Discovering Properties about Arrays in Simple Programs

Nicolas Halbwachs and Mathias Péron

Grenoble – France

Objective

```
x := A[1] ; i := 2 ; j := n ;
while i ≤ j do
    if A[i] < x then
        A[i − 1] := A[i] ;
        i := i + 1
    else
        while j ≥ i and A[j] ≥ x do
            j := j − 1
        if j > i then
            A[i − 1] := A[j]; A[j] := A[i] ; i := i + 1 ; j := j − 1
A[i − 1] := x ;
```

- simple programs
- properties to discover

Objective

```
x := A[1] ; i := 2 ; j := n ;
while i ≤ j do
    if A[i] < x then
        A[i − 1] := A[i] ;
        i := i + 1
    else
        while j ≥ i and A[j] ≥ x do
            j := j − 1
        if j > i then
            A[i − 1] := A[j]; A[j] := A[i] ; i := i + 1 ; j := j − 1
A[i − 1] := x ;
```

- simple programs
    - one-dimensional arrays
- properties to discover

Objective

```
x := A[1] ; i := 2 ; j := n ;
while i ≤ j do
  if A[i] < x then
    A[i − 1] := A[i] ;
    i := i + 1
  else
    while j ≥ i and A[j] ≥ x do
      j := j − 1
    if j > i then
      A[i − 1] := A[j]; A[j] := A[i] ; i := i + 1 ; j := j − 1
A[i − 1] := x ;
```

- simple programs
  - one-dimensional arrays indexed by cte
- properties to discover

Objective

```
x := A[1] ; i := 2 ; j := n ;
while i ≤ j do
    if A[i] < x then
        A[i − 1] := A[i] ;
        i := i + 1
    else
        while j ≥ i and A[j] ≥ x do
            j := j − 1
        if  j > i then
            A[i − 1] := A[j]; A[j] := A[i] ; i := i + 1 ; j := j − 1
A[i − 1] := x ;
```

- simple programs
  - one-dimensional arrays indexed by cte or var + cte
- properties to discover

Objective

```
x := A[1] ; i := 2 ; j := n ;
while i ≤ j do
    if A[i] < x then
        A[i − 1] := A[i] ;
        i := i + 1
    else
        while j ≥ i and A[j] ≥ x do
            j := j − 1
        if j > i then
            A[i − 1] := A[j]; A[j] := A[i] ; i := i + 1 ; j := j − 1
A[i − 1] := x ;
```

- simple programs
  - one-dimensional arrays indexed by cte or var + cte
  - loop progression : ++/−−
- properties to discover

Objective

$x := A[1]\ ;\ i := 2\ ;\ j := n\ ;$
while $i \leq j$ **do**
  if $A[i] < x$ **then**
    $A[i-1] := A[i]\ ;$
    $i := i + 1$
  else
    while $j \geq i$ *and* $A[j] \geq x$ **do**
      $j := j - 1$
    if $j > i$ **then**
      $A[i-1] := A[j];\ A[j] := A[i]\ ;\ i := i + 1\ ;\ j := j - 1$
$A[i-1] := x\ ;$

$$\{1 < i < n \wedge \forall \ell, (1 \leq \ell < i) \Rightarrow (A[\ell] \leq x) \wedge \forall \ell, (i \leq \ell \leq n) \Rightarrow (A[\ell] > x) \ldots\}$$

- simple programs
  - one-dimensional arrays indexed by cte or var + cte
  - loop progression : $++/--$
- properties to discover

Objective

$x := A[1]$; $i := 2$; $j := n$;
while $i \leq j$ **do**
  if $A[i] < x$ **then**
    $A[i-1] := A[i]$;
    $i := i+1$
  else
    while $j \geq i$ *and* $A[j] \geq x$ **do**
      $j := j-1$
    if $j > i$ **then**
      $A[i-1] := A[j]; A[j] := A[i]$; $i := i+1$; $j := j-1$
$A[i-1] := x$;
$\{1 < i < n \wedge \forall \ell, (1 \leq \ell < i) \Rightarrow (A[\ell] \leq x)$
$\wedge \forall \ell, (i \leq \ell \leq n) \Rightarrow (A[\ell] > x) \ldots\}$

- simple programs
  - one-dimensional arrays indexed by cte or var + cte
  - loop progression : $++/--$
- properties to discover
  - about indices

Objective

$x := A[1]$; $i := 2$; $j := n$;
while $i \leq j$ **do**
  if $A[i] < x$ **then**
    $A[i-1] := A[i]$;
    $i := i + 1$
  else
    while $j \geq i$ *and* $A[j] \geq x$ **do**
      $j := j - 1$
    if $j > i$ **then**
      $A[i-1] := A[j]$; $A[j] := A[i]$; $i := i + 1$; $j := j - 1$
$A[i-1] := x$;

$$\{1 < i < n \wedge \forall \ell, (1 \leq \ell < i) \Rightarrow (A[\ell] \leq x) \wedge \forall \ell, (i \leq \ell \leq n) \Rightarrow (A[\ell] > x) \ldots\}$$

- simple programs
  - one-dimensional arrays indexed by cte or var + cte
  - loop progression : $++/--$
- properties to discover
  - about indices
  - about arrays: use 1 $\forall$var ℓ unary property

Objective

- simple programs
  - one-dimensional arrays indexed by cte or var + cte
  - loop progression : $++/--$
- properties to discover
  - about indices
  - about arrays: use 1 $\forall$var ℓ unary property

$i := 2$;
while $i \leq n$ **do**
  $x := A[i]$; $j := i - 1$;
  while $j \geq 1$ *and* $A[j] > x$ **do**
    $A[j+1] := A[j]$; $j := j - 1$
  $A[j+1] := x$;
  $i := i + 1$

$\{i = n + 1 \ \wedge \ \forall \ell, (2 \leq \ell \leq n) \Rightarrow (A[\ell - 1] \leq A[\ell])\}$

Objective

- simple programs
  - one-dimensional arrays indexed by cte or var + cte
  - loop progression : $++/--$
- properties to discover
  - about indices
  - about arrays: use 1 $\forall$var ℓ unary or relational property

```
i := 2 ;
while i ≤ n do
    x := A[i]; j := i − 1 ;
    while j ≥ 1 and A[j] > x do
        A[j + 1] := A[j] ; j := j − 1
    A[j + 1] := x ;
    i := i + 1
```

$\{i = n + 1 \ \wedge \ \forall \ell, (2 \leq \ell \leq n) \Rightarrow (A[\ell - 1] \leq A[\ell])\}$

Reaching the Objective

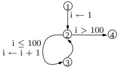
- reminder: invariant synthesis, no verification

- framework: abstract interpretation

  theory of approximate computation of fixpoint equations

  ▶ abstract domains

```
i ← 1 ;
while i ≤ 100 do
  i ← i + 1 ;
```



$R1 = all$
$R2 = (R1\ [i \leftarrow 1]) \cup R3$
$R3 = (R2 \cap (i \leq 100))\ [i \leftarrow i+1]$
$R4 = R3 \cap (i > 100)$

- properties about indices
  ▶ not hard for simple programs
- array bound checking
  ▶ assumed

Reaching the Objective

- reminder: invariant synthesis, no verification

```
(*************************************************************************)
(*                                                                       *)
(* FIND, an historical example.                                          *)
(*                                                                       *)
(* The proof of this program was originally done by C. A. R. Hoare       *)
(* and fully detailed in the following paper:                            *)
(*                                                                       *)
(* C. A. R. Hoare, "Proof of a Program: FIND", Communications of the     *)
(* ACM, 14(1), 39--45, January 1971.                                     *)
(*                                                                       *)
(*************************************************************************)
(* Jean-Christophe FILLIATRE, February 98                                 *)
(*************************************************************************)

let find =
  init:
  let m = ref 1 in let n = ref N in
  while !m < !n do
    let r = A[f] in let i = ref !m in let j = ref !n in
    begin
      while !i <= !j do
        label L;

        while A[!i] < r do
          i := !i + 1
        done;

        while r < A[!j] do
          j := !j - 1
        done;

        if !i <= !j then begin
          let w = A[!i] in begin A[!i] := A[!j]; A[!j] := w end;
          i := !i + 1;
          j := !j - 1
        end
      done;

      if f <= !j then
        n := !j
      else if !i <= f then
        m := !i
      else
        begin n := f; m := f end
    end
  done
```

```
(*************************************************************************)
(*                                                                       *)
(* FIND, an historical example.                                          *)
(*                                                                       *)
(* The proof of this program was originally done by C. A. R. Hoare       *)
(* and fully detailed in the following paper:                            *)
(*                                                                       *)
(* C. A. R. Hoare, "Proof of a Program: FIND", Communications of the     *)
(* ACM, 14(1), 39--45, January 1971.                                     *)
(*                                                                       *)
(*************************************************************************)
(* Jean-Christophe FILLIATRE, February 98                                 *)
(*************************************************************************)

let find =
  { array_length(A) = N+1 }
  init:
  let m = ref 1 in let n = ref N in
  while !m < !n do
    { invariant m_invariant(m,A) and n_invariant(n,A) and permut(A,A@init)
           and 1 <= m and n <= N as Inv_mn}
    let r = A[f] in let i = ref !m in let j = ref !n in
    begin
      while !i <= !j do
        { invariant i_invariant(m,n,i,r,A) and j_invariant(m,n,j,r,A)
               and m_invariant(m,A) and n_invariant(n,A)
               and 0 <= j and i <= N+1
               and termination(i,j,m,n,r,A)
               and permut(A,A@init) as Inv_ij}
        label L;

        while A[!i] < r do
          { invariant i_invariant(m, n, i, r, A)
                 and i@L <= i and i <= n
                 and termination(i, j, m, n, r, A) as Inv_i}
          i := !i + 1
        done;

        while r < A[!j] do
          { invariant j_invariant(m, n, j, r, A)
                 and j <= j@L and m <= j
                 and termination(i, j, m, n, r, A) as Inv_j}
          j := !j - 1
        done;

        if !i <= !j then begin
```

Reaching the Objective

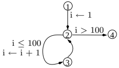
- reminder: invariant synthesis, no verification


- framework: abstract interpretation

  theory of approximate computation of fixpoint equations

  ▶ abstract domains

```
i ← 1 ;
while i ≤ 100 do
  i ← i + 1 ;
```



R1 = *all*
R2 = (R1 $[i \leftarrow 1]$) ∪ R3
R3 = (R2 ∩$(i \leq 100)$) $[i \leftarrow i+1]$
R4 = R3 ∩$(i > 100)$

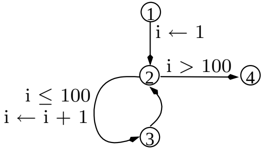
- properties about indices
  ▶ not hard for simple programs
- array bound checking
  ▶ assumed

Reaching the Objective

- reminder: invariant synthesis, no verification

- framework: abstract interpretation
  theory of approximate computation of fixpoint equations
  - abstract domains

i ← 1 ;
while $i \leq 100$ **do**
  i ← i + 1 ;

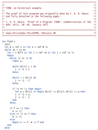


$$R1 = all$$
$$R2 = (R1\ [i \leftarrow 1]) \cup R3$$
$$R3 = (R2 \cap (i \leq 100))\ [i \leftarrow i + 1]$$
$$R4 = R3 \cap (i > 100)$$

- properties about indices

Reaching the Objective

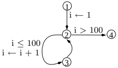
- reminder: invariant synthesis, no verification



- framework: abstract interpretation

  theory of approximate computation of fixpoint equations
  - abstract domains

i ← 1 ;
while $i \leq 100$ **do**
 i ← i + 1 ;

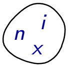


R1 = *all*
R2 = (R1 $[i \leftarrow 1]$) ∪ R3
R3 = (R2 $\cap (i \leq 100)$) $[i \leftarrow i + 1]$
R4 = R3 $\cap (i > 100)$

- properties about indices
  - not hard for simple programs
- array bound checking
  - assumed

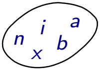
Related Abstract Domains

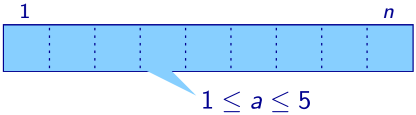
summarization [Astrée team 03] [Gopan *et al* 04]	summarization + partitioning [Gopan *et al* 05]	∀-quantified domain [Gulwani *et al* 08]



Related Abstract Domains

- summarization [Astrée team 03] [Gopan *et al* 04]
- summarization + partitioning [Gopan *et al* 05]
- ∀-quantified domain [Gulwani *et al* 08]





Abstract each array A by one variable a

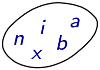
- interpretation: $\forall \ell, 1 \leq \ell \leq n \Rightarrow 1 \leq A[\ell] \leq 5$
- assignment $A[i] := expr$ is weak assignment to variable a:

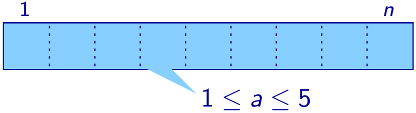
$$\textbf{if } ? \textbf{ then } a \leftarrow expr$$

e.g. $\{a \geq 10\}\ A[i] := 9\ \{a \geq 9\}$

Related Abstract Domains

- summarization [Astrée team 03] [Gopan *et al* 04]
- summarization + partitioning [Gopan *et al* 05]
- ∀-quantified domain [Gulwani *et al* 08]



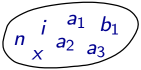


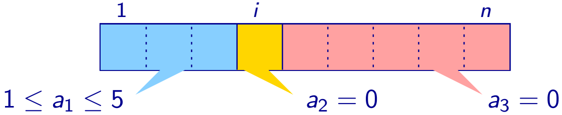
Conclusion:

- you can only loose information (weak assignment, and no gained from conditionals)
- only unary properties discovered

Related Abstract Domains

- summarization [Astrée team 03] [Gopan *et al* 04]
- summarization + partitioning [Gopan *et al* 05]
- ∀-quantified domain [Gulwani *et al* 08]





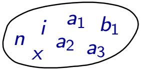
Partition each array A into symbolic slices and abstract them by variables a_p

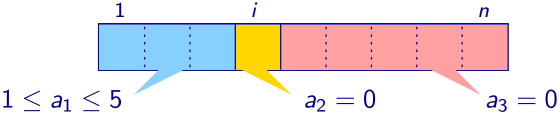
- interpretation: $(\forall \ell, 1 \leq \ell < i \Rightarrow 1 \leq A[\ell] \leq 5) \wedge A[i] = 0 \wedge \ldots$
- assignment $A[i] := expr$ is strong assignment to variable a_2:

$$a_2 \leftarrow expr$$

Related Abstract Domains

- summarization [Astrée team 03] [Gopan *et al* 04]
- summarization + partitioning [Gopan *et al* 05]
- $\forall$-quantified domain [Gulwani *et al* 08]



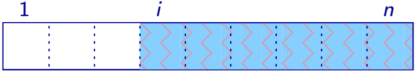


Conclusion:

- only unary properties discovered
- relations between array elements can be checked
  e.g. $\{\forall \ell, 1 \le \ell < i \Rightarrow A[\ell] = B[\ell]\}$

Related Abstract Domains

- summarization [Astrée team 03] [Gopan *et al* 04]
- summarization + partitioning [Gopan *et al* 05]
- ∀-quantified domain [Gulwani *et al* 08]



$$\forall k_1 \forall k_2, i \leq k_1 < k_2 \leq n \Rightarrow A[k_1] \leq A[k_2]$$

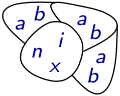
Formulas over universally quantified variables k_p, using uninterpreted functions to represent array accesses

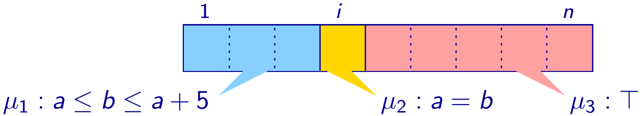
- highly expressive properties inferred (templates: $A[\star] \leq A[\star]$)
- sometimes no such expressiveness is required:

$$\forall \ell, i < \ell \leq n \Rightarrow A[\ell - 1] \leq A[\ell]$$

Our Proposition

- summarization [Astrée team 03] [Gopan *et al* 04]
- summarization + partitioning [Gopan *et al* 05]
- partitioning + slice properties
- ∀-quantified domain [Gulwani *et al* 08]





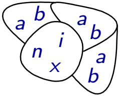
Partition arrays into symbolic slices and associate them properties with element-wise semantics

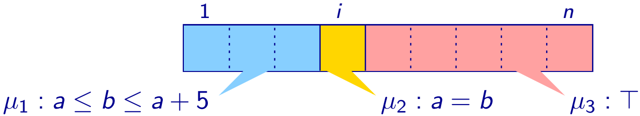
- interpretation: $(\forall \ell, 1 \leq \ell < i \Rightarrow A[\ell] \leq B[\ell] \leq A[\ell] + 5) \wedge \ldots$
- assignment $A[i] := expr$ is strong assignment to a in μ_2:

$$\mu_2 \rightsquigarrow \mu_2[a \leftarrow expr]$$

Our Proposition

- summarization [Astrée team 03] [Gopan *et al* 04]
- summarization + partitioning [Gopan *et al* 05]
- partitioning + slice properties
- ∀-quantified domain [Gulwani *et al* 08]





Conclusion:

- relational properties can be discovered ...
- ... that hold strictly within same symbolic slice

Abstract Values

Properties to discover

Abstract values

Abstract Values

Properties to discover

- about indices $\rho(i, j, n \ldots)$
- about arrays: use 1 $\forall$var, ℓ
  unary or relational

Abstract values

Abstract Values

Properties to discover

- about indices $\rho(i, j, n \ldots)$
- about arrays: use 1 $\forall$var, ℓ
  unary or relational

$$\bigwedge \varphi(\ell, i, j, n \ldots) \Rightarrow \mu(A[\ell + c_1], B[\ell + c_2], x \ldots)$$

Abstract values

Abstract Values

Properties to discover

- about indices $\rho(i, j, n \ldots)$
- about arrays: use 1 $\forall$var, ℓ
  unary or relational

$$\bigwedge \varphi(\ell, i, j, n \ldots) \Rightarrow \mu(A[\ell + c_1], B[\ell + c_2], x \ldots)$$

Abstract values

- parameterized

L_N lattice for indices, L_C lattice for contents

Abstract Values

Properties to discover

- about indices $\rho(i, j, n \ldots)$
- about arrays: use 1 $\forall$var, ℓ
  unary or relational

$$\bigwedge \varphi(\ell, i, j, n \ldots) \Rightarrow \mu(A[\ell + c_1], B[\ell + c_2], x \ldots)$$

Abstract values

- parameterized
  L_N lattice for indices, L_C lattice for contents
- partition based
  e.g. $1 \leq \ell \leq i$ $\{\varphi_p\}_{p \in P}$ $\varphi_p \in L_N$

Abstract Values

Properties to discover

- about indices $\rho(i, j, n \ldots)$
- about arrays: use 1 $\forall$var, ℓ
  unary or relational

$$\bigwedge \varphi(\ell, i, j, n \ldots) \Rightarrow \mu(A[\ell + c_1], B[\ell + c_2], x \ldots)$$

Abstract values

- parameterized
  L_N lattice for indices, L_C lattice for contents
- partition based
  e.g. $1 \leq \ell \leq i$ $\{\varphi_p\}_{p \in P}$ $\varphi_p \in L_N$
- slice variables
  $A[\ell + c]$ represented by var. a^c $\{\mu_p\}_{p \in P}$ $\mu_p \in L_C$

Abstract Values

Properties to discover

- about indices $\rho(i, j, n \ldots)$
- about arrays: use 1 $\forall$var, ℓ
  unary or relational

$$\bigwedge \varphi(\ell, i, j, n \ldots) \Rightarrow \mu(A[\ell + c_1], B[\ell + c_2], x \ldots)$$

Abstract values, over $\{\varphi_p\}_{p \in P}$

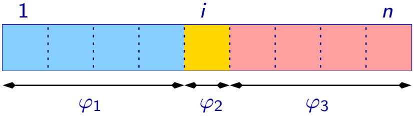
$$(\rho, \ \{\mu_p\}_{p \in P})$$

- parameterized
  L_N lattice for indices, L_C lattice for contents
- partition based
  e.g. $1 \leq \ell \leq i$ $\{\varphi_p\}_{p \in P}$ $\varphi_p \in L_N$
- slice variables
  $A[\ell + c]$ represented by var. a^c $\{\mu_p\}_{p \in P}$ $\mu_p \in L_C$

Abstract Values *(Example 1)*

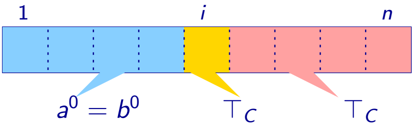
- parameters $\quad L_N = \textit{potential constraints}, L_C = \textit{equations}$
- partition

$$\varphi_1 : 1 \leq \ell < i \leq n$$
$$\varphi_2 : 1 \leq \ell = i \leq n$$
$$\varphi_3 : 1 \leq i < \ell \leq n$$



- abstract value

$$\begin{pmatrix} \rho : 1 \leq i \leq n \\ \mu_1 : a^0 = b^0 \\ \mu_2 : \top_C \\ \mu_3 : \top_C \end{pmatrix}$$



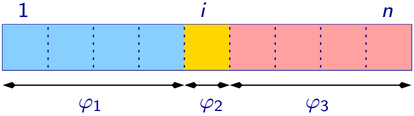
- interpretation

$$1 \leq i \leq n \quad \wedge \quad \forall \ell, 1 \leq \ell < i \Rightarrow A[\ell] = B[\ell]$$

Abstract Values *(Example 1)*

- parameters $L_N = potential\ constraints, L_C = equations$
- partition

$\varphi_1 : 1 \leq \ell < i \leq n$
$\varphi_2 : 1 \leq \ell = i \leq n$
$\varphi_3 : 1 \leq i < \ell \leq n$



- abstract value

$$\begin{pmatrix} \rho : i = n+1 \\ \mu_1 : a^0 = b^0 \\ \mu_2 : \top_C \\ \mu_3 : \top_C \end{pmatrix}$$

If $\rho \Rightarrow \neg(\exists \ell \varphi_p)$
μ_p can be normalized to $\bot_C$

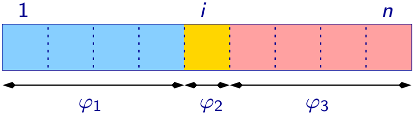
- interpretation

$$1 \leq i \leq n \quad \wedge \quad \forall \ell, 1 \leq \ell < i \Rightarrow A[\ell] = B[\ell]$$

Abstract Values *(Example 1)*

- parameters $L_N = potential\ constraints, L_C = equations$
- partition

$$\varphi_1 : 1 \leq \ell < i \leq n$$
$$\varphi_2 : 1 \leq \ell = i \leq n$$
$$\varphi_3 : 1 \leq i < \ell \leq n$$



- abstract value

$$\begin{pmatrix} \rho : i = n + 1 \\ \mu_1 : a^0 = b^0 \\ \mu_2 : \bot_C \\ \mu_3 : \bot_C \end{pmatrix}$$

If $\rho \Rightarrow \neg(\exists \ell \varphi_p)$
μ_p can be normalized to $\bot_C$

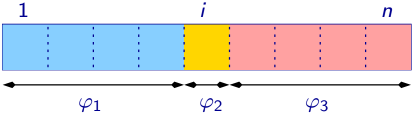
- interpretation

$$1 \leq i \leq n \quad \wedge \quad \forall \ell, 1 \leq \ell < i \Rightarrow A[\ell] = B[\ell]$$

Abstract Values *(Example 1)*

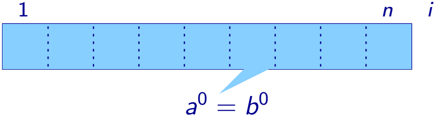
- parameters $L_N = potential\ constraints, L_C = equations$
- partition

$\varphi_1 : 1 \leq \ell < i \leq n$
$\varphi_2 : 1 \leq \ell = i \leq n$
$\varphi_3 : 1 \leq i < \ell \leq n$



- abstract value

$$\begin{pmatrix} \rho : i = n+1 \\ \mu_1 : a^0 = b^0 \\ \mu_2 : \bot_C \\ \mu_3 : \bot_C \end{pmatrix}$$



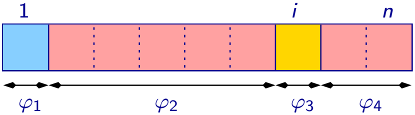
- interpretation

$$i = n+1 \quad \wedge \quad \forall \ell, 1 \leq \ell \leq n \Rightarrow A[\ell] = B[\ell]$$

Abstract Values *(Example 2)*

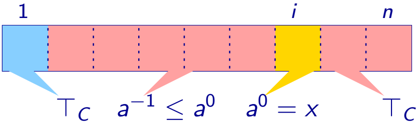
- parameters $L_N = potential\ constraints, L_C = comparisons$
- partition

$\varphi_1 : 1 = \ell \leq n$
$\varphi_2 : 2 \leq \ell < i \leq n$
$\varphi_3 : 2 \leq \ell = i \leq n$
$\varphi_4 : 2 \leq i < \ell \leq n$



- abstract value

$$\begin{pmatrix} \rho : 2 \leq i \leq n \\ \mu_1 : \top_C \\ \mu_2 : a^{-1} \leq a^0 \\ \mu_3 : a^0 = x \\ \mu_4 : \top_C \end{pmatrix}$$



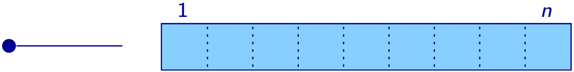
- interpretation

$$2 \leq i \leq n \quad \wedge \quad \forall \ell, 2 \leq \ell < i \Rightarrow A[\ell - 1] \leq A[\ell] \wedge A[i] = x$$

Analysis at Work *(Partition Choice)*

- decide partitions at each control point [Gopan, Reps, Sagiv '05]
- fixpoint computation over the abstract domain

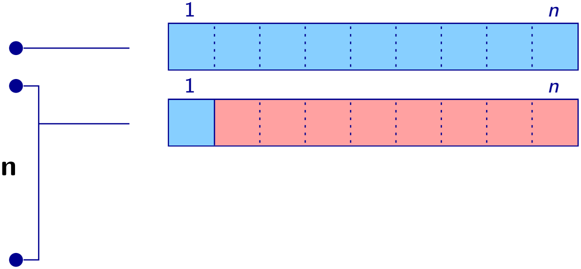
$max := A[1]$;
$i := 2$;
while $i \leq n$ **do**
  if $max < A[i]$ **then**
    $max := A[i]$
  $i := i + 1$



Analysis at Work *(Partition Choice)*

- decide partitions at each control point [Gopan, Reps, Sagiv '05]
  - index initializations
  - index expressions of arrays in guards / assignments
- fixpoint computation over the abstract domain

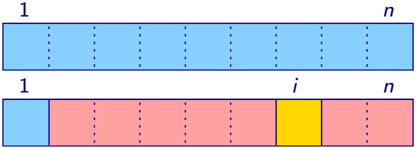
$max := A[1]$;
$i := 2$;
while $i \leq n$ **do**
  if $max < A[i]$ **then**
    $max := A[i]$
  $i := i + 1$



Analysis at Work *(Partition Choice)*

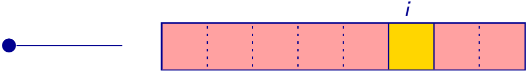
- decide partitions at each control point [Gopan, Reps, Sagiv '05]
  - index initializations
  - index expressions of arrays in guards / assignments
- fixpoint computation over the abstract domain

$max := A[1]$;
$i := 2$;
while $i \leq n$ **do**
  if $max < A[i]$ **then**
    $max := A[i]$
  $i := i + 1$



Analysis at Work *(Partition Choice)*

- decide partitions at each control point [Gopan, Reps, Sagiv '05]
  - index initializations
  - index expressions of arrays in guards / assignments
- fixpoint computation over the abstract domain



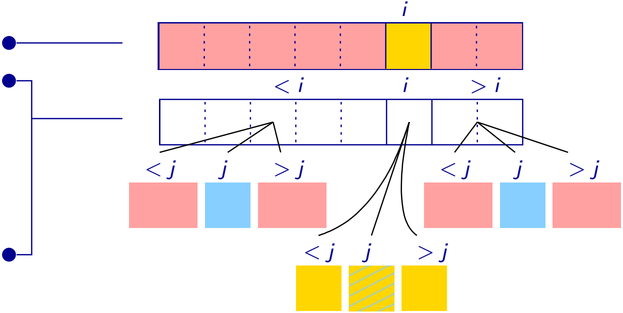

```
... ;
while j do
    ... ;
    A[j] := ... ;
    ... ;
```

Analysis at Work *(Partition Choice)*

- decide partitions at each control point [Gopan, Reps, Sagiv '05]
  - index initializations
  - index expressions of arrays in guards / assignments

  ▶ distinguish aliases !
- fixpoint computation over the abstract domain

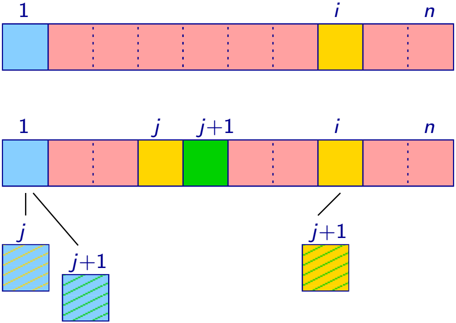
```
... ;
while j do
    ... ;
    A[j] := ... ;
    ... ;
```



Analysis at Work *(Partition Choice)*

- Example: insertion sort

$i := 2$;
while $i \leq n$ **do**
  $x := A[i]$; $j := i - 1$;
  while $j \geq 1$ *and* $A[j] > x$ **do**
    $A[j+1] := A[j]$;
    $j := j - 1$
  $A[j+1] := x$;
  $i := i + 1$

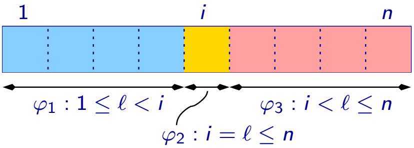


▶ $\{\exists \ell \varphi_p\}_{p \in P}$ gives all considered disjunctive cases

Analysis at Work *(Fixpoint Computation)*

$i := 1$;
while $i \leq n$ **do**
  $A[i] := B[i]$;
  $i := i + 1$

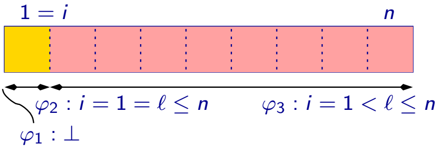
ρ	μ_1	μ_2	μ_3



Analysis at Work *(Fixpoint Computation)*

$i := 1$;
while $i \leq n$ **do**
  $A[i] := B[i]$;
  $i := i + 1$

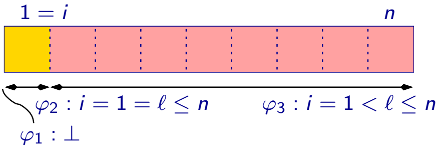
	ρ	μ_1	μ_2	μ_3
⋆	$i = 1$	$\bot$	$\top$	$\top$



Analysis at Work *(Fixpoint Computation)*

$i := 1$;
while $i \leq n$ **do**
  $A[i] := B[i]$;
  $i := i + 1$

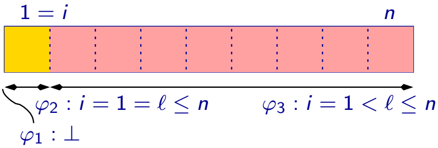
	ρ	μ_1	μ_2	μ_3
	$i = 1$	$\bot$	$\top$	$\top$
⋆	$i = 1 \leq n$	$\bot$	$\top$	$\top$



Analysis at Work *(Fixpoint Computation)*

$i := 1$;
while $i \leq n$ **do**
  $A[i] := B[i]$;
  $i := i + 1$

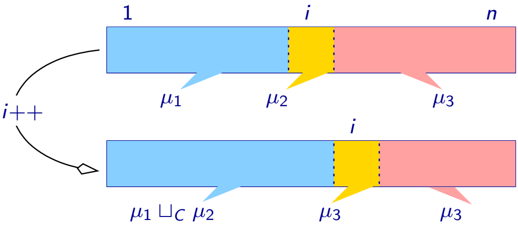
	ρ	μ_1	μ_2	μ_3
	$i = 1$	$\bot$	$\top$	$\top$
	$i = 1 \leq n$	$\bot$	$\top$	$\top$
⋆	$i = 1 \leq n$	$\bot$	$a^0 = b^0$	$\top$



Analysis at Work *(Fixpoint Computation)*

$i := 1$;
while $i \leq n$ **do**
  $A[i] := B[i]$;
  $i := i + 1$

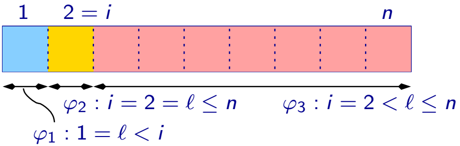
ρ	μ_1	μ_2	μ_3
$i = 1$	$\bot$	$\top$	$\top$
$i = 1 \leq n$	$\bot$	$\top$	$\top$
$i = 1 \leq n$	$\bot$	$a^0 = b^0$	$\top$



Analysis at Work *(Fixpoint Computation)*

$i := 1$;
while $i \leq n$ **do**
  $A[i] := B[i]$;
  $i := i + 1$

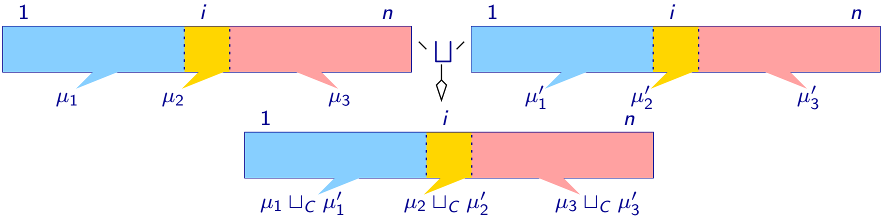
	ρ	μ_1	μ_2	μ_3
	$i = 1$	$\bot$	$\top$	$\top$
	$i = 1 \leq n$	$\bot$	$\top$	$\top$
	$i = 1 \leq n$	$\bot$	$a^0 = b^0$	$\top$
⋆	$i = 2 \leq n+1$	$a^0 = b^0$	$\top$	$\top$



Analysis at Work *(Fixpoint Computation)*

$i := 1$;
while $i \leq n$ **do**
  $A[i] := B[i]$;
  $i := i + 1$

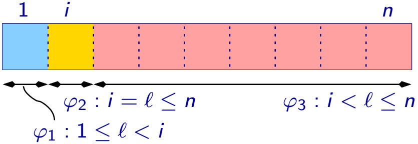
ρ	μ_1	μ_2	μ_3
$i = 1$	$\bot$	$\top$	$\top$
$i = 1 \leq n$	$\bot$	$\top$	$\top$
$i = 1 \leq n$	$\bot$	$a^0 = b^0$	$\top$
$i = 2 \leq n{+}1$	$a^0 = b^0$	$\top$	$\top$



Analysis at Work *(Fixpoint Computation)*

$i := 1$;
while $i \leq n$ **do**
  $A[i] := B[i]$;
  $i := i + 1$

	ρ	μ_1	μ_2	μ_3
	$i = 1$	$\bot$	$\top$	$\top$
⋆	$1 \leq i \leq 2$	$a^0 = b^0$	$\top$	$\top$



Analysis at Work *(Fixpoint Computation)*

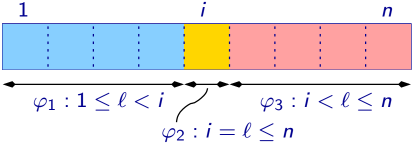
$i := 1$;

while $i \leq n$ **do**

  $A[i] := B[i]$;

  $i := i + 1$

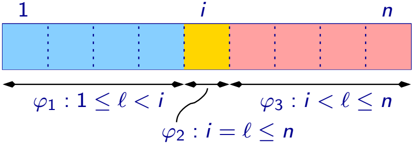
	ρ	μ_1	μ_2	μ_3
	$i = 1$	$\bot$	$\top$	$\top$
∇	$1 \leq i \leq n$	$a^0 = b^0$	$\top$	$\top$



Analysis at Work *(Fixpoint Computation)*

$i := 1$;
while $i \leq n$ **do**
  $A[i] := B[i]$;
  $i := i + 1$

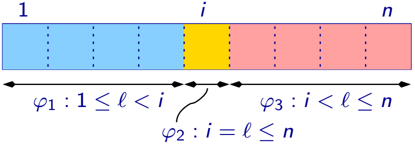
	ρ	μ_1	μ_2	μ_3
	$i = 1$	$\bot$	$\top$	$\top$
	$1 \leq i \leq n$	$a^0 = b^0$	$\top$	$\top$
$\star$	$1 \leq i \leq n$	$a^0 = b^0$	$a^0 = b^0$	$\top$



Analysis at Work *(Fixpoint Computation)*

$i := 1$;
while $i \leq n$ **do**
  $A[i] := B[i]$;
  $i := i + 1$

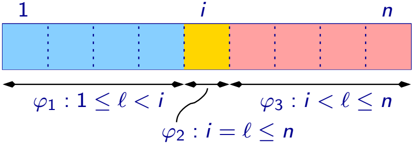
	ρ	μ_1	μ_2	μ_3
	$i = 1$	$\bot$	$\top$	$\top$
	$1 \leq i \leq n$	$a^0 = b^0$	$\top$	$\top$
	$1 \leq i \leq n$	$a^0 = b^0$	$a^0 = b^0$	$\top$
⋆	$2 \leq i \leq n{+}1$	$a^0 = b^0$	$\top$	$\top$



Analysis at Work *(Fixpoint Computation)*

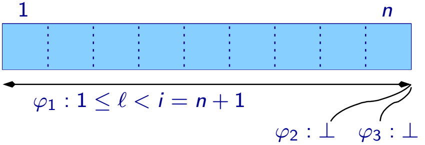
$i := 1$;
while $i \leq n$ **do**
  $A[i] := B[i]$;
  $i := i + 1$

ρ	μ_1	μ_2	μ_3
$i = 1$	$\bot$	$\top$	$\top$
$1 \leq i \leq n$	$a^0 = b^0$	$\top$	$\top$
$1 \leq i \leq n$	$a^0 = b^0$	$a^0 = b^0$	$\top$
$2 \leq i \leq n{+}1$	$a^0 = b^0$	$\top$	$\top$



Analysis at Work *(Fixpoint Computation)*

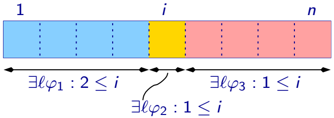
Program		ρ	μ_1	μ_2	μ_3
$i := 1$;		$i = 1$	$\bot$	$\top$	$\top$
while $i \leq n$ **do**		$1 \leq i \leq n$	$a^0 = b^0$	$\top$	$\top$
$A[i] := B[i]$;		$1 \leq i \leq n$	$a^0 = b^0$	$a^0 = b^0$	$\top$
$i := i + 1$		$2 \leq i \leq n{+}1$	$a^0 = b^0$	$\top$	$\top$
	$\star$	$i = n + 1$	$a^0 = b^0$	$\bot$	$\bot$



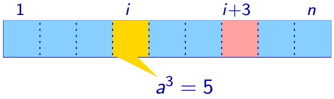
$\{\forall \ell, 1 \leq \ell \leq n \Rightarrow A[\ell] = B[\ell]\}$

Normalization in Details

- Use normalization procedures of L_N and L_C
  - Normalization to $\bot$
    if unfeasible indices property: $\rho = \bot_N$
  - All properties does not depend on ℓ !
    scalar consistency in slice properties
    $\exists \ell \varphi_2 \Rightarrow \exists \ell \varphi_1 \Rightarrow ScalarProperty(\mu_1)$

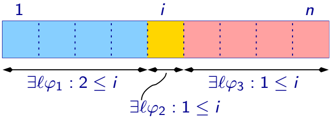


  - Deduce in a reasonnable way array properties
    shift consistency in slice properties

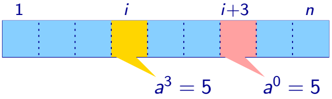


Normalization in Details

- Use normalization procedures of L_N and L_C
  - Normalization to $\bot$
    if unfeasible indices property: $\rho = \bot_N$
  - All properties does not depend on ℓ !
    scalar consistency in slice properties
    $\exists \ell \varphi_2 \Rightarrow \exists \ell \varphi_1 \Rightarrow ScalarProperty(\mu_1)$

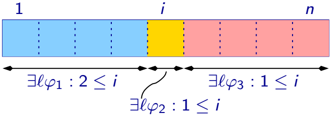


  - Deduce in a reasonnable way array properties
    shift consistency in slice properties

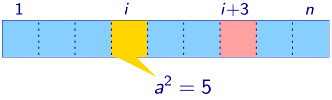


Normalization in Details

- Use normalization procedures of L_N and L_C
  - Normalization to $\bot$
    if unfeasible indices property: $\rho = \bot_N$
  - All properties does not depend on ℓ !
    scalar consistency in slice properties
    $\exists \ell \varphi_2 \Rightarrow \exists \ell \varphi_1 \Rightarrow ScalarProperty(\mu_1)$



  - Deduce in a reasonnable way array properties
    shift consistency in slice properties



Some Results

program	$\vert\{\varphi_p\}_{p \in P}\vert$	*# slice var. in μ_p avg (max)*	*time (s)*
array copy	3	0 (0)	0.02
sequence init.	4	0.8 (2)	0.05
maximum search	4	0.8 (2)	0.10
sentinel	9	0 (1)	0.21
first not null	13	0 (1)	2.25
insertion sort	4-10	4.6 (11)	5.38
find (quicksort)	14	6.7 (14)	22.87

Prototype tool written in OCAML

- $L_N = L_C =$ *potential constraints* (DBM)

Some Results

program	$\lvert\{\varphi_p\}_{p\in P}\rvert$	*# slice var. in* μ_p *avg (max)*	*time (s)*
array copy	3	0 (0)	0.02
sequence init.	4	0.8 (2)	0.05
maximum search	4	0.8 (2)	0.10
sentinel	9	0 (1)	0.21
first not null	13	0 (1)	2.25
insertion sort	4-10	4.6 (11)	5.38
find (quicksort)	14	6.7 (14)	22.87

Good results on one-loop programs

Some Results

program	$\lvert\{\varphi_p\}_{p\in P}\rvert$	*# slice var. in μ_p avg (max)*	*time (s)*
array copy	3	0 (0)	0.02
sequence init.	4	0.8 (2)	0.05
maximum search	4	0.8 (2)	0.10
sentinel	9	0 (1)	0.21
first not null	13	0 (1)	2.25
insertion sort	4-10	4.6 (11)	5.38
find (quicksort)	14	6.7 (14)	22.87

A longstanding challenge in array bound checking

$A[n] := x \; ; \; i := 1 \; ;$
while $A[i] \neq x$ **do**
$\quad i := i + 1$

$$\{1 \leq i \leq n \;\wedge\; A[i] = x \;\wedge\; (\forall \ell, 1 \leq \ell < i \Rightarrow A[\ell] \neq x)\}$$

Some Results

program	$\|\{\varphi_p\}_{p\in P}\|$	*# slice var. in* μ_p *avg (max)*	*time (s)*
array copy	3	0 (0)	0.02
sequence init.	4	0.8 (2)	0.05
maximum search	4	0.8 (2)	0.10
sentinel	9	0 (1)	0.21
first not null	13	0 (1)	2.25
insertion sort	4-10	4.6 (11)	5.38
find (quicksort)	14	6.7 (14)	22.87

Reasonable results on relatively intricate multi-loops program

- sensitive to: number of slices + slice variables

Some Results

program	$\|\{\varphi_p\}_{p \in P}\|$	*# slice var. in μ_p avg (max)*	*time (s)*
array copy	3	0 (0)	0.02
sequence init.	4	0.8 (2)	0.05
maximum search	4	0.8 (2)	0.10
sentinel	9	0 (1)	0.21
first not null	13	0 (1)	2.25
insertion sort	4-10	4.6 (11)	5.38
find (quicksort)	14	6.7 (14)	22.87

Reasonable results on relatively intricate multi-loops program

- sensitive to: number of slices + slice variables

Conclusions

Achievements

- fully-automatic discovery of properties on array contents

Future work

- extend the class of simple programs
  - loops with steps, recursivity
- handle more expressive properties
  - non convex slices
- new analysis for the multiset of contents of arrays
  - domain for multi-sets