technology
from seed

Automatic Equivalence Checking of UF+IA Programs

Nuno Lopes and José Monteiro

Why Equivalence Checking?

- Algorithm recognition
- Regression checking
- Manual optimization checking
- Compiler optimization verification
- Information flow (non-interference) proofs

Example:
Are these programs equivalent?

```
i := 0
while i < n do
    k := f(k, i)
    i := i + 1
```

```
i := n
while i ≥ 1 do
    k := f(k, n - i)
    i := i - 1

if n ≤ 0 then
    i := 0
else
    i := n
```

Example:
Sequential composition is no solution

assume $v = \underline{v}$
$i := 0$

while $i < n$ **do**
  $k :=$ **f**$(k,\ i)$
  $i := i + 1$

$\underline{i} := \underline{n}$
while $\underline{i} \geq 1$ **do**
  $\underline{k} :=$ **f**$(\underline{k},\ \underline{n} - \underline{i})$
  $\underline{i} := \underline{i} - 1$

if $\underline{n} \leq 0$ **then**
  $\underline{i} := 0$
else
  $\underline{i} := \underline{n}$

assert $v = \underline{v}$

Outline

- Program equivalence
- Example
- Algorithm
- Application to compiler optimizations
- Evaluation: CORK

Program equivalence

- Functional equivalence: non-deterministic behavior is not supported
- Partial equivalence: check only terminating paths

Running example


```
i := 0
while i < n do
    k := f(k, i)
    i := i + 1
```

```
i := n
while i ≥ 1 do
    k := f(k, n - i)
    i := i - 1

if n ≤ 0 then
    i := 0
else
    i := n
```

Running example: 1) Sequential composition

assume $v = \underline{v}$
$i := 0$

while $i < n$ **do**
    $k :=$ **f**(k, i)
    $i := i + 1$

$\underline{i} := \underline{n}$
while $\underline{i} \geq 1$ **do**
    $\underline{k} :=$ **f**$(\underline{k}, \underline{n} - \underline{i})$
    $\underline{i} := \underline{i} - 1$

if $\underline{n} \leq 0$ **then**
    $\underline{i} := 0$
else
    $\underline{i} := \underline{n}$

assert $v = \underline{v}$

Running example:
2) Eliminate UFs

```
assume v = v̲
i := 0

while i < n do
    k := f(k, i)
    i := i + 1


(…)

assert v = v̲
```

Running example:
3) Eliminate Loops


```
assume v = v̲
i := 0

while i < n do
    k := a·k + b·i + c
    i := i + 1

(…)

assert v = v̲
```

$$R_i(j) = R_i(j\text{-}1) + 1$$
$$R_i(0) = 0$$
$$R_k(j) = a \times R_k(j\text{-}1) + b \times R_i(j\text{-}1) + c$$
$$R_k(0) = k_0$$

$$R_i(j) = j$$

$$R_k(j) = \frac{b\left(a^j - aj + j - 1\right) + (a-1)\left(a^j\left((a-1)k_0 + c\right) - c\right)}{(a-1)^2}$$

Running example:
3) Eliminate Loops


```
assume v = v
i := 0

if i < n then
    assume R_i(j-1) < n ∧ R_i(j) ≥ n
    k := R_k(j)
    i := R_i(j)

i := n
if i ≥ 1 then
    assume V_i(j-1) ≥ 1 ∧ V_i(j) < 1
    k := V_k(j)
    i := V_i(j)

if n ≤ 0 then
    i := 0
else
    i := n

assert v = v
```

Outline

technology
from seed

- Program equivalence
- Example
- Algorithm
- Application to compiler optimizations
- Evaluation: CORK

Algorithm

1. Sequential composition
2. Replace UFs with polynomials
3. Replace loops with recurrences
4. Prove safety of resulting program

Algorithm:
1) Sequential Composition

assume v = $\underline{v}$

P1(v)

P2($\underline{v}$)

assert v = $\underline{v}$

Algorithm: Function u

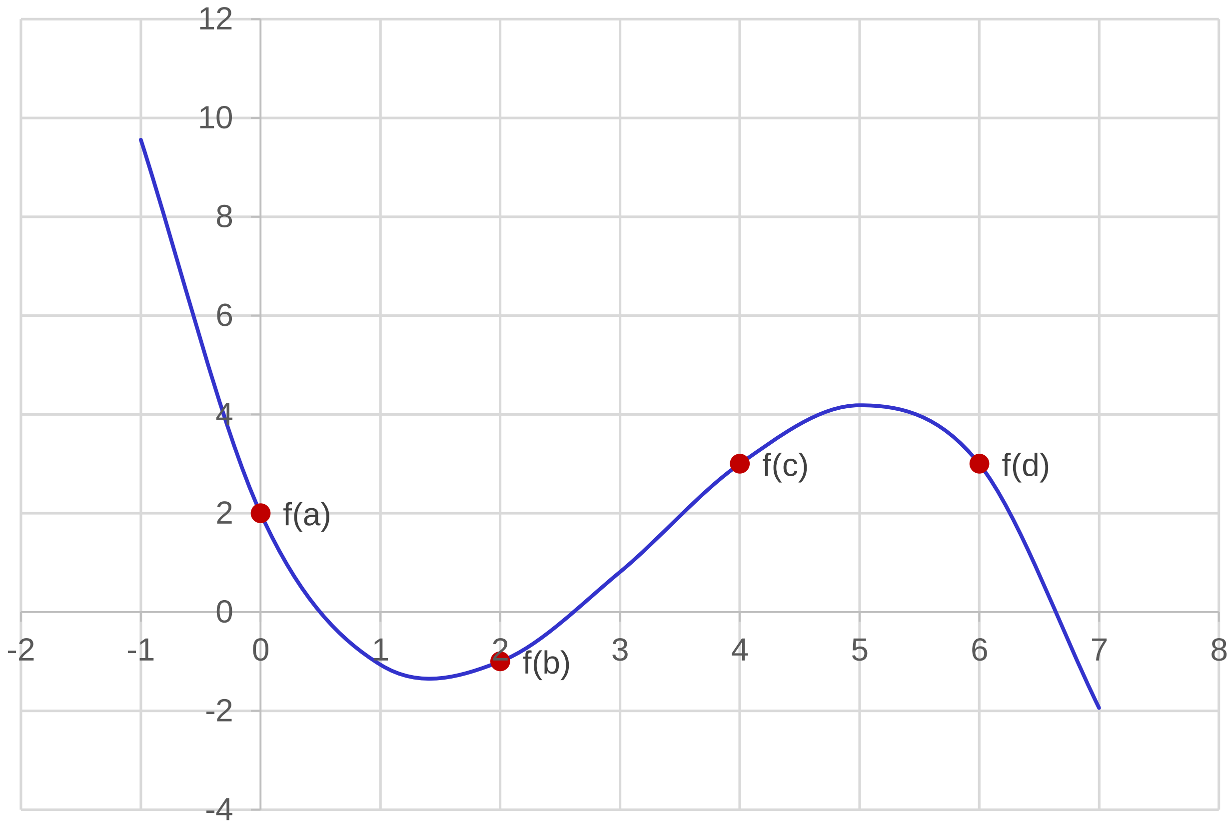
- u(*f*, *i*) is equal to the maximum number of applications of *f* with distinct values in the *i*th parameter in all paths minus one

```
if i < n then
   k := f(y, 3)
else
   k := f(z, 3)

if f(x, 3) < 0 ∧ k < 0 then
   …
```

$$u(f, 1) = 1$$
$$u(f, 2) = 0$$

Algorithm:
Polynomial Interpolation



Algorithm:
2) UF -> Polynomial

- UFs are rewritten to polynomials over its inputs

$$T(e) = \sum_{i=1}^{n} \sum_{j=0}^{u(\mathrm{UF},i)} \mathrm{UF_{ij}} \times \left(\mathrm{T(e_i)}\right)^{\mathrm{j}}, \qquad \text{if } e = \mathrm{UF}(e_1, \dots, e_n)$$

Algorithm:
3) Loops -> Recurrences

while b **do**
    c

$\longrightarrow$

if b **then**
    assume $\sigma_{n-1}(b) \wedge \sigma_n(\neg b)$
    $v_i := \sigma_n(v_i)$
else
    assume $n = 0$

Algorithm:
4) Prove safety of resulting program

- Resulting program is correct iff the 2 programs are partially equivalent
- Standard model checkers or VC gen + constraint solving can now prove correctness

Outline

technology
from seed

- Program equivalence
- Example
- Algorithm
- Application to compiler optimizations
- Evaluation: CORK

Compiler Optimizations

- Compiler optimization
  - Transformation function
  - Precondition
  - Profitability heuristic

Loop Unrolling


```
while I < N do
    S
    I := I + 1
```

⇒

```
while (I + 1) < N do
    S
    I := I + 1
    S
    I := I + 1

if I < N then
    S
    I := I + 1
```

<u>Precondition:</u>

$R(S) = \{I, N, c_1\}$

$W(S) = \{c_1\}$

Loop Unrolling: Example instantiation

```
while I < N do
    S := i + 2
    I := I + 1
```

⇒

```
while (I + 1) < N do
    S := i + 2
    I := I + 1
    S := i + 2
    I := I + 1

if I < N then
    S := i + 2
    I := I + 1
```

S	≡	x := i + 2
I	≡	i
N	≡	n

Compiler Optimizations: Our abstraction

- Transformation function specified as 2 template programs
- Precondition specified as read/write sets for template statements and expressions plus IA formulas

Transformation function to UF+IA Program

technology
from seed

- A transformation function can be written as two UF+IA programs
  - Template statements are converted to UFs, that read and write from/to their read/write sets

$$S \quad \rightarrow \quad x, y := S_x(y, z), S_y(y, z)$$

Precondition:
$R(S) = \{y, z\}$
$W(S) = \{x, y\}$

Outline

- Program equivalence
- Example
- Algorithm
- Application to compiler optimizations
- Evaluation: CORK

CORK: Compiler Optimization Correctness Checker

- Implemented in OCaml (~1,100 LoC)
- Uses Wolfram Mathematica 8 for constraint and recurrence solving

CORK: Results

technology
from seed

Optimization	PEC	# Sat. queries	# Recurrences	Time
Code hoisting	✓	2	0	0.32s
Constant propagation	✓	0	0	0.33s
Copy propagation	✓	0	0	0.33s
If-conversion	✓	2	0	0.34s
Partial redundancy elimin.	✓	2	0	0.34s
Loop invariant code motion	✓	7	5	3.48s
Loop peeling	✓	9	5	3.26s
Loop unrolling	✓	13	8	12.17s
Loop unswitching	✓	14	14	8.19s
Software pipelining	✓	9	5	8.02s
Loop fission	$\checkmark_p$	10	12	23.45s
Loop fusion	$\checkmark_p$	10	12	23.34s
Loop interchange	$\checkmark_p$	15	24	29.30s
Loop reversal	$\checkmark_p$	7	5	8.41s
Loop skewing	$\checkmark_p$	16	24	8.50s
Loop flattening	×	—	—	FAIL
Loop strength reduction	×	6	4	5.63s
Loop tiling	×	7	9	10.94s

Benchmarks available from http://web.ist.utl.pt/nuno.lopes/cork/

Conclusion

- Presented a new algorithm to prove equivalence of UF+IA programs
- Presented CORK, a compiler optimization verifier, that can prove more optimizations correct than others


technology
from seed
inesc id
lisboa