$

where $V_1 = x_1, y_1, z_1$ and $V_2 = x_2, y_2, z_2$

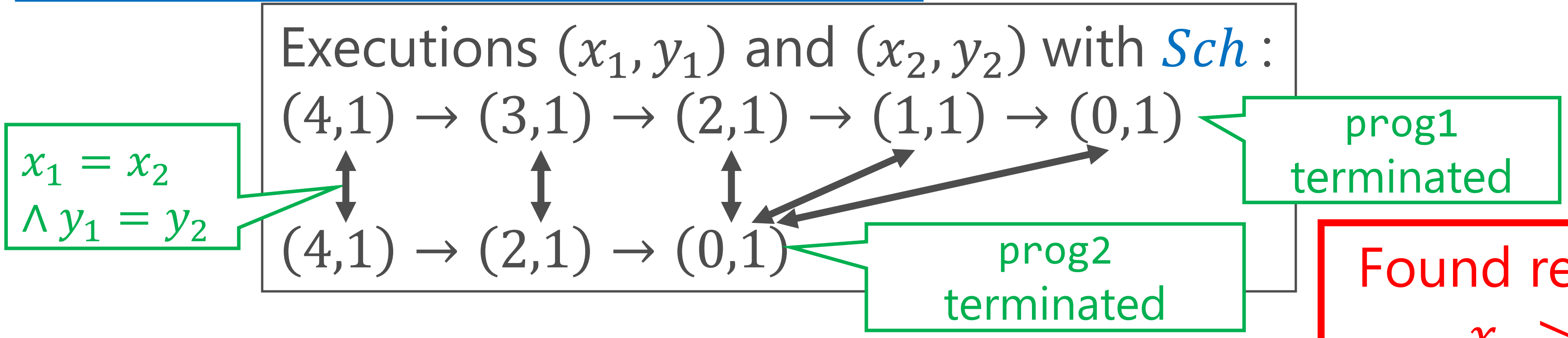
Semantic Self-Composition for Asymmetric Co-Termination

- Find a ***fair semantic scheduler*** Sch, a ***relational invariant***, and a ***well-founded relation*** under Sch

```
prog1(int x, int y) { while(x>0) { x=x-y; } }
prog2(int x, int y) { while(x>0) { x=x-2*y; } }
```

Found scheduler Sch dictates:
both programs move if $x_1 > 0 \wedge x_2 > 0$,
prog1 moves if $x_1 > 0 \wedge x_2 \le 0$

Found well-founded relation says:
if $x_1 > 0 \wedge x_2 \le 0$, then prog1 repeatedly ***decreases*** $\boldsymbol{x_1}$
but $\boldsymbol{x_1}$ ***is lower bounded by 0***



Found relational invariant implies:
$x_1 > 0 \wedge x_2 \le 0 \Rightarrow y_1 \ge 1$

Prog. & Spec. → pfwCSP → SAT or UNSAT

represents a ***relational invariant*** preserved by a ***semantic scheduler***

represents a ***total function*** used to select a bound b for each state V

... ***t*** is ... the ... plies

Input:

```
prog1(x, y) {
  while(x>0)
  { x=x-y }
}
```

- $I(0, b, V) \Leftarrow F_\lambda(V, b) \land x$ ...
- $I(d', b, x_1', y_1, x_2, y_2) \Leftarrow$ ...
  $(x_1 > 0 \land x_1' = x_1 - y_1$ ...
  $(x_1 > 0 \land x_2 > 0 \Rightarrow d'$ ...
- $R_\Downarrow(V_1, x_1 - y_1, y_1) \Leftarrow I($ ... $x_1 > 0 \land x_2 \leq 0,$

the scheduler is ***fair***: all unfinished programs must be ***eventually*** scheduled ***(necessary for soundness)***

$\left(Sch_1(d, b, V) \lor Sch_2(d, b, V) \lor Sch_{1,2}(d, b, V)\right)$
$\Leftarrow I(d, b, V) \land (x_1 > 0 \lor x_2 > 0),$
$x_1 > 0 \Leftarrow I(d, b, V) \land Sch_1(d, b, V) \land x_2 > 0,$
$x_2 > 0 \Leftarrow I(d, b, V) \land Sch_2(d, b, V) \land x_1 > 0,$
$d \in [-b, b] \land b \geq 0 \Leftarrow I(d, b, V_1, V_2) \land x_1 > 0 \land x_2 > 0$

represents a ***well-founded relation*** witnessing the ***termination of prog1*** ***relative to*** the ***termination of prog2***

the difference d between the numbers of steps taken by the two is within the bound b

Semantic Self-Composition for (TI-/TS-)GNI

- Find a ***fair semantic scheduler***, a ***relational invariant***, a ***well-founded relation***, and ***strategies for the non-deterministic choices of the angelic side***
- Augment the encodings for **TI-NI** and **Co-Term** with
  - ***predicate variables*** that represent the ***strategies:***
    total functions from states to choices of the angelic side
  - ***prophecy variables*** that represent the final outputs of ***the demonic side (necessary for the completeness)***
- Please refer to [CAV 2021] for details and examples

[CAV 2021] Unno et al. Constraint-Based Relational Verification.

Implementation and Evaluation

functional
Artifact
CAV AEC
Evaluated
available

- Evaluated our solver **PCSat** for solving **pfwCSP** on 20 relational verification problems:
    - 15 solved fully automatically, 5 required small hints

Program	Time (s)	#Iters	Program	Time (s)	#Iters
DoubleSquareNI_hFT	17.762	42	HalfSquareNI	11.853	35
DoubleSquareNI_hTF	26.495	55	ArrayInsert‡	118.671	73
DoubleSquareNI_hFF	2.944	9	SquareSum†‡	337.596	117
DoubleSquareNI_hTT	4.055	11	SimpleTS_GNI1	5.397	14
CotermIntro1	19.322	80	SimpleTS_GNI2	8.919	26
CotermIntro2	15.871	73	InfBranchTS_GNI	2.607	4
TS_GNI_hFT†	47.083	78	TI_GNI_hFT†	4.389	16
TS_GNI_hTF	5.076	17	TI_GNI_hTF	2.277	6
TS_GNI_hFF	7.174	24	TI_GNI_hFF	2.968	6
TS_GNI_hTT†	23.495	53	TI_GNI_hTT	4.148	22

The CoAR Verification & Synthesis Tool Chain

(https://github.com/hiroshi-unno/coar)

- Intermediate languages: (cf. CHCs, SyGuS, SemGuS, ...)
  - **pfwnCSP**: **p**redicate **C**onstraint **S**atisfaction **P**roblem with **f**unctionality, **w**ell-foundedness, & **n**on-emptiness constrains [AAAI20, CAV21dt, CAV21rel, POPL23opt...]
  - **μCLP**: **C**onstraint **L**ogic **P**rogram with arbitrarily nested inductive & co-inductive predicates (≈ fixpoint logic modulo theories) [POPL23mod, ...]
- Backends:
  - **PCSat** : **pfwnCSP** constraint solver/optimizer [AAAI20, CAV21dt, CAV21rel, POPL23opt, ...]
  - **MuVal** : **μCLP** solver based on **pfwnCSP** solving [CAV21dt, POPL23mod, ...]
  - **MuCyc** : **μCLP** solver based on cyclic-proof search [CAV17, POPL22, ...]
- Frontends:
  - Constraint generator for C [SAS19]
  - Constraint generator for LTS [CAV21dt, ...] (LLVM IR to LTS translator available)
  - **RCaml** : constraint generator for OCaml [FLOPS08, PPDP09, POPL13, SAS15, POPL18, LICS18, CAV18, POPL23aem, POPL24, ...]

Discussion

- Semantic self-composition has a high theoretical potential and promising experimental results have actually been obtained
- Current limitations
  - Relational verifiers based on semantic self-composition exhibit increased search costs and reduced efficiency as the capability of representable schedulers grows
- Future directions
  - Leverage various existing syntactic and semantic abstraction, search pruning, and symmetry breaking techniques to accelerate the search

Entailment Checking in $\boldsymbol{\mu}\mathbf{CLP}$

Program Refinement as *Generalized* Model Checking

- Program refinement verification $\models p \leq q$ generalizes ordinary model checking $p \models \phi$
  - **A specification of p is given as a program q** instead of a logical formula ϕ
  - q can encode the given ϕ (if the programming language is expressive enough)
  - q can be **a reference implementation** (cf. seL4 Project) or **an abstract model** represented as a highly non-deterministic program
- This motivates me to investigate entailment checking problems $\psi_1 \models \psi_2$ in a first-order fixpoint logic modulo theories we call μCLP [CAV 2017, LICS 2018, POPL 2023]
- **Relational verification boils down to entailment checking in μCLP**

[CAV 2017] Unno et al. Automating Induction for Solving Horn Clauses.
[LICS 2018] Nanjo et al. A Fixpoint Logic and Dependent Effects for Temporal Property Verification.
[POPL 2023] Unno et al. Modular Primal-Dual Fixpoint Logic Solving for Temporal Verification.

Example: Functional Program & Relational Spec.

```
(* recursive function to compute "x × y" *)
let rec mult x y =
  if y = 0 then 0 else x + mult x (y - 1)

(* tail recursive function to compute "x × y + a" *)
let rec mult_acc x y a =
  if y = 0 then a else mult_acc x (y - 1) (a + x)

(* functional equivalence of mult and mult_acc *)
let main x y a = assert (mult x y + a = mult_acc x y a)
```

CHCs Constraint Generation based on Dependent Refinement Types [PPDP 2009]

```
let rec mult x y =
  if y = 0 then 0
  else x + mult x (y - 1)
let rec mult_acc x y a =
  if y = 0 then a
  else mult_acc x (y - 1) (a + x)
```

```
let main x y a =
  assert (mult x y + a
            = mult_acc x y a)
```

$$P(x,0,0)$$
$$P(x,y,x+r) \Leftarrow P(x,y-1,r) \wedge y \neq 0$$
$$Q(x,0,a,a)$$
$$Q(x,y,a,r) \Leftarrow Q(x,y-1,a+x,r) \wedge y \neq 0$$
$$s_1 + a = s_2 \Leftarrow P(x,y,s_1) \wedge Q(x,y,a,s_2)$$

[PPDP 2009] Unno, Kobayashi. Dependent Type Inference with Interpolants.

CHC Solving via Entailment Checking in μCLP

The CHCs on the right is satisfiable if and only if the following entailment holds in μ**CLP**

$$P(x, y, s_1), Q(x, y, a, s_2) \vDash s_1 + a = s_2$$

where

$$P(x, y, z) =_\mu \begin{array}{c} y = 0 \lor \\ y \neq 0 \land P(x, y-1, r) \land z = x + r \end{array}$$

$$Q(x, y, a, r) =_\mu \begin{array}{c} y = 0 \land r = a \lor \\ y \neq 0 \land Q(x, y-1, a+x, r) \end{array}$$

$$P(x, 0, 0)$$
$$P(x, y, x+r) \Leftarrow P(x, y-1, r) \land y \neq 0$$
$$Q(x, 0, a, a)$$
$$Q(x, y, a, r) \Leftarrow Q(x, y-1, a+x, r) \land y \neq 0$$
$$s_1 + a = s_2 \Leftarrow P(x, y, s_1) \land Q(x, y, a, s_2)$$

μ**CLP**: An Extension of CLP with ***Quantifiers*** and ***Arbitrarily-Nested (Co-)Inductive Predicates***

- Can be seen as a first-order ***fixpoint logic*** modulo background theories T

(formulas) $\phi ::= \bot \mid \top \mid A(\vec{t}) \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \forall x. \phi \mid \exists x. \phi \mid p(\vec{t})$

(terms) $t ::= x \mid f(\vec{t})$ (predicates) $p ::= X \mid \boldsymbol{\mu X. \lambda \vec{x}. \phi} \mid \boldsymbol{\nu X. \lambda \vec{x}. \phi}$

  - A ranges over *predicate* symbols and f ranges over *function* symbols in T,
  - x ranges over *term* variables and X ranges over *predicate* variables,
  - Predicates occur only positively in $\boldsymbol{\mu X. \lambda \vec{x}. \phi}$ and $\boldsymbol{\nu X. \lambda \vec{x}. \phi}$ for monotonicity
  - Least fixpoints $\boldsymbol{\mu X. \lambda \vec{x}. \phi}$ represent *inductive predicates*, and greatest fixpoints $\boldsymbol{\nu X. \lambda \vec{x}. \phi}$ represent *co-inductive predicates*
    - We also use equational form: $X(\vec{x}) =_\mu \phi$ and $X(\vec{x}) =_\nu \phi$

- Examples (integer arithmetic as T):
- ➢ $(\mu X. \lambda x. x = 0 \vee X(x-1))(x) \Leftrightarrow x = 0 \vee x = 1 \vee x = 2 \vee \cdots \Leftrightarrow \exists z \geq 0. x = z$
- ➢ $(\nu X. \lambda x. x \geq 0 \wedge X(x+1))(x) \Leftrightarrow x \geq 0 \wedge x + 1 \geq 0 \wedge x + 2 \geq 0 \wedge \cdots \Leftrightarrow \forall z \geq 0. x + z \geq 0$

Entailment Checking via Inductive Theorem Proving

$P(x, 0, 0) \quad P(x, y, x + r) \Leftarrow P(x, y - 1, r) \land y \neq 0$

$Q(x, 0, a, a) \quad Q(x, y, a, r) \Leftarrow Q(x, y - 1, a + x, r) \land y \neq 0$

$s_1 + a = s_2 \Leftarrow P(x, y, s_1) \land Q(x, y, a, s_2)$

$$\frac{\models y = 0 \land r = 0}{P(x, y, r)} \quad \frac{P(x, y - 1, r - x) \quad \models y \neq 0}{P(x, y, r)}$$

$$\frac{\models y = 0 \land a = r}{Q(x, y, a, r)} \quad \frac{Q(x, y - 1, a + x, r) \quad \models y \neq 0}{Q(x, y, a, r)}$$

Prove this by induction on derivation of $P(x, y, s_1)$

$P(x, y, s_1) \land Q(x, y, a, s_2) \models s_1 + a = s_2$

Principle of Induction on Derivation

$$\forall D.\ \psi(D) \quad \text{if and only if}$$
$$\forall D.\ \big(\forall D'.\, D' \prec D \Rightarrow \psi(D')\big) \Rightarrow \psi(D)$$

where $D' \prec D$ represents that
D' is a strict sub-derivation of D

$$D = \dfrac{\dfrac{\dfrac{D_1}{J_3} \quad D_2}{J_2} \quad D_3 \quad \dfrac{D_4}{J_4}}{J_1}$$

Assume $\psi(D_1), \psi(D_2)$,
$\psi(D_3), \psi(D_4)$,
$\psi\left(\dfrac{\vdots}{J_2}\right),\ \psi\left(\dfrac{\vdots}{J_3}\right), \psi\left(\dfrac{\vdots}{J_4}\right)$
and prove $\psi(D)$

CHC Solving:

$$P(x, 0, 0)$$

$$P(x, y, x + r) \Leftarrow P(x, y - 1, r) \wedge y \neq 0$$

$$Q(x, 0, a, a)$$

$$Q(x, y, a, r) \Leftarrow Q(x, y - 1, a + x, r) \wedge y \neq 0$$

$$s_1 + a = s_2 \Leftarrow P(x, y, s_1) \wedge Q(x, y, a, s_2)$$

Induction hypotheses and lemmas

Judgment

$$\emptyset; P(x, y, s_1), Q(x, y, a, s_2) \vdash s_1 + a = s_2$$

Premises

$$\frac{\models y = 0 \wedge \ldots}{P(x, y, r)} \qquad \frac{\ldots, y - 1, r - x) \quad \models y \neq 0}{P(x, y, r)}$$

$$\frac{\models y = 0 \wedge a = r}{Q(x, y, a, r)} \qquad \frac{Q(x, y - 1, a + x, r) \quad \models y \neq 0}{Q(x, y, a, r)}$$

$$\frac{\models y = 0 \land r = 0}{P(x, y, r)} \qquad \frac{P(x, y-1, r-x) \quad \models y \neq 0}{P(x, y, r)}$$

$$\frac{\models y = 0 \land a = r}{Q(x, y, a, r)} \qquad \frac{Q(x, y-1, a+x, r) \quad \models y \neq 0}{Q(x, y, a, r)}$$

Add an induction hypothesis

Guard to avoid unsound application

$$\gamma = \begin{array}{l} \forall x', y', s_1', a', s_2'.\, D\big(P(x', y', s_1')\big) \prec D\big(P(x, y, s_1)\big) \land \\ \quad P(x', y', s_1') \land Q(x', y', a', s_2') \Rightarrow s_1' + a' = s_2' \end{array}$$

Induct **Unfold**

Case analysis on the last rule used

$$\frac{\gamma; \cdots, y = 0 \land s_1 = 0 \vdash \cdots \qquad \gamma; \cdots, P(x, y-1, s_1 - x), y \neq 0 \vdash \cdots}{\emptyset; \underline{P(x, y, s_1)}, Q(x, y, a, s_2) \vdash s_1 + a = s_2}$$

$$\frac{\models y = 0 \wedge r = 0}{P(x, y, r)} \qquad \frac{P(x, y-1, r-x) \quad \models y \neq 0}{P(x, y, r)}$$

$$\frac{\models y = 0 \wedge a = r}{Q(x, y, a, r)} \qquad \frac{Q(x, y-1, a+x, r) \quad \models y \neq 0}{Q(x, y, a, r)}$$

Case analysis on the last rule used

Unfold

$$\dfrac{\dfrac{\gamma; \cdots, \cdots \wedge y = 0 \wedge a = s_2 \vdash \cdots \qquad \gamma; \cdots, Q(x, y-1, a+x, s_2), \cdots \wedge y \neq 0 \vdash \cdots}{\gamma; P(x, y, s_1), Q(x, y, a, s_2), y = 0 \wedge s_1 = 0 \vdash s_1 + a = s_2}}{\emptyset; P(x, y, s_1), Q(x, y, a, s_2) \vdash s_1 + a = s_2}$$

$$\frac{\models y = 0 \land r = 0}{P(x, y, r)} \qquad \frac{P(x, y-1, r-x) \quad \models y \neq 0}{P(x, y, r)}$$

$$\frac{\models y = 0 \land a = r}{Q(x, y, a, r)} \qquad \frac{Q(x, y-1, a+x, r) \quad \models y \neq 0}{Q(x, y, a, r)}$$

$$\frac{\dfrac{\gamma; \cdots, \cdots \land y = 0 \land a = s_2 \vdash \ \cdots}{\gamma; P(x, y, s_1), Q(x, y, a, s_2), y = 0 \land s_1 = 0 \vdash s_1 + a = s_2}}{\emptyset; P(x, y, s_1), Q(x, y, a, s_2) \vdash s_1 + a = s_2}$$

$$\frac{\models y = 0 \land r = 0}{P(x, y, r)} \qquad \frac{P(x, y-1, r-x) \quad \models y \neq 0}{P(x, y, r)}$$

$$\frac{\models y = 0 \land a = r}{Q(x, y, a, r)} \qquad \frac{Q(x, y-1, a+x, r) \quad \models y \neq 0}{Q(x, y, a, r)}$$

Validity checking

Valid

$$\cfrac{\cfrac{\cfrac{\models y = 0 \land s_1 = 0 \land a = s_2 \Rightarrow s_1 + a = s_2}{\gamma; \cdots, y = 0 \land s_1 = 0 \land a = s_2 \vdash s_1 + a = s_2}}{\gamma; P(x, y, s_1), Q(x, y, a, s_2), y = 0 \land s_1 = 0 \vdash s_1 + a = s_2}}{\emptyset; P(x, y, s_1), Q(x, y, a, s_2) \vdash s_1 + a = s_2}$$

$$\frac{\models y = 0 \land r = 0}{P(x, y, r)} \qquad \frac{P(x, y-1, r-x) \quad \models y \neq 0}{P(x, y, r)}$$

$$\frac{\models y = 0 \land a = r}{Q(x, y, a, r)} \qquad \frac{Q(x, y-1, a+x, r) \quad \models y \neq 0}{Q(x, y, a, r)}$$

$$\frac{\dfrac{\gamma; \cdots, Q(x, y-1, a+x, s_2), \cdots \land y \neq 0 \vdash \cdots}{\gamma; P(x, y, s_1), Q(x, y, a, s_2), y = 0 \land s_1 = 0 \vdash s_1 + a = s_2}}{\emptyset; P(x, y, s_1), Q(x, y, a, s_2) \vdash s_1 + a = s_2}$$

$$\frac{\models y = 0 \land r = 0}{P(x, y, r)} \qquad \frac{P(x, y-1, r-x) \qquad \models y \neq 0}{P(x, y, r)}$$

$$\frac{\models y = 0 \land a = r}{Q(x, y, a, r)} \qquad \frac{Q(x, y-1, a+x, r) \qquad \models y \neq 0}{Q(x, y, a, r)}$$

Valid

$$\frac{\frac{\frac{\models y = 0 \land s_1 = 0 \land y \neq 0 \Rightarrow s_1 + a = s_2}{\gamma; \cdots, Q(x, y-1, a+x, s_2), y = 0 \land s_1 = 0 \land y \neq 0 \vdash s_1 + a = s_2}}{\gamma; P(x, y, s_1), Q(x, y, a, s_2), y = 0 \land s_1 = 0 \vdash s_1 + a = s_2}}{\emptyset; P(x, y, s_1), Q(x, y, a, s_2) \vdash s_1 + a = s_2}$$

$$\frac{\models y = 0 \land r = 0}{P(x,y,r)} \qquad \frac{P(x, y-1, r-x) \quad \models y \neq 0}{P(x,y,r)}$$

$$\frac{\models y = 0 \land a = r}{Q(x,y,a,r)} \qquad \frac{Q(x, y-1, a+x, r) \quad \models y \neq 0}{Q(x,y,a,r)}$$

$$\frac{\gamma; \cdots, y = 0 \land s_1 = 0 \vdash \cdots}{\emptyset; P(x,y,s_1), Q(x,y,a,s_2) \vdash s_1 + a = s_2}$$

$$\frac{\models y = 0 \land r = 0}{P(x, y, r)} \qquad \frac{P(x, y-1, r-x) \quad \models y \neq 0}{P(x, y, r)}$$

$$\frac{\models y = 0 \land a = r}{Q(x, y, a, r)} \qquad \frac{Q(x, y-1, a+x, r) \quad \models y \neq 0}{Q(x, y, a, r)}$$

$$\frac{\gamma; \cdots, P(x, y-1, s_1 - x), y \neq 0 \vdash \cdots}{\emptyset; P(x, y, s_1), Q(x, y, a, s_2) \vdash s_1 + a = s_2}$$

$$\frac{\models y = 0 \land r = 0}{P(x, y, r)} \qquad \frac{P(x, y-1, r-x) \quad \models y \neq 0}{P(x, y, r)}$$

$$\frac{\models y = 0 \land a = r}{Q(x, y, a, r)} \qquad \frac{Q(x, y-1, a+x, r) \quad \models y \neq 0}{Q(x, y, a, r)}$$

Unfold

Case analysis on the last rule used

$$\frac{\dfrac{\gamma; \cdots, \cdots \land y = 0 \land a = s_2 \vdash \ \cdots \qquad \gamma; \cdots, Q(x, y-1, a+x, s_2), \cdots \land y \neq 0 \vdash \ \cdots}{\gamma; P(x, y, s_1), Q(x, y, a, s_2), P(x, y-1, s_1 - x), y \neq 0 \vdash s_1 + a = s_2}}{\emptyset; P(x, y, s_1), Q(x, y, a, s_2) \vdash s_1 + a = s_2}$$

$$\frac{\models y = 0 \land r = 0}{P(x, y, r)} \qquad \frac{P(x, y-1, r-x) \qquad \models y \neq 0}{P(x, y, r)}$$

$$\frac{\models y = 0 \land a = r}{Q(x, y, a, r)} \qquad \frac{Q(x, y-1, a+x, r) \qquad \models y \neq 0}{Q(x, y, a, r)}$$

$$\frac{\dfrac{\gamma; \cdots, \cdots \land y = 0 \land a = s_2 \vdash \cdots}{\gamma; P(x, y, s_1), Q(x, y, a, s_2), P(x, y-1, s_1 - x), y \neq 0 \vdash s_1 + a = s_2}}{\emptyset; P(x, y, s_1), Q(x, y, a, s_2) \vdash s_1 + a = s_2}$$

$$\frac{\models y = 0 \wedge r = 0}{P(x, y, r)} \qquad \frac{P(x, y-1, r-x) \quad \models y \neq 0}{P(x, y, r)}$$

$$\frac{\models y = 0 \wedge a = r}{Q(x, y, a, r)} \qquad \frac{Q(x, y-1, a+x, r) \quad \models y \neq 0}{Q(x, y, a, r)}$$

Valid

$$\frac{\dfrac{\dfrac{\models y \neq 0 \wedge y = 0 \wedge a = s_2 \Rightarrow s_1 + a = s_2}{\gamma; \cdots, y \neq 0 \wedge y = 0 \wedge a = s_2 \vdash s_1 + a = s_2}}{\gamma; P(x, y, s_1), Q(x, y, a, s_2), P(x, y-1, s_1 - x), y \neq 0 \vdash s_1 + a = s_2}}{\emptyset; P(x, y, s_1), Q(x, y, a, s_2) \vdash s_1 + a = s_2}$$

$$\frac{\models y = 0 \land r = 0}{P(x,y,r)} \qquad \frac{P(x, y-1, r-x) \quad \models y \neq 0}{P(x,y,r)}$$

$$\frac{\models y = 0 \land a = r}{Q(x,y,a,r)} \qquad \frac{Q(x, y-1, a+x, r) \quad \models y \neq 0}{Q(x,y,a,r)}$$

$$\frac{\dfrac{\gamma; \cdots, Q(x, y-1, a+x, s_2), \cdots \land y \neq 0 \vdash \cdots}{\gamma; P(x,y,s_1), Q(x,y,a,s_2), P(x, y-1, s_1 - x), y \neq 0 \vdash s_1 + a = s_2}}{\emptyset; P(x,y,s_1), Q(x,y,a,s_2) \vdash s_1 + a = s_2}$$

$$\frac{\models y = 0 \wedge r = 0}{P(x, y, r)} \qquad \frac{P(x, y-1, r-x) \quad \models y \neq 0}{P(x, y, r)}$$

$$\frac{\models y = 0 \wedge a = r}{Q(x, y, a, r)} \qquad \frac{Q(x, y-1, a+x, r) \quad \models y \neq 0}{Q(x, y, a, r)}$$

$$\sigma(\gamma) = \begin{array}{c} D(P(x, y-1, s_1 - x)) \prec D(P(x, y, s_1)) \wedge P(x, y-1, s_1 - x) \wedge \\ Q(x, y-1, a+x, s_2) \Rightarrow (s_1 - x) + (a + x) = s_2 \end{array}$$

IndHyp Apply induction hypothesis

$$\frac{\frac{\frac{\gamma; \cdots, y \neq 0 \wedge (s_1 - x) + (a + x) = s_2 \vdash s_1 + a = s_2}{\gamma; \cdots, P(x, y-1, s_1 - x), Q(x, y-1, a+x, s_2), y \neq 0 \vdash s_1 + a = s_2}}{\gamma; P(x, y, s_1), Q(x, y, a, s_2), P(x, y-1, s_1 - x), y \neq 0 \vdash s_1 + a = s_2}}{\emptyset; P(x, y, s_1), Q(x, y, a, s_2) \vdash s_1 + a = s_2}$$

$$\frac{\models y = 0 \land r = 0}{P(x,y,r)} \qquad \frac{P(x,y-1,r-x) \qquad \models y \neq 0}{P(x,y,r)}$$

$$\frac{\models y = 0 \land a = r}{Q(x,y,a,r)} \qquad \frac{Q(x,y-1,a+x,r) \qquad \models y \neq 0}{Q(x,y,a,r)}$$

Valid

$$\cfrac{\cfrac{\cfrac{\cfrac{\models y \neq 0 \land (s_1 - x) + (a + x) = s_2 \Rightarrow s_1 + a = s_2}{\gamma; \cdots, y \neq 0 \land (s_1 - x) + (a + x) = s_2 \vdash s_1 + a = s_2}}{\gamma; \cdots, {\color{red}P(x, y-1, s_1 - x)}, Q(x, y-1, a+x, s_2), y \neq 0 \vdash s_1 + a = s_2}}{\gamma; P(x,y,s_1), Q(x,y,a,s_2), {\color{red}P(x, y-1, s_1 - x)}, y \neq 0 \vdash s_1 + a = s_2}}{\emptyset; P(x,y,s_1), Q(x,y,a,s_2) \vdash s_1 + a = s_2}$$

QED

Properties of the Inductive Proof System for CHCs Solving

- Soundness: If the goal is proved, the original CHCs have a solution (which may not be expressible in the background theory)
- *Relative* Completeness: If the original CHCs have a solution *expressible in the background theory*, the goal is provable

Automating Induction

- Use the following rule application strategy:
  - Repeatedly apply INDHYP until no new premises are added
  - Apply VALID whenever a new premise is added
  - Select some $P(\tilde{t})$ and apply INDUCT and UNFOLD
- Close a proof branch by VALID that uses
  - SMT solvers: provide efficient and powerful reasoning about **data structures** (e.g., integers, reals, algebraic data structures) but predicates are abstracted as uninterpreted functions
  - CHC solvers: provide bit costly but powerful reasoning about **inductive predicates**

A Prototype Entailment Checker **MuCyc**
http://lfp.dip.jp/rcaml/

- Use **Z3** and **SPACER** respectively as the backend SMT and CHC solvers
- Integrated with a dependent refinement type based CHC generation tool **RCaml** for OCaml
- Currently support entailments in
  - The fragment corresponding to CHCs: $P_1(\overrightarrow{x_1}), \ldots, P_n(\overrightarrow{x_n}) \models \phi$ and
  - $P_1(\overrightarrow{x_1}), \ldots, P_n(\overrightarrow{x_n}) \models Q(\vec{y})$, which is useful for program refinement verification and proving lemmas to prove entailments in the above fragment (cf. commutativity proof of mult)
- Can prove and then exploit lemmas which are:
  - User-supplied,
  - Heuristically conjectured from the given constraints, or
  - Automatically generated by an abstract interpreter
- Can generate a counterexample (if any)

Experiments on IsaPlanner Benchmark Set

- 85 (mostly) relational verification problems of total functions on inductively defined data structures

Inductive Theorem Prover	#Successfully Proved
RCaml	68
Zeno	82 [Sonnex+ ’12]
HipSpec	80 [Claessen+ ’13]
CVC4	80 [Reynolds+ ’15]
ACL2s	74 (according to [Sonnex+ ’12])
IsaPlanner	47 (according to [Sonnex+ ’12])
Dafny	45 (according to [Sonnex+ ’12])

Support automatic lemma discovery & goal generalization

Experiments on Benchmark Programs with Advanced Language Features & Side-Effects

- 30 (mostly) relational verification problems for:
  - Complex integer functions: Ackermann, McCarthy91
  - Nonlinear real functions: dyn_sys
  - Higher-order functions: fold_left, fold_right, repeat, find, …
  - Exceptions: find
  - Non-terminating functions: mult, sum, …
  - Non-deterministic functions: randpos
  - Imperative procedures: mult_Ccode

ID	specification	kind	features	result	time (sec.)
1	$\texttt{mult}\ x\ y + a = \texttt{mult\_acc}\ x\ y\ a$	equiv	P	✓	0.378
2	$\texttt{mult}\ x\ y = \texttt{mult\_acc}\ x\ y\ 0$	equiv	P	✓†	0.803
3	$\texttt{mult}\ (1+x)\ y = y + \texttt{mult}\ x\ y$	equiv	P	✓	0.403
4	$y \geq 0 \Rightarrow \texttt{mult}\ x\ (1+y) = x + \texttt{mult}\ x\ y$	equiv	P	✓	0.426
5	$\texttt{mult}\ x\ y = \texttt{mult}\ y\ x$	comm	P	✓‡	0.389
6	$\texttt{mult}\ (x+y)\ z = \texttt{mult}\ x\ z + \texttt{mult}\ y\ z$	dist	P	✓	1.964
7	$\texttt{mult}\ x\ (y+z) = \texttt{mult}\ x\ y + \texttt{mult}\ x\ z$	dist	P	✓	4.360
8	$\texttt{mult}\ (\texttt{mult}\ x\ y)\ z = \texttt{mult}\ x\ (\texttt{mult}\ y\ z)$	assoc	P	✗	n/a
9	$0 \leq x_1 \leq x_2 \wedge 0 \leq y_1 \leq y_2 \Rightarrow \texttt{mult}\ x_1\ y_1 \leq \texttt{mult}\ x_2\ y_2$	mono	P	✓	0.416
10	$\texttt{sum}\ x + a = \texttt{sum\_acc}\ x\ a$	equiv		✓	0.576
11	$\texttt{sum}\ x = x + \texttt{sum}\ (x-1)$	equiv		✓	0.452
12	$x \leq y \Rightarrow \texttt{sum}\ x \leq \texttt{sum}\ y$	mono		✓	0.593
24	$\texttt{noninter}\ h_1\ l_1\ l_2\ l_3 = \texttt{noninter}\ h_2\ l_1\ l_2\ l_3$	nonint	P	✓	1.203
25	$\texttt{try find\_opt}\ p\ l = \texttt{Some}\ (\texttt{find}\ p\ l)\ \texttt{with}$ $\texttt{Not\_Found} \rightarrow \texttt{find\_opt}\ p\ l = \texttt{None}$	equiv	H, E	✓	1.065
26	$\texttt{try mem}\ (\texttt{find}\ ((=)\ x)\ l)\ l\ \texttt{with Not\_Found} \rightarrow \neg(\texttt{mem}\ x\ l)$	equiv	H, E	✓	1.056
27	$\texttt{sum\_list}\ l = \texttt{fold\_left}\ (+)\ 0\ l$	equiv	H	✓	6.148
28	$\texttt{sum\_list}\ l = \texttt{fold\_right}\ (+)\ l\ 0$	equiv	H	✓	0.508
29	$\texttt{sum\_fun randpos}\ n > 0$	equiv	H,D	✓	0.319
30	$\texttt{mult}\ x\ y = \texttt{mult\_Ccode}(x, y)$	equiv	P, C	✓	0.303

- 28 (2 required lemmas) successfully proved by **MuCyc**
- 3 proved by CHC constraint solver **μZ PDR**
- 2 proved by inductive theorem prover **CVC4** (if inductive predicates are encoded using uninterpreted functions)

† A lemma $P_{\texttt{mult\_acc}}(x, y, a, r) \Rightarrow P_{\texttt{mult\_acc}}(x, y, a - x, r - x)$ is used

‡ A lemma $P_{\texttt{mult}}(x, y, r) \Rightarrow P_{\texttt{mult}}(x - 1, y, r - y)$ is used

Used a machine with Intel(R) Xeon(R) CPU (2.50 GHz, 16 GB of memory).

Discussion

- The integration of **SMT solving**, **CHC solving**, and **inductive theorem proving** resulted in an automated **relational verifier** across programs in various paradigms with **advanced language features** and **side-effects**
- Current limitations
  - Limited support for automatic lemma discovery and goal generalization
  - Does not support the full fragment of $\boldsymbol{\mu}$**CLP**
- Future directions
  - Generalize the recently observed connection of (co)inductive theorem proving to invariant and ranking function synthesis [LICS 2018, POPL 2023] and software model checking [POPL 2022] to the full fragment of μCLP

[LICS 2018] Nanjo et al. A Fixpoint Logic and Dependent Effects for Temporal Property Verification.
[POPL 2022] Tsukada, Unno. Software Model-Checking as Cyclic-Proof Search.
[POPL 2023] Unno et al. Modular Primal-Dual Fixpoint Logic Solving for Temporal Verification.

Outline

1. Introduction
2. Challenges in Relational Verification
3. Automating Relational Verification
    1. Self-Composition (or Product Programs) [CAV 2021]
    2. Entailment Checking in μCLP [CAV 2017]
4. **Current Limitations and Future Directions**

[CAV 2021] Unno et al. Constraint-Based Relational Verification.
[CAV 2017] Unno et al. Automating Induction for Solving Horn Clauses.

Current Limitations of Both Approaches

- It often becomes impossible to establish program refinement and equivalence, when there is **movement of statements across loops or recursions**
  - E.g., loop-invariant code motion, loop interchange, loop fusion, …
  - Using **commutativity** or **idempotency** at the right times may help establish program refinement or equivalence, but more research is needed to automate it
    - Although not automated, the proof system for entailment in **μCLP** can prove commutativity and idempotency and use them as lemmas to prove other entailments that involve reordering

Ongoing and Future Work

- Develop a general theory and algorithms for **aligning reordered executions**
- Improve the efficiency of semantic self-composition by incorporating abstraction, search pruning, and symmetry breaking techniques
- Automate relational entailments checking in the full class of μCLP
- Automate program verification of:
  - **Temporal relational properties** expressed in hyperlogics (HyperLTL, HyperCTL*, …)
  - **Probabilistic relational properties**, motivated from security, privacy, cryptography, and machine learning

Conclusion

- Relational verification amounts to synthesis of **relational invariants** and **schedulers**
- Emerging **semantic self-composition** techniques enable precise alignment but require further development to be refined into an efficient solver
- An alternative approach based on **entailment checking in μCLP**, a first-order fixpoint logic, shows promise, though it requires more automation through the adoption of software model checking and theorem proving techniques to fully realize the potential of this approach
- In both automated approaches, **aligning reordered executions** remains a challenge

Acknowledgements

- This talk is based in part on the following two papers
  1. Hiroshi Unno, Tachio Terauchi, Eric Koskinen:
     Constraint-Based Relational Verification. CAV (1) 2021: 742-766
  2. Hiroshi Unno, Sho Torii, Hiroki Sakamoto:
     Automating Induction for Solving Horn Clauses. CAV (2) 2017: 571-591
- I would like to thank my research collaborators Tachio and Eric as well as my former students Sho and Hiroki.

Questions?