Definability of Accelerated Relations in a Theory of Arrays and its Applications

F. Alberti[1], S. Ghilardi[2], N. Sharygina[1]

[1]University of Lugano, Switzerland
[2] University of Milan, Italy

9th International Symposium on Frontiers of Combining Systems
September 18, 2013

Context: Reachability analysis

$$\mathcal{S}_T \;=\; (\; \mathbf{v} \;,\; I(\mathbf{v}) \;,\; \tau(\mathbf{v}, \mathbf{v}') \;)$$

- **Ingredients**: transition system $\mathcal{S}_T$ and a safety property $P(\mathbf{v})$
- **Reachability analysis**: establish if it is possible to reach $\neg P(\mathbf{v})$

Context: Reachability analysis

$$\mathcal{S}_T \ = \ (\ \mathbf{v} \ , \ I(\mathbf{v}) \ , \ \tau(\mathbf{v}, \mathbf{v}') \ )$$

- **Ingredients**: transition system $\mathcal{S}_T$ and a safety property $P(\mathbf{v})$
- **Reachability analysis**: establish if it is possible to reach $\neg P(\mathbf{v})$

⇒ T is Presburger arithmetic enriched with free function symbols

- satisfiability and validity with respect to structures having the standard structure of natural numbers as reduct
- $\mathbf{v}$ contains free unary function symbols ($\mathbf{a}$) and free constants ($\mathbf{c}$)

- We iteratively compute the preimage of $\neg P$ applying backward τ

Context: Reachability analysis

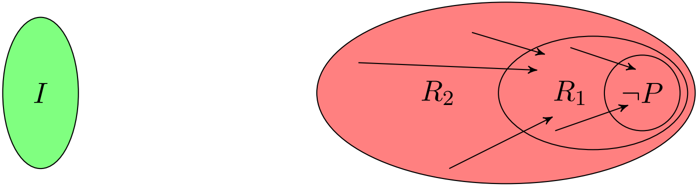
Backward search

- We iteratively compute the preimage of $\neg P$ applying backward τ

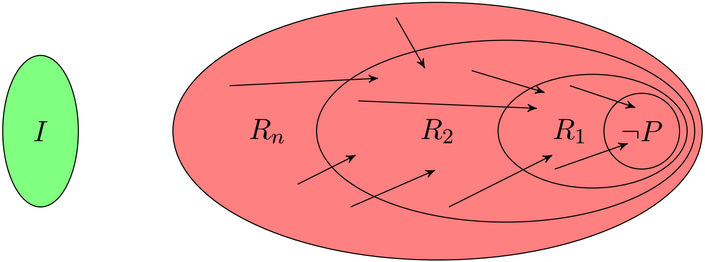
Context: Reachability analysis

Backward search

- We iteratively compute the preimage of $\neg P$ applying backward τ



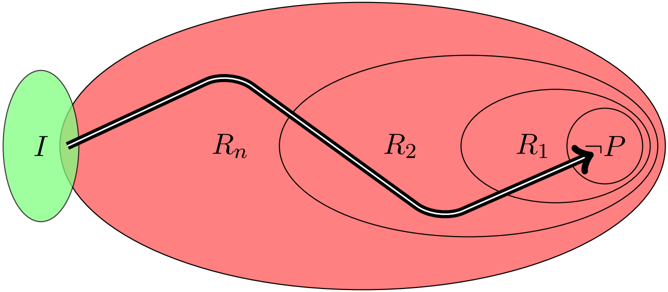
- We iteratively compute the preimage of $\neg P$ applying backward τ



Context: Reachability analysis

Backward search

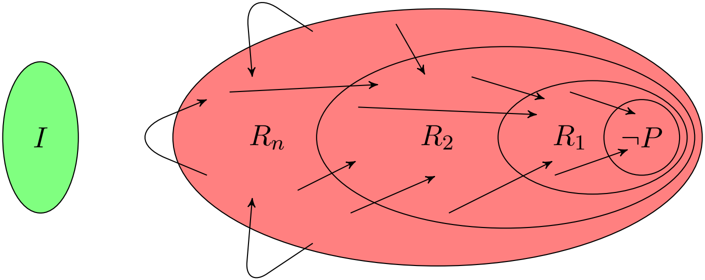
- We iteratively compute the preimage of $\neg P$ applying backward τ
- ... until we find an intersection with the set of initial states...



Context: Reachability analysis

Backward search

- We iteratively compute the preimage of $\neg P$ applying backward τ
- ... until we find an intersection with the set of initial states...
- ... or a (global) fix-point.

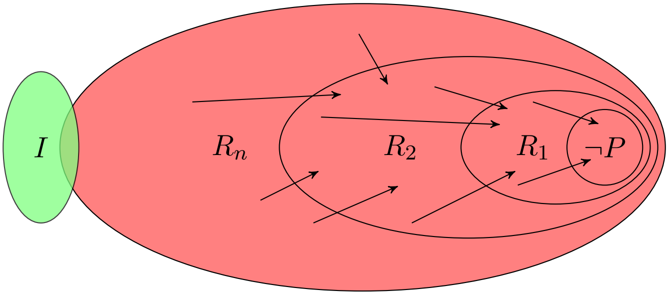


Context: Reachability analysis

Backward search

Reduce intersection and fix-point test to SMT problems:

- Intersection test: is $I \wedge R_n$ T-satisfiable?

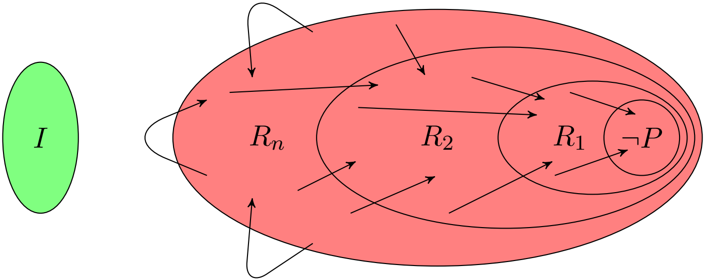


Context: Reachability analysis

Backward search

Reduce intersection and fix-point test to SMT problems:

- Intersection test: is $I \wedge R_n$ T-satisfiable?
- Fix-point test: is $R_{n+1} \rightarrow R_n$ T-valid?
- ...or dually: is $R_{n+1} \wedge \neg R_n$ T-unsatisfiable?



Context: Reachability analysis

Backward search - divergence

- Precise reachability analysis (usually) diverges on infinite-state systems

Context: Reachability analysis

Backward search - divergence

- Precise reachability analysis (usually) diverges on infinite-state systems
- Common experience with verification of annotated code

Context: Reachability analysis

Backward search - divergence

- Precise reachability analysis (usually) diverges on infinite-state systems

- Common experience with verification of annotated code

$\Rightarrow$ Acceleration can help in limiting divergence!

```
        procedure Find( int e ) {
l_I         i = 0;
l_L         while ( i < L ∧ a[i] ≠ e ) {
               i = i + 1;
            }
l_F         assert ( ∀x.(0 ≤ x < i) → a[x] ≠ e );
        }
```

[1] Assume we exit the loop because we reach the end of the array.

Acceleration

Example[1]

```
        procedure Find( int e ) {
l_I         i = 0;
l_L         while ( i < L ∧ a[i] ≠ e ) {
              i = i + 1;
            }
l_F         assert ( ∀x.(0 ≤ x < i) → a[x] ≠ e );
        }
```

$$\tau_1 := pc = l_L \quad \wedge \quad \underbrace{\mathtt{i} < \mathtt{L} \wedge \mathtt{a}[\mathtt{i}] \neq \mathtt{e}}_{\text{guard}} \quad \wedge \quad \underbrace{\mathtt{i}' = \mathtt{i} + 1}_{\text{update}}$$

[1] Assume we exit the loop because we reach the end of the array.

Acceleration

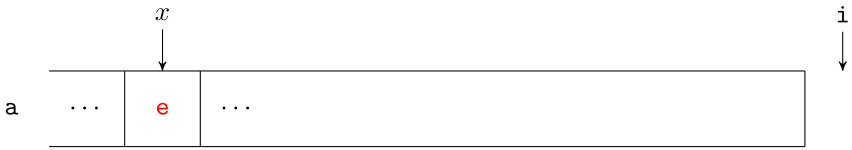
Example[1]

$$\tau_1 := pc = l_L \quad \wedge \quad \underbrace{\mathtt{i} < \mathtt{L} \wedge \mathtt{a}[\mathtt{i}] \neq \mathtt{e}}_{\text{guard}} \quad \wedge \quad \underbrace{\mathtt{i}' = \mathtt{i} + 1}_{\text{update}}$$

[1] Assume we exit the loop because we reach the end of the array.

Acceleration

Example[1]



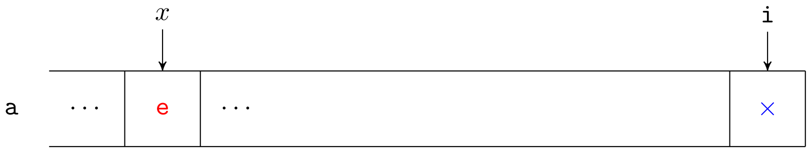
$$\exists x. 0 \leq x \wedge x < i \wedge a[x] = e \wedge i \geq L$$

$$\tau_1 := pc = l_L \quad \wedge \quad \underbrace{\mathtt{i} < \mathtt{L} \wedge \mathtt{a}[\mathtt{i}] \neq \mathtt{e}}_{\text{guard}} \quad \wedge \quad \underbrace{\mathtt{i}' = \mathtt{i} + 1}_{\text{update}}$$

[1] Assume we exit the loop because we reach the end of the array.

Acceleration

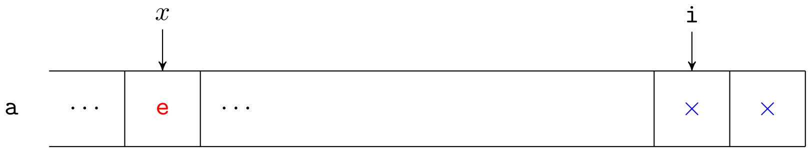
Example[1]



$\exists x. 0 \leq x \land x < i+1 \land a[x] = e \land i+1 = L \land$
$a[i] \neq e$

$$\tau_1 := pc = l_L \quad \land \quad \underbrace{\mathtt{i} < \mathtt{L} \land \mathtt{a}[\mathtt{i}] \neq \mathtt{e}}_{\text{guard}} \quad \land \quad \underbrace{\mathtt{i}' = \mathtt{i} + 1}_{\text{update}}$$

[1] Assume we exit the loop because we reach the end of the array.



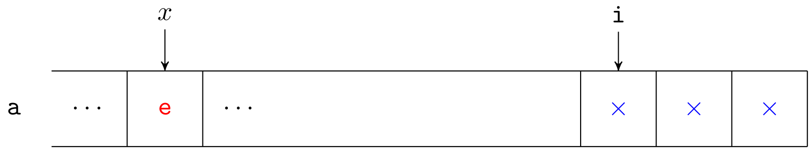
$$\exists x. 0 \leq x \land x < i + 2 \land a[x] = e \land i + 2 = L \land$$
$$a[i] \neq e \land a[i+1] \neq e$$

$$\tau_1 := pc = l_L \quad \land \quad \underbrace{\mathtt{i} < \mathtt{L} \land \mathtt{a}[\mathtt{i}] \neq \mathtt{e}}_{\text{guard}} \quad \land \quad \underbrace{\mathtt{i}' = \mathtt{i} + 1}_{\text{update}}$$

[1] Assume we exit the loop because we reach the end of the array.

Acceleration

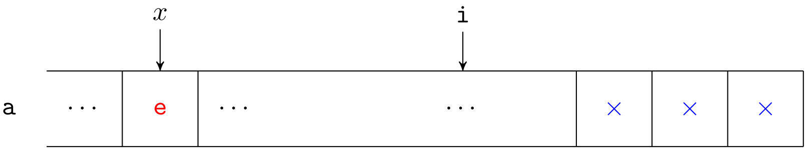
Example[1]



$$\exists x. 0 \leq x \wedge x < i + 3 \wedge a[x] = e \wedge i + 3 = L \wedge$$
$$a[i] \neq e \wedge a[i+1] \neq e \wedge a[i+2] \neq e$$

$$\tau_1 := pc = l_L \quad \wedge \quad \underbrace{\mathtt{i} < \mathtt{L} \wedge \mathtt{a}[\mathtt{i}] \neq \mathtt{e}}_{\text{guard}} \quad \wedge \quad \underbrace{\mathtt{i}' = \mathtt{i} + 1}_{\text{update}}$$

[1] Assume we exit the loop because we reach the end of the array.



$$\exists x. 0 \leq x \wedge x < i + n \wedge a[x] = e \wedge i + n = L \wedge$$

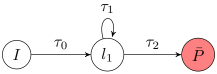
$$\bigwedge_{k=0}^{n-1} a[i+k] \neq e$$

$$\tau_1 := pc = l_L \quad \wedge \quad \underbrace{\mathtt{i} < \mathtt{L} \wedge \mathtt{a}[\mathtt{i}] \neq \mathtt{e}}_{\text{guard}} \quad \wedge \quad \underbrace{\mathtt{i}' = \mathtt{i} + 1}_{\text{update}}$$

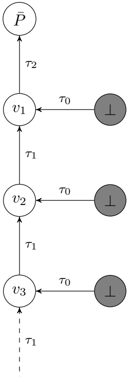
[1] Assume we exit the loop because we reach the end of the array.

Acceleration
Preventing divergence

Find control-flow graph:



Precise backward reachability



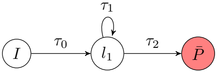
With accelerated transitions

(desired behavior)

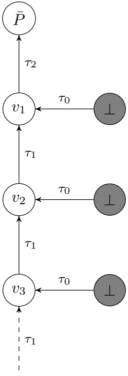
Acceleration

Preventing divergence

Find control-flow graph:



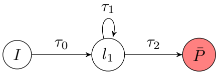
Precise backward reachability



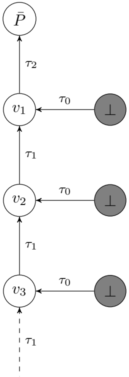
With accelerated transitions

(desired behavior)

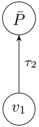
Find control-flow graph:



Precise backward reachability



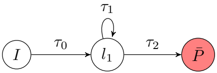
With accelerated transitions (desired behavior)



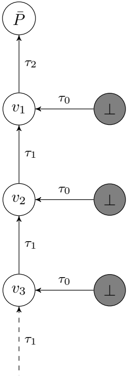
Acceleration

Preventing divergence

Find control-flow graph:

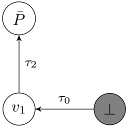


Precise backward reachability



With accelerated transitions

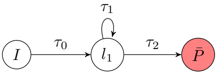
(desired behavior)



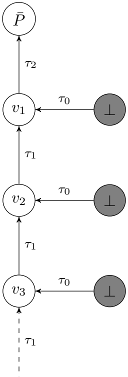
Acceleration

Preventing divergence

Find control-flow graph:

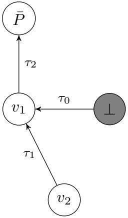


Precise backward reachability



With accelerated transitions

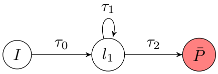
(desired behavior)



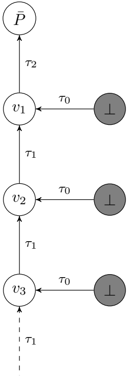
Acceleration

Preventing divergence

Find control-flow graph:

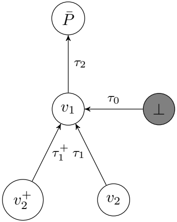


Precise backward reachability



With accelerated transitions

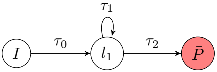
(desired behavior)



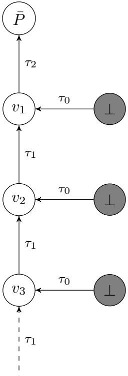
Acceleration

Preventing divergence

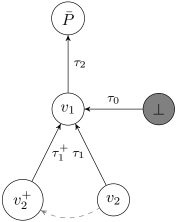
Find control-flow graph:



Precise backward reachability



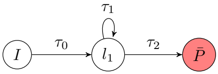
With accelerated transitions (desired behavior)



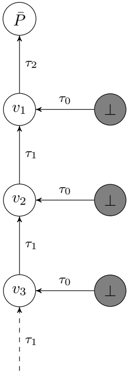
Acceleration

Preventing divergence

Find control-flow graph:

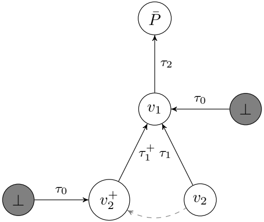


Precise backward reachability



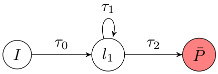
With accelerated transitions

(desired behavior)

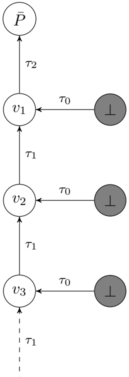


Acceleration
Preventing divergence

Find control-flow graph:

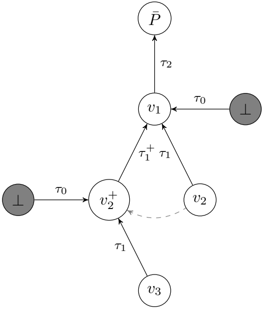


Precise backward reachability

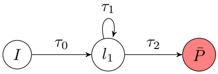


With accelerated transitions

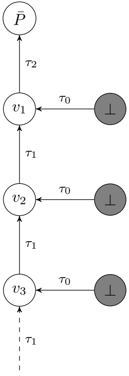
(desired behavior)



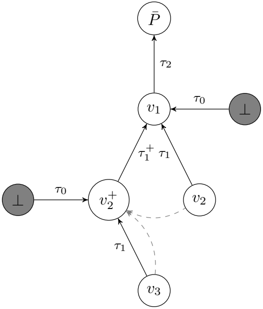
Find control-flow graph:



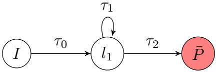
Precise backward reachability



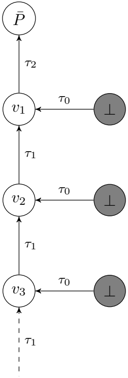
With accelerated transitions (desired behavior)



Find control-flow graph:

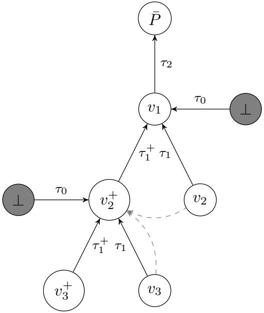


Precise backward reachability



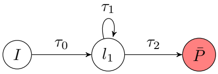
With accelerated transitions

(desired behavior)

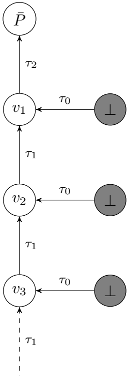


Acceleration

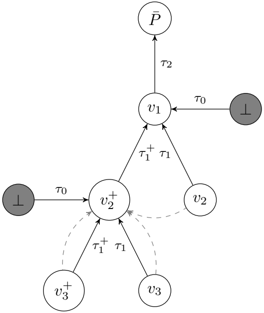
Preventing divergence

Find control-flow graph: 

Precise backward reachability



With accelerated transitions (desired behavior)



Acceleration: Transitive closure τ^+ of transitions τ encoding cyclic actions

Acceleration
State of the art

Acceleration: Transitive closure τ^+ of transitions τ encoding cyclic actions

Challenges:

- In general transitive closure cannot be expressed in FOL

Acceleration: Transitive closure τ^+ of transitions τ encoding cyclic actions

Challenges:

In general transitive closure cannot be expressed in FOL

Only some (important) classes of τ's allow the definability of τ^+

- Polling-based systems [BBD$^+$02]
- Imperative programs over integers [BIK10]

Acceleration

State of the art

Acceleration: Transitive closure τ^+ of transitions τ encoding cyclic actions

Challenges:

In general transitive closure cannot be expressed in FOL

Only some (important) classes of τ's allow the definability of τ^+

- Polling-based systems [BBD$^+$02]
- Imperative programs over integers [BIK10]

- What about arrays?

In theory:

- Identification of classes of transitions τ **over arrays** admitting definable acceleration

In theory:

- Identification of classes of transitions τ **over arrays** admitting definable acceleration
- Determine the **price** to pay for expressing τ^+

Acceleration for arrays

Contributions

In theory:

- Identification of classes of transitions τ **over arrays** admitting definable acceleration
- Determine the **price** to pay for expressing τ^+

In practice:

- Template-based solution
  - ✔ High degree of automation
  - ✔ Computationally cheap

Acceleration for arrays

Contributions

In theory:

- Identification of classes of transitions τ **over arrays** admitting definable acceleration
- Determine the **price** to pay for expressing τ^+

In practice:

- Template-based solution
  - ✔ High degree of automation
  - ✔ Computationally cheap
- Combination with abstraction-based frameworks

$$\tau_1 := pc = l_L \quad \wedge \quad \underbrace{\mathtt{i} < \mathtt{L} \wedge \mathtt{a}[\mathtt{i}] \neq \mathtt{e}}_{\text{guard}} \quad \wedge \quad \underbrace{\mathtt{i}' = \mathtt{i} + 1}_{\text{update}}$$

Acceleration for arrays

Example

$$\tau_1 := pc = l_L \quad \wedge \quad \underbrace{\mathtt{i} < \mathtt{L} \wedge \mathtt{a}[\mathtt{i}] \neq \mathtt{e}}_{\text{guard}} \quad \wedge \quad \underbrace{\mathtt{i}' = \mathtt{i} + 1}_{\text{update}}$$

$$\Downarrow$$

Acceleration for arrays

Example

$$\tau_1 := pc = l_L \quad \wedge \quad \underbrace{\mathtt{i} < \mathtt{L} \wedge \mathtt{a}[\mathtt{i}] \neq \mathtt{e}}_{\text{guard}} \quad \wedge \quad \underbrace{\mathtt{i}' = \mathtt{i} + 1}_{\text{update}}$$

$$\Downarrow$$

$$\tau_1^+ := \exists y. \begin{pmatrix} y > 0 \wedge pc = l_L \ \wedge \\ \forall j.(\ \mathtt{i} \leq j < \mathtt{i} + y \quad \rightarrow \quad j < \mathtt{L} \wedge \mathtt{a}[j] \neq \mathtt{e}\ ) \\ \mathtt{i}' = \mathtt{i} + y \end{pmatrix}$$

$$\tau_1 := pc = l_L \quad \wedge \quad \underbrace{\mathtt{i} < \mathtt{L} \wedge \mathtt{a}[\mathtt{i}] \neq \mathtt{e}}_{\text{guard}} \quad \wedge \quad \underbrace{\mathtt{i}' = \mathtt{i} + 1}_{\text{update}}$$

$$\Downarrow$$

$$\tau_1^+ := \exists y. \begin{pmatrix} y > 0 \wedge pc = l_L \ \wedge \\ \forall j.(\ \mathtt{i} \leq j < \mathtt{i} + y \quad \rightarrow \quad j < \mathtt{L} \wedge \mathtt{a}[j] \neq \mathtt{e}\ ) \\ \mathtt{i}' = \mathtt{i} + y \end{pmatrix}$$

Definition (Iterators)

A tuple of m-ary terms $\mathbf{u}(\underline{x})$ is said to be an *iterator* iff there exists an m-tuple of $m+1$-ary terms $\mathbf{u}^*(\underline{x}, y)$ such that for any natural number n it happens that the formula

$$\mathbf{u}^n(\underline{x}) = \mathbf{u}^*(\underline{x}, \bar{n})$$

is valid.

Definition (Iterators)

A tuple of m-ary terms $\mathbf{u}(\underline{x})$ is said to be an *iterator* iff there exists an m-tuple of $m+1$-ary terms $\mathbf{u}^*(\underline{x}, y)$ such that for any natural number n it happens that the formula

$$\mathbf{u}^n(\underline{x}) = \mathbf{u}^*(\underline{x}, \bar{n})$$

is valid.

Example

$\mathbf{u}(x) := x + 1$

The formal framework

Iterators

Definition (Iterators)

A tuple of m-ary terms $\mathbf{u}(\underline{x})$ is said to be an *iterator* iff there exists an m-tuple of $m+1$-ary terms $\mathbf{u}^*(\underline{x}, y)$ such that for any natural number n it happens that the formula

$$\mathbf{u}^n(\underline{x}) = \mathbf{u}^*(\underline{x}, \bar{n})$$

is valid.

Example

$\mathbf{u}(x) := x + 1$

$\mathbf{u}^*(x, y) := x + y$

Definition (Selectors)

Given an iterator $\mathbf{u}(\underline{x})$, an m-ary term $\kappa(x_1, \ldots, x_m)$ is a *selector* for $\mathbf{u}(\underline{x})$ iff there is an $m+1$-ary term $\iota(x_1, \ldots, x_m, y)$ yielding the validity of the formula

$$z = \kappa(\mathbf{u}^*(\underline{x}, y)) \to y = \iota(\underline{x}, z)$$

The formal framework

Selectors

Definition (Selectors)

Given an iterator $\mathbf{u}(\underline{x})$, an m-ary term $\kappa(x_1, \ldots, x_m)$ is a *selector* for $\mathbf{u}(\underline{x})$ iff there is an $m+1$-ary term $\iota(x_1, \ldots, x_m, y)$ yielding the validity of the formula

$$z = \kappa(\mathbf{u}^*(\underline{x}, y)) \to y = \iota(\underline{x}, z)$$

- Most likely κ is a projection

Definition (Selectors)

Given an iterator $\mathbf{u}(\underline{x})$, an m-ary term $\kappa(x_1, \ldots, x_m)$ is a *selector* for $\mathbf{u}(\underline{x})$ iff there is an $m+1$-ary term $\iota(x_1, \ldots, x_m, y)$ yielding the validity of the formula

$$z = \kappa(\mathbf{u}^*(\underline{x}, y)) \rightarrow y = \iota(\underline{x}, z)$$

- Most likely κ is a projection

- Can a cell z be reached in m iterations?
- The number $\iota(\underline{x}, z)$ gives "the only possible candidate" y number of iterations
- $z = \kappa(\mathbf{u}^*(\underline{x}, y))$ checks if the candidate y is correct

$$\texttt{while}\ (\ \texttt{true}\ )\ \{\ a[i] = 0; i = i + 2;\ \}$$

The formal framework
Example

$$\texttt{while (true) } \{\ a[i] = 0; i = i + 2;\ \}$$

- iterator: $u(i) := i + 2$

$$\texttt{while (true) } \{\ a[i] = 0; i = i + 2;\ \}$$

- iterator: $u(i) := i + 2 \qquad u^*(i, y) = i + 2y$

The formal framework
Example

$$\texttt{while (true) } \{ \ a[i] = 0; i = i + 2; \ \}$$

- iterator: $u(i) := i + 2 \qquad u^*(i, y) = i + 2y$
- $\kappa(x) := x$

The formal framework
Example

$$\texttt{while (true) } \{\ a[i] = 0; i = i + 2;\ \}$$

- iterator: $u(i) := i + 2 \qquad u^*(i, y) = i + 2y$
- $\kappa(x) := x$
- $\iota(i, z) := \left\lfloor \frac{z-i}{2} \right\rfloor$

$$\texttt{while (true) } \{\ a[i] = 0; i = i + 2;\ \}$$

- iterator: $u(i) := i + 2 \qquad u^*(i, y) = i + 2y$
- $\kappa(x) := x$
- $\iota(i, z) := \left\lfloor \frac{z-i}{2} \right\rfloor$

Example

- $i = 3$

The formal framework

Example

$$\texttt{while (true) } \{\ a[i] = 0; i = i + 2;\ \}$$

- iterator: $u(i) := i + 2$ $\quad u^*(i, y) = i + 2y$
- $\kappa(x) := x$
- $\iota(i, z) := \left\lfloor \frac{z-i}{2} \right\rfloor$

Example

- $i = 3$
- $a[7]$ in 3 iterations?

The formal framework
Example

$$\texttt{while (true) } \{\ a[i] = 0; i = i + 2;\ \}$$

- iterator: $u(i) := i + 2 \qquad u^*(i, y) = i + 2y$
- $\kappa(x) := x$
- $\iota(i, z) := \left\lfloor \frac{z-i}{2} \right\rfloor$

Example

- $i = 3$
- $a[7]$ in 3 iterations?
- $\iota(i, z) = \left\lfloor \frac{7-3}{2} \right\rfloor$

The formal framework
Example

$$\texttt{while (true) } \{\ a[i] = 0; i = i + 2;\ \}$$

- iterator: $u(i) := i + 2$ $\qquad u^*(i, y) = i + 2y$
- $\kappa(x) := x$
- $\iota(i, z) := \left\lfloor \frac{z-i}{2} \right\rfloor$

Example

- $i = 3$
- $a[7]$ in 3 iterations?
- $\iota(i, z) = \left\lfloor \frac{7-3}{2} \right\rfloor = 2$

The formal framework

Example

$$\texttt{while (true) } \{\ a[i] = 0; i = i + 2;\ \}$$

- iterator: $u(i) := i + 2 \qquad u^*(i, y) = i + 2y$
- $\kappa(x) := x$
- $\iota(i, z) := \left\lfloor \frac{z-i}{2} \right\rfloor$

Example

- $i = 3$
- $a[7]$ in 3 iterations?
- $\iota(i, z) = \left\lfloor \frac{7-3}{2} \right\rfloor = 2$ ✔

$$\texttt{while (true) } \{\ a[i] = 0; i = i + 2;\ \}$$

- iterator: $u(i) := i + 2 \qquad u^*(i, y) = i + 2y$
- $\kappa(x) := x$
- $\iota(i, z) := \left\lfloor \frac{z-i}{2} \right\rfloor$

Example

- $i = 3$
- $a[7]$ in 3 iterations?
- $\iota(i, z) = \left\lfloor \frac{7-3}{2} \right\rfloor = 2$ ✔
- $u^*(i, 2) = 3 + 2 \cdot 2 = 7$

$$\texttt{while (true) } \{\ a[i] = 0; i = i + 2;\ \}$$

- iterator: $u(i) := i + 2$ $u^*(i, y) = i + 2y$
- $\kappa(x) := x$
- $\iota(i, z) := \left\lfloor \frac{z-i}{2} \right\rfloor$

Example

- $i = 3$
- $a[7]$ in 3 iterations?
- $\iota(i, z) = \left\lfloor \frac{7-3}{2} \right\rfloor = 2$ ✔
- $u^*(i, 2) = 3 + 2 \cdot 2 = 7$ ✔

The formal framework

Example

$$\texttt{while (true) } \{\ a[i] = 0; i = i + 2;\ \}$$

- iterator: $u(i) := i + 2$ $\qquad u^*(i, y) = i + 2y$
- $\kappa(x) := x$
- $\iota(i, z) := \left\lfloor \frac{z-i}{2} \right\rfloor$

Example

- $i = 3$
- $a[7]$ in 3 iterations?
- $\iota(i, z) = \left\lfloor \frac{7-3}{2} \right\rfloor = 2$ ✔
- $u^*(i, 2) = 3 + 2 \cdot 2 = 7$ ✔

- $i = 3$

The formal framework

Example

$$\texttt{while (true) } \{\ a[i] = 0; i = i + 2;\ \}$$

- iterator: $u(i) := i + 2 \qquad u^*(i, y) = i + 2y$
- $\kappa(x) := x$
- $\iota(i, z) := \left\lfloor \frac{z-i}{2} \right\rfloor$

Example

- $i = 3$
- $a[7]$ in 3 iterations?
- $\iota(i, z) = \left\lfloor \frac{7-3}{2} \right\rfloor = 2$ ✔
- $u^*(i, 2) = 3 + 2 \cdot 2 = 7$ ✔

- $i = 3$
- $a[6]$ in 3 iterations?

$$\texttt{while (true) } \{\ a[i] = 0; i = i + 2;\ \}$$

- iterator: $u(i) := i + 2$ $\quad u^*(i, y) = i + 2y$
- $\kappa(x) := x$
- $\iota(i, z) := \left\lfloor \frac{z-i}{2} \right\rfloor$

Example

- $i = 3$
- $a[7]$ in 3 iterations?
- $\iota(i, z) = \left\lfloor \frac{7-3}{2} \right\rfloor = 2$ ✔
- $u^*(i, 2) = 3 + 2 \cdot 2 = 7$ ✔

- $i = 3$
- $a[6]$ in 3 iterations?
- $\iota(i, z) = \left\lfloor \frac{6-3}{2} \right\rfloor$

The formal framework

Example

$$\texttt{while (true) } \{\ a[i] = 0; i = i + 2;\ \}$$

- iterator: $u(i) := i + 2$ $\quad u^*(i, y) = i + 2y$
- $\kappa(x) := x$
- $\iota(i, z) := \left\lfloor \frac{z-i}{2} \right\rfloor$

Example

- $i = 3$
- $a[7]$ in 3 iterations?
- $\iota(i, z) = \left\lfloor \frac{7-3}{2} \right\rfloor = 2$ ✔
- $u^*(i, 2) = 3 + 2 \cdot 2 = 7$ ✔

- $i = 3$
- $a[6]$ in 3 iterations?
- $\iota(i, z) = \left\lfloor \frac{6-3}{2} \right\rfloor = 1$

The formal framework
Example

$$\texttt{while (true) } \{\ a[i] = 0; i = i + 2;\ \}$$

- iterator: $u(i) := i + 2$ $\quad u^*(i, y) = i + 2y$
- $\kappa(x) := x$
- $\iota(i, z) := \left\lfloor \frac{z-i}{2} \right\rfloor$

Example

- $i = 3$
- $a[7]$ in 3 iterations?
- $\iota(i, z) = \left\lfloor \frac{7-3}{2} \right\rfloor = 2$ ✔
- $u^*(i, 2) = 3 + 2 \cdot 2 = 7$ ✔

- $i = 3$
- $a[6]$ in 3 iterations?
- $\iota(i, z) = \left\lfloor \frac{6-3}{2} \right\rfloor = 1$ ✔

The formal framework
Example

$$\texttt{while (true) } \{\ a[i] = 0; i = i + 2;\ \}$$

- iterator: $u(i) := i + 2$ $u^*(i, y) = i + 2y$
- $\kappa(x) := x$
- $\iota(i, z) := \left\lfloor \frac{z-i}{2} \right\rfloor$

Example

- $i = 3$
- $a[7]$ in 3 iterations?
- $\iota(i, z) = \left\lfloor \frac{7-3}{2} \right\rfloor = 2$ ✔
- $u^*(i, 2) = 3 + 2 \cdot 2 = 7$ ✔

- $i = 3$
- $a[6]$ in 3 iterations?
- $\iota(i, z) = \left\lfloor \frac{6-3}{2} \right\rfloor = 1$ ✔
- $u^*(i, 1) = 3 + 2 \cdot 1 = 5$

The formal framework

Example

$$\texttt{while (true) } \{\ a[i] = 0; i = i + 2;\ \}$$

- iterator: $u(i) := i + 2 \qquad u^*(i, y) = i + 2y$
- $\kappa(x) := x$
- $\iota(i, z) := \left\lfloor \frac{z-i}{2} \right\rfloor$

Example

- $i = 3$
- $a[7]$ in 3 iterations?
- $\iota(i, z) = \left\lfloor \frac{7-3}{2} \right\rfloor = 2$ ✔
- $u^*(i, 2) = 3 + 2 \cdot 2 = 7$ ✔

- $i = 3$
- $a[6]$ in 3 iterations?
- $\iota(i, z) = \left\lfloor \frac{6-3}{2} \right\rfloor = 1$ ✔
- $u^*(i, 1) = 3 + 2 \cdot 1 = 5$ ✘

Definition (Local ground assignment)

A *local ground assignment* is a ground assignment of the form

$$
\begin{aligned}
& pc = l \ \wedge \ \phi_L(\mathbf{a}, \mathbf{c}) \ \wedge \ pc' = l \ \wedge \\
& \mathbf{a}' = wr(\mathbf{a}, \kappa(\tilde{\mathbf{c}}), \mathbf{t}(\mathbf{a}, \mathbf{c})) \ \wedge \ \tilde{\mathbf{c}}' = \mathbf{u}(\tilde{\mathbf{c}}) \ \wedge \mathbf{d}' = \mathbf{d}
\end{aligned}
$$

Definition (Local ground assignment)

A *local ground assignment* is a ground assignment of the form

$$
\begin{aligned}
& pc = l \ \wedge \ \phi_L(\mathbf{a}, \mathbf{c}) \ \wedge \ pc' = l \ \wedge \\
& \mathbf{a}' = wr(\mathbf{a}, \kappa(\tilde{\mathbf{c}}), \mathbf{t}(\mathbf{a}, \mathbf{c})) \ \wedge \ \tilde{\mathbf{c}}' = \mathbf{u}(\tilde{\mathbf{c}}) \ \wedge \mathbf{d}' = \mathbf{d}
\end{aligned}
$$

where

(i) $\mathbf{c} = \tilde{\mathbf{c}}, \mathbf{d}$;

Definition (Local ground assignment)

A *local ground assignment* is a ground assignment of the form

$$pc = l \ \wedge \ \phi_L(\mathbf{a}, \mathbf{c}) \ \wedge \ pc' = l \ \wedge$$
$$\mathbf{a}' = wr(\mathbf{a}, \kappa(\tilde{\mathbf{c}}), \mathbf{t}(\mathbf{a}, \mathbf{c})) \ \wedge \ \tilde{\mathbf{c}}' = \mathbf{u}(\tilde{\mathbf{c}}) \ \wedge \mathbf{d}' = \mathbf{d}$$

where

(i) $\mathbf{c} = \tilde{\mathbf{c}}, \mathbf{d}$;

(ii) $\mathbf{u} = u_1, \ldots, u_{|\tilde{\mathbf{c}}|}$ is an iterator;

Definition (Local ground assignment)

A *local ground assignment* is a ground assignment of the form

$$pc = l \ \wedge \ \phi_L(\mathbf{a}, \mathbf{c}) \ \wedge \ pc' = l \ \wedge$$
$$\mathbf{a}' = wr(\mathbf{a}, \kappa(\tilde{\mathbf{c}}), \mathbf{t}(\mathbf{a}, \mathbf{c})) \ \wedge \ \tilde{\mathbf{c}}' = \mathbf{u}(\tilde{\mathbf{c}}) \ \wedge \mathbf{d}' = \mathbf{d}$$

where

(i) $\mathbf{c} = \tilde{\mathbf{c}}, \mathbf{d}$;

(ii) $\mathbf{u} = u_1, \ldots, u_{|\tilde{\mathbf{c}}|}$ is an iterator;

(iii) the terms κ are a selector assignment for $\mathbf{a}$ relative to $\mathbf{u}$;

Definition (Local ground assignment)

A *local ground assignment* is a ground assignment of the form

$$pc = l \ \wedge \ \phi_L(\mathbf{a}, \mathbf{c}) \ \wedge \ pc' = l \ \wedge$$
$$\mathbf{a}' = wr(\mathbf{a}, \kappa(\tilde{\mathbf{c}}), \mathbf{t}(\mathbf{a}, \mathbf{c})) \ \wedge \ \tilde{\mathbf{c}}' = \mathbf{u}(\tilde{\mathbf{c}}) \ \wedge \mathbf{d}' = \mathbf{d}$$

where

(i) $\mathbf{c} = \tilde{\mathbf{c}}, \mathbf{d}$;

(ii) $\mathbf{u} = u_1, \ldots, u_{|\tilde{\mathbf{c}}|}$ is an iterator;

(iii) the terms κ are a selector assignment for $\mathbf{a}$ relative to $\mathbf{u}$;

(iv) the formula $\phi_L(\mathbf{a}, \mathbf{c})$ and the terms $\mathbf{t}(\mathbf{a}, \mathbf{c})$ are *purely arithmetical* over the set of terms $\{\mathbf{c}, \mathbf{a}(\kappa(\tilde{\mathbf{c}}))\} \cup \{a_i(d_j)\}_{1 \le i \le s, 1 \le j \le |\mathbf{d}|}$;

Definition (Local ground assignment)

A *local ground assignment* is a ground assignment of the form

$$pc = l \;\wedge\; \phi_L(\mathbf{a}, \mathbf{c}) \;\wedge\; pc' = l \;\wedge$$
$$\mathbf{a}' = wr(\mathbf{a}, \kappa(\tilde{\mathbf{c}}), \mathbf{t}(\mathbf{a}, \mathbf{c})) \;\wedge\; \tilde{\mathbf{c}}' = \mathbf{u}(\tilde{\mathbf{c}}) \;\wedge \mathbf{d}' = \mathbf{d}$$

where

(i) $\mathbf{c} = \tilde{\mathbf{c}}, \mathbf{d}$;

(ii) $\mathbf{u} = u_1, \dots, u_{|\tilde{\mathbf{c}}|}$ is an iterator;

(iii) the terms κ are a selector assignment for $\mathbf{a}$ relative to $\mathbf{u}$;

(iv) the formula $\phi_L(\mathbf{a}, \mathbf{c})$ and the terms $\mathbf{t}(\mathbf{a}, \mathbf{c})$ are *purely arithmetical* over the set of terms $\{\mathbf{c}, \mathbf{a}(\kappa(\tilde{\mathbf{c}}))\} \cup \{a_i(d_j)\}_{1 \le i \le s, 1 \le j \le |\mathbf{d}|}$;

(v) the guard ϕ_L contains the conjuncts $\kappa_i(\tilde{\mathbf{c}}) \neq d_j$, for $1 \le i \le s$ and $1 \le j \le |\mathbf{d}|$.

The formal framework

Contribution

Theorem

If τ is a local ground assignment, then τ^+ is a Σ^0_2-assignment.

Francesco Alberti, Silvio Ghilardi, and Natasha Sharygina.
Tackling divergence: abstraction and acceleration in array programs.
Technical Report 2012/01, University of Lugano, oct 2012.

The formal framework
Contribution

Theorem
If τ is a local ground assignment, then τ^+ is a Σ_2^0-assignment.

Francesco Alberti, Silvio Ghilardi, and Natasha Sharygina.
Tackling divergence: abstraction and acceleration in array programs.
Technical Report 2012/01, University of Lugano, oct 2012.

- The proof of the theorem shows the "template" for τ^+

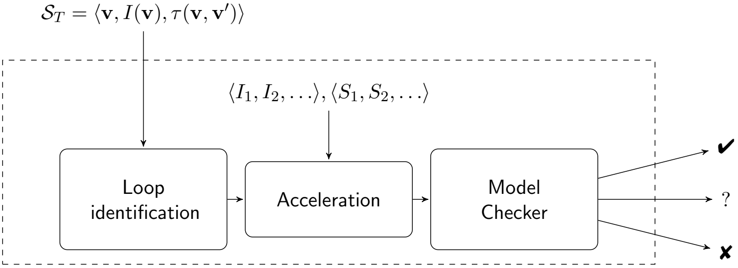
Theorem

If τ is a local ground assignment, then τ^+ is a Σ^0_2-assignment.

Francesco Alberti, Silvio Ghilardi, and Natasha Sharygina.
Tackling divergence: abstraction and acceleration in array programs.
Technical Report 2012/01, University of Lugano, oct 2012.

- The proof of the theorem shows the "template" for τ^+
- The template is parametric with respect to
  - iterators
  - selectors

Tool architecture



Acceleration for arrays

Practical issue - classification of formulas

Different kind of formulas[2] representing the (backward reachable) state-space:

- *ground* – formulas of the kind $\phi(\mathbf{v})$

[2] In all the formulas we admit the term $a(t)$ only if t is a variable or a constant.

Acceleration for arrays

Practical issue - classification of formulas

Different kind of formulas[2] representing the (backward reachable) state-space:

- *ground* – formulas of the kind $\phi(\mathbf{v})$
- Σ_1^0 – formulas of the kind $\exists \underline{i}.\phi(\underline{i}, \mathbf{v})$

[2] In all the formulas we admit the term $a(t)$ only if t is a variable or a constant.

Different kind of formulas[2] representing the (backward reachable) state-space:

- *ground* – formulas of the kind $\phi(\mathbf{v})$
- Σ_1^0 – formulas of the kind $\exists \underline{i}.\phi(\underline{i}, \mathbf{v})$
- Σ_2^0 – formulas of the kind $\exists \underline{i} \forall \underline{j}.\phi(\underline{i}, \underline{j}, \mathbf{v})$

[2] In all the formulas we admit the term $a(t)$ only if t is a variable or a constant.

Acceleration for arrays

Practical issue - classification of formulas

Different kind of formulas[2] representing the (backward reachable) state-space:

- *ground* – formulas of the kind $\phi(\mathbf{v})$
- Σ^0_1 – formulas of the kind $\exists \underline{i}.\phi(\underline{i}, \mathbf{v})$
- Σ^0_2 – formulas of the kind $\exists \underline{i} \forall \underline{j}.\phi(\underline{i}, \underline{j}, \mathbf{v})$

Σ^0_2-formulas might not fall in any known decidable fragment [BMS06, GdM09]

[2] In all the formulas we admit the term $a(t)$ only if t is a variable or a constant.

Transition formulas can be:

- *ground* assignment – transitions of the kind $\tau(\mathbf{v}, \mathbf{v}')$
- Σ_1^0-assignment – transitions of the kind $\exists \underline{i}.\tau(\underline{i}, \mathbf{v}, \mathbf{v}')$
- Σ_2^0-assignment – transitions of the kind $\exists \underline{i} \forall \underline{j}.\tau(\underline{i}, \underline{j}, \mathbf{v}, \mathbf{v}')$

Transition formulas can be:

- *ground* assignment – transitions of the kind $\tau(\mathbf{v}, \mathbf{v}')$
- Σ^0_1-assignment – transitions of the kind $\exists \underline{i}.\tau(\underline{i}, \mathbf{v}, \mathbf{v}')$
- Σ^0_2-assignment – transitions of the kind $\exists \underline{i} \forall \underline{j}.\tau(\underline{i}, \underline{j}, \mathbf{v}, \mathbf{v}')$

- Preimages with respect to a Σ^0_2-assignment are Σ^0_2-formulas

Acceleration for arrays

Practical issue - classification of transitions

Transition formulas can be:

- *ground* assignment – transitions of the kind $\tau(\mathbf{v}, \mathbf{v}')$
- Σ^0_1-assignment – transitions of the kind $\exists \underline{i}.\tau(\underline{i}, \mathbf{v}, \mathbf{v}')$
- Σ^0_2-assignment – transitions of the kind $\exists \underline{i} \forall \underline{j}.\tau(\underline{i}, \underline{j}, \mathbf{v}, \mathbf{v}')$

- Preimages with respect to a Σ^0_2-assignment are Σ^0_2-formulas
- This prevents the practical application of the theoretical result!

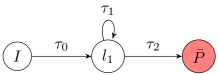
Transition formulas can be:

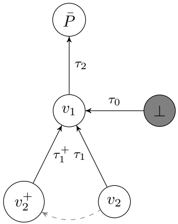
- *ground* assignment – transitions of the kind $\tau(\mathbf{v}, \mathbf{v}')$
- Σ_1^0-assignment – transitions of the kind $\exists \underline{i}.\tau(\underline{i}, \mathbf{v}, \mathbf{v}')$
- Σ_2^0-assignment – transitions of the kind $\exists \underline{i} \forall \underline{j}.\tau(\underline{i}, \underline{j}, \mathbf{v}, \mathbf{v}')$

- Preimages with respect to a Σ_2^0-assignment are Σ_2^0-formulas
- This prevents the practical application of the theoretical result!
- Solution: over-approximate problematic Σ_2^0-formulas with their *monotonic abstraction* [AGP+12]

Acceleration for arrays

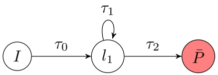
Example

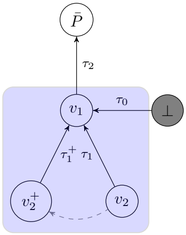




Acceleration for arrays

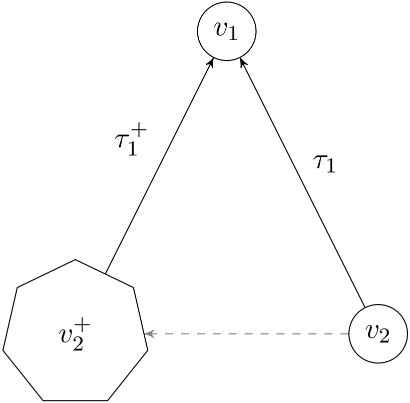
Example





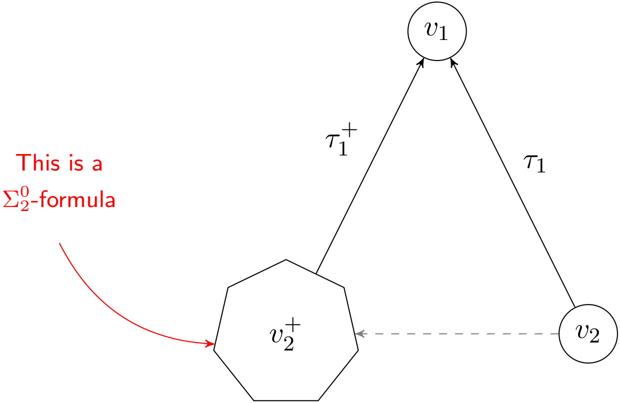
Acceleration for arrays

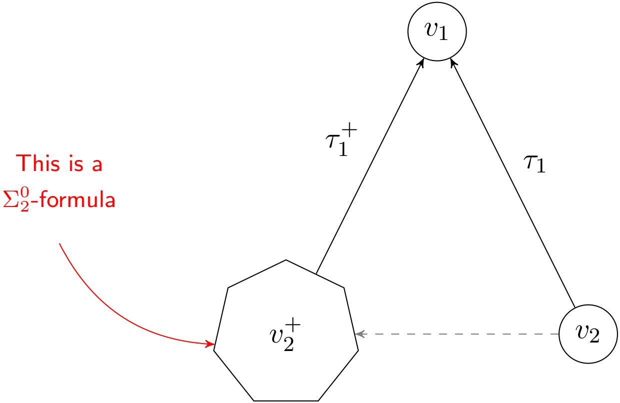
Example



Acceleration for arrays

Example

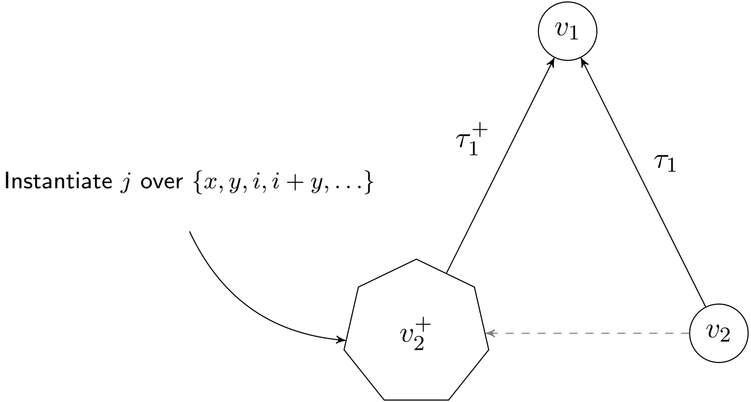




$$\exists x, y \forall j. \begin{pmatrix} pc = l_L \land y > 0 \land \\ (i \leq j < i + y \rightarrow j < L \land a[j] \neq e) \land \\ 0 \leq x < i \land a[x] = e \land i + y \geq L \end{pmatrix}$$

Acceleration for arrays

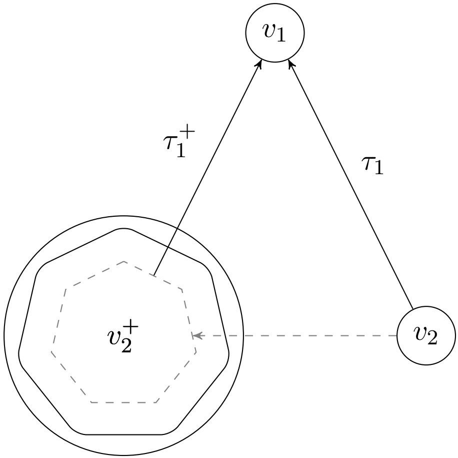
Example



$$\exists x, y \forall j. \begin{pmatrix} pc = l_L \wedge y > 0 \wedge \\ (i \leq j < i + y \rightarrow j < L \wedge a[j] \neq e) \wedge \\ 0 \leq x < i \wedge a[x] = e \wedge i + y \geq L \end{pmatrix}$$

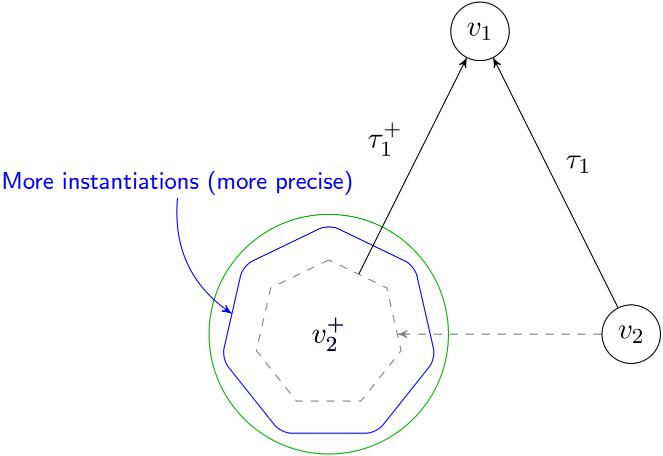
Acceleration for arrays

Example

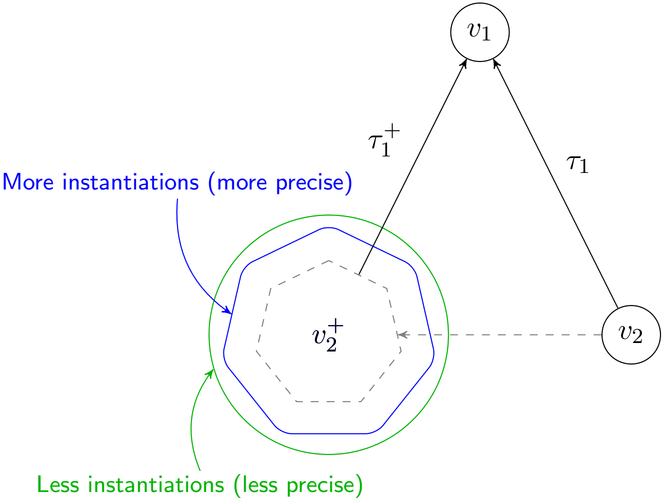


Acceleration for arrays

Example

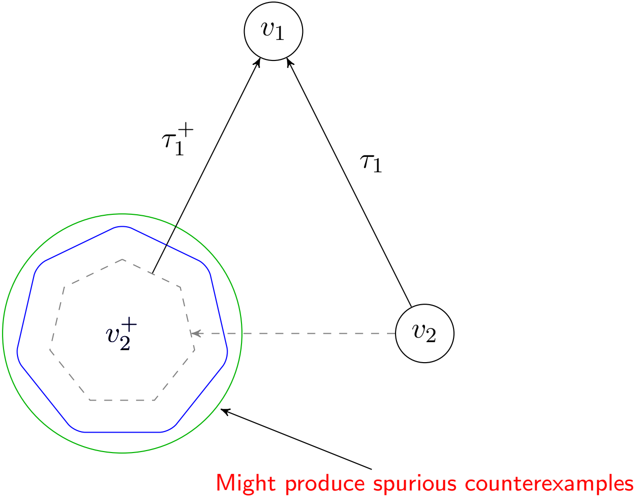


Acceleration for arrays
Example



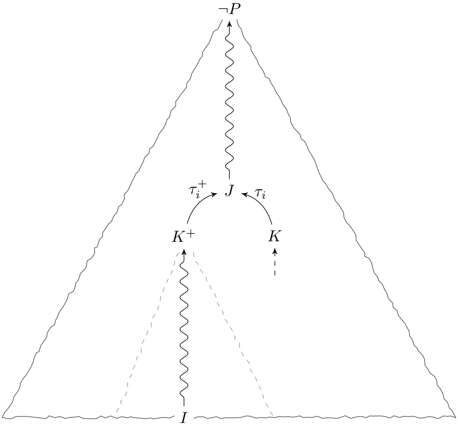
Acceleration for arrays

Example



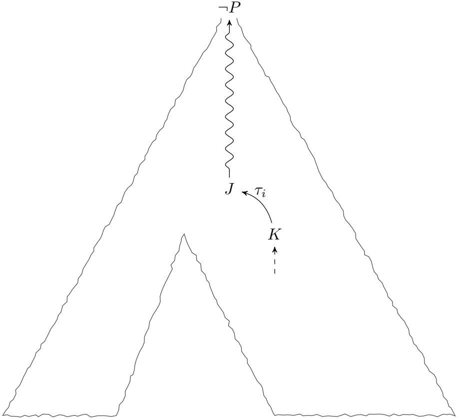
Acceleration for arrays

Ad-hoc refinement for monotonic abstraction



Acceleration for arrays

Ad-hoc refinement for monotonic abstraction



Acceleration for arrays
Experiments

- Implemented in the MCMT model checker

Acceleration for arrays
Experiments

- Implemented in the MCMT model checker

- Tested on 55 challenging benchmarks on arrays
  - initializing
  - searching
  - sorting
  - etc.

Acceleration for arrays

Experiments

```
function allDiff ( int a[N] ) :
r = true;
for (i = 1; i < N ∧ r; i++)
    for (j = i-1; j ≥ 0 ∧ r; j--)
        if (a[i] = a[j]) r = false;
assert (r → (∀x, y(0 ≤ x < y < N) → (a[x] ≠ a[y])))
```

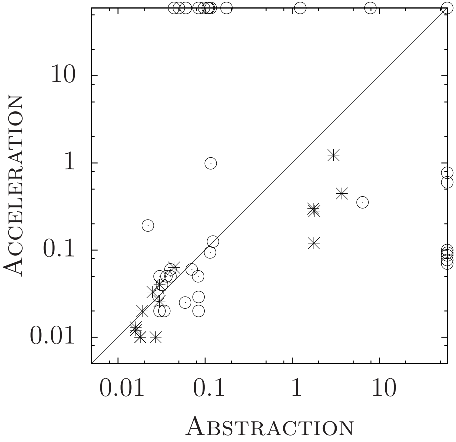
Acceleration for arrays

Experiments

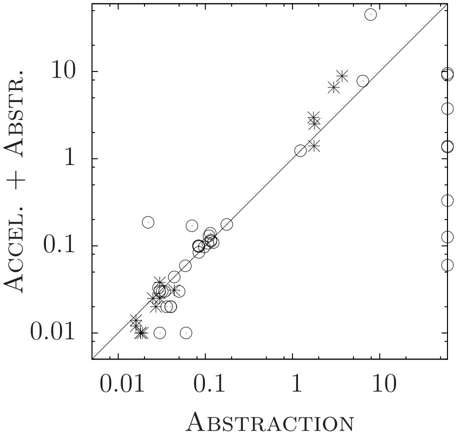
```
function allDiff ( int a[N] ) :
r = true;
for (i = 1; i < N ∧ r; i++)
    for (j = i-1; j ≥ 0 ∧ r; j--)
        if (a[i] = a[j]) r = false;
assert (r → (∀x, y(0 ≤ x < y < N) → (a[x] ≠ a[y])))
```

Acceleration for arrays

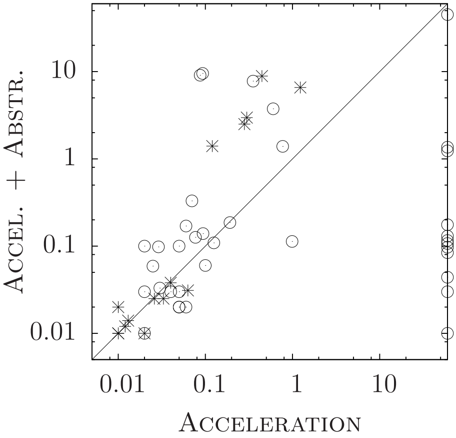
Experiments



MCMT running time



MCMT running time



MCMT running time

Conclusion

- Accelerations of *local ground assignments* are Σ_2^0-assignments

Conclusion

- Accelerations of *local ground assignments* are Σ^0_2-assignments
- Template-based computation of τ^+
  - High degree of automation
  - Computationally cheap

Conclusion

- Accelerations of *local ground assignments* are Σ_2^0-assignments
- Template-based computation of τ^+
  - High degree of automation
  - Computationally cheap
- *monotonic abstraction* to over-approximate problematic preimages with respect to accelerated transitions

Conclusion

- Accelerations of *local ground assignments* are Σ^0_2-assignments
- Template-based computation of τ^+
  - High degree of automation
  - Computationally cheap
- *monotonic abstraction* to over-approximate problematic preimages with respect to accelerated transitions
- Experimental evidence that acceleration and abstraction are mutually beneficial

Conclusion

- Accelerations of *local ground assignments* are Σ^0_2-assignments
- Template-based computation of τ^+
  - High degree of automation
  - Computationally cheap
- *monotonic abstraction* to over-approximate problematic preimages with respect to accelerated transitions
- Experimental evidence that acceleration and abstraction are mutually beneficial

Thank you! Questions?

References I

Francesco Alberti, Silvio Ghilardi, Elena Pagani, Silvio Ranise, and Gian Paolo Rossi.
Universal guards, relativization of quantifiers, and failure models in Model Checking Modulo Theories.
JSAT, 8(1/2):29–61, 2012.

Francesco Alberti, Silvio Ghilardi, and Natasha Sharygina.
Tackling divergence: abstraction and acceleration in array programs.
Technical Report 2012/01, University of Lugano, oct 2012.

References II

Gerd Behrmann, Johan Bengtsson, Alexandre David, Kim G. Larsen, Paul Pettersson, and Wang Yi.
UPPAAL implementation secrets.
In Werner Damm and Ernst-Rüdiger Olderog, editors, *FTRTFT*, volume 2469 of *Lecture Notes in Computer Science*, pages 3–22. Springer, 2002.

Marius Bozga, Radu Iosif, and Filip Konecný.
Fast acceleration of ultimately periodic relations.
In Tayssir Touili, Byron Cook, and Paul Jackson, editors, *CAV*, volume 6174 of *Lecture Notes in Computer Science*, pages 227–242. Springer, 2010.

References III

Aaron R. Bradley, Zohar Manna, and Henny B. Sipma.
What's decidable about arrays?
In E. Allen Emerson and Kedar S. Namjoshi, editors, *VMCAI*, volume 3855 of *Lecture Notes in Computer Science*, pages 427–442. Springer, 2006.

Yeting Ge and Leonardo M. de Moura.
Complete instantiation for quantified formulas in satisfiabiliby modulo theories.
In Ahmed Bouajjani and Oded Maler, editors, *CAV*, volume 5643 of *Lecture Notes in Computer Science*, pages 306–320. Springer, 2009.