Abstract Program Slicing: Abstract interpretation-based approaches to Slicing

Isabella Mastroeni
(Đurica Nikolić and Damiano Zanardini)

Dipartimento di Informatica, University of Verona, Italy

30 April 2012

PROGRAM SLICING: BASIC NOTIONS

Program Slicing

A program decomposition technique that extracts from programs statements which affect parameters of interest

Slicing Criterion

Contains different parameters of interest (e.g., $\mathcal{C} = (V, n)$ [Weiser '79])

Program Slice

An executable program obtained that way

PROGRAM SLICING: BASIC NOTIONS

Example

```
1  begin
2    read(x,y);
3    total := 0.0;
4    sum := 0.0;
5    if x <= 1
6    then sum := y;
7    else begin
8              read(z);
9              total := x*y;
10         end;
11   write(total, sum);
12 end.
```

```
begin
  read(x, y);
  if x <= 1
  then
  else
    read(z);
end.
```

(12, z)

```
begin
  read(x, y);
end.
```

(9, x)

```
begin
  read(x, y);
   total := 0.0;
   if x <= 1
  then
  else
   total := x*y;
end.
```

(12, total)

⇒ Slices depend on slicing criterion

THE IDEA

Limitations

- Sometimes standard criteria are too strong

Weakening slicing

- Suppose we want a variable x to have a property ρ at some point n
- The exact value of x can be expressed as $\rho = id = \lambda a.a$
- We are interested in the statements that affect $\rho(x)$ at n
- Abstract slices should be smaller

THE IDEA

Example

$$P \stackrel{\text{def}}{=} \begin{cases} a & := & 1; \\ b & := & b+1; \\ c & := & c+2; \\ d & := & c+b+a-a+c; \end{cases}$$

Abstract criterion: Parity of d

THE IDEA

Example

$$P \stackrel{\text{def}}{=} \begin{cases} a & := & 1; \\ b & := & b+1; \\ c & := & c+2; \\ d & := & c+b+a-a+c; \end{cases}$$

Abstract criterion: Parity of d

THE IDEA

Example

$$P \stackrel{\text{def}}{=} \begin{cases} b & := & b+1; \\ d & := & c+b+a-a+c; \end{cases}$$

Abstract criterion: Parity of *d*

Related works

[Amtoft & Banerjee '07]

Slicing by means of a calculus for independencies

- Syntactic dependencies
- Forward slicing

[Rival '05]

Abstract dependencies

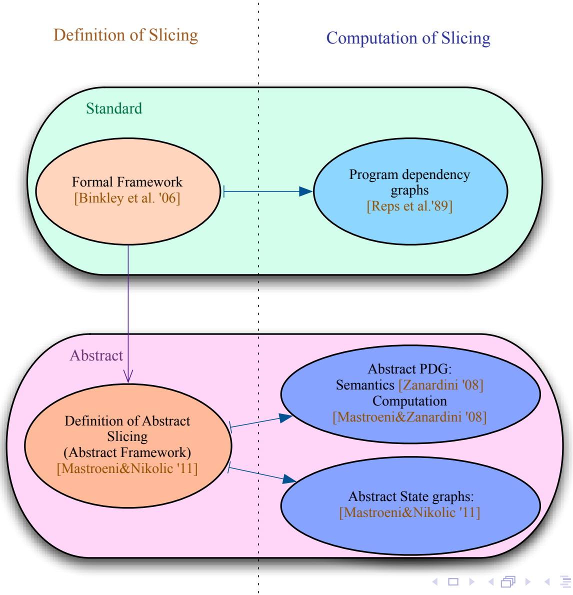
- Mathematical, set theoretic definition of dependencies
- Applied to Alarm diagnosis

[Hong et al. '05]

Abstract Slicing

- Only for predicate abstractions
- Considers a subset of possible executions

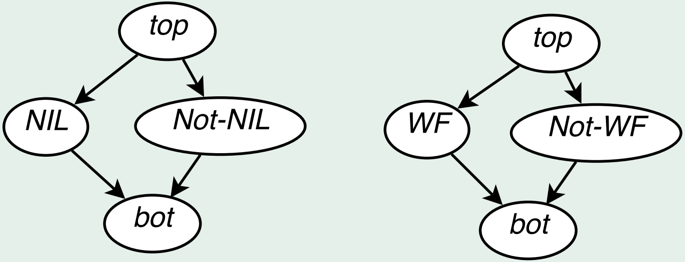
ABSTRACT INTERPRETATION-BASED APPROACHES



REVERSING WELL-FORMED LISTS [Zanardini '08]

Well-formed lists: $\langle 1, 2, 3, 4, [0]\rangle{+}{+}\langle 5, 6, [0]\rangle = \langle 1, 2, 3, 4, 5, 6, [0]\rangle$

The properties of interest are represented by abstract domains for *nullity* and *well-formedness*:



ρ_{nil} ρ_{WF}

$$wellFormed(x) \equiv notNil(x) \wedge lastEl(x).data = 0$$

REVERSING WELL-FORMED LISTS [Zanardini '08]

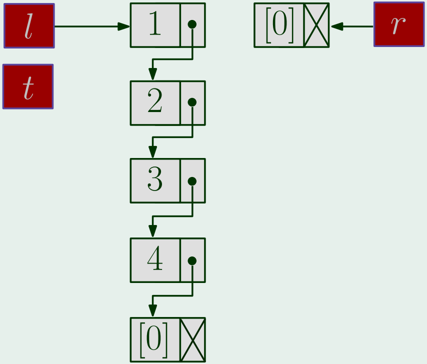
Reversing the list

```
list rev(list l) {
  list *last;
  list *tmp;
  while (l->next != null){
    tmp = l->next;
    l->next = last;
    last = l;
    l = tmp;
  }
  return last;
}
```

REVERSING WELL-FORMED LISTS [Zanardini '08]

Reversing the list

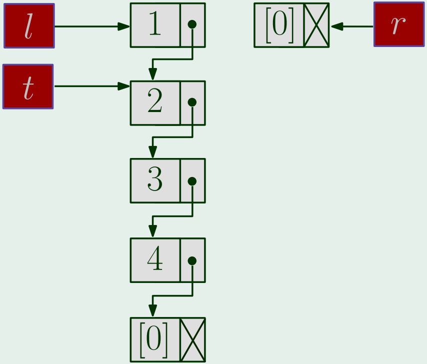
```
list rev(list l) {
  list *r;
  list *t;
  while (l->next != null){
    t = l->next;
    l->next = r;
    r = l;
    l = t;
  }
  return r;
}
```



REVERSING WELL-FORMED LISTS [Zanardini '08]

Reversing the list

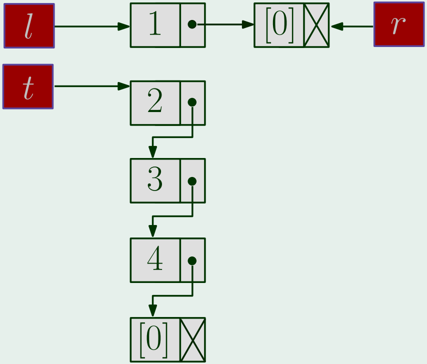
```
list rev(list l) {
  list *r;
  list *t;
  while (l->next != null){
    t = l->next;
    l->next = r;
    r = l;
    l = t;
  }
  return r;
}
```



REVERSING WELL-FORMED LISTS [Zanardini '08]

Reversing the list

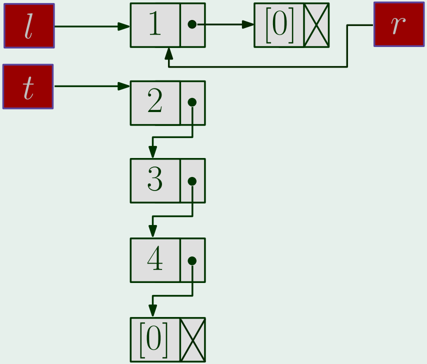
```
list rev(list l) {
  list *r;
  list *t;
  while (l->next != null){
    t = l->next;
    l->next = r;
    r = l;
    l = t;
  }
  return r;
}
```



REVERSING WELL-FORMED LISTS [Zanardini '08]

Reversing the list

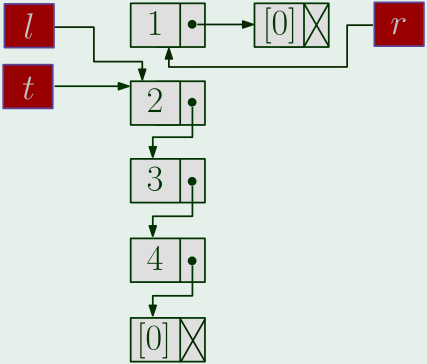
```
list rev(list l) {
  list *r;
  list *t;
  while (l->next != null){
    t = l->next;
    l->next = r;
    r = l;
    l = t;
  }
  return r;
}
```



Reversing Well-Formed Lists [Zanardini '08]

Reversing the list

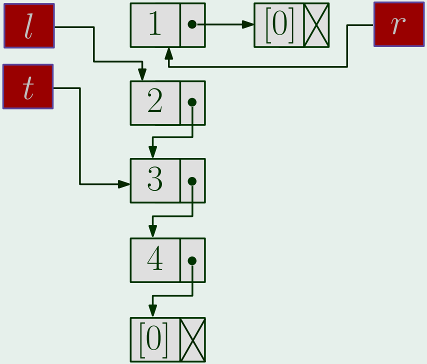
```
list rev(list l) {
  list *r;
  list *t;
  while (l->next != null){
    t = l->next;
    l->next = r;
    r = l;
    l = t;
  }
  return r;
}
```



REVERSING WELL-FORMED LISTS [Zanardini '08]

Reversing the list

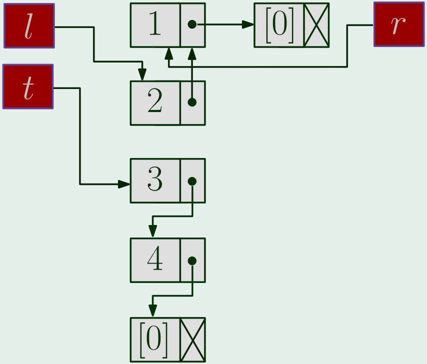
```
list rev(list l) {
  list *r;
  list *t;
  while (l->next != null){
    t = l->next;
    l->next = r;
    r = l;
    l = t;
  }
  return r;
}
```



REVERSING WELL-FORMED LISTS [Zanardini '08]

Reversing the list

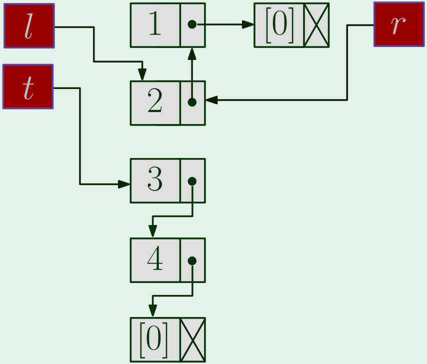
```
list rev(list l) {
  list *r;
  list *t;
  while (l->next != null){
    t = l->next;
    l->next = r;
    r = l;
    l = t;
  }
  return r;
}
```



REVERSING WELL-FORMED LISTS [Zanardini '08]

Reversing the list

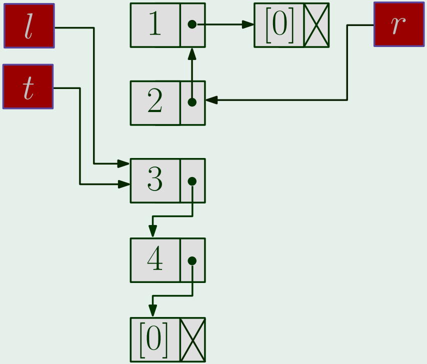
```
list rev(list l) {
  list *r;
  list *t;
  while (l->next != null){
    t = l->next;
    l->next = r;
    r = l;
    l = t;
  }
  return r;
}
```



Reversing Well-Formed Lists [Zanardini '08]

Reversing the list

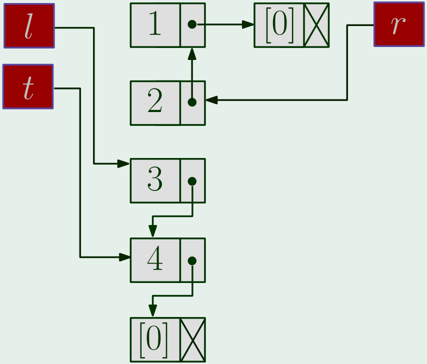
```
list rev(list l) {
  list *r;
  list *t;
  while (l->next != null){
    t = l->next;
    l->next = r;
    r = l;
    l = t;
  }
  return r;
}
```



REVERSING WELL-FORMED LISTS [Zanardini '08]

Reversing the list

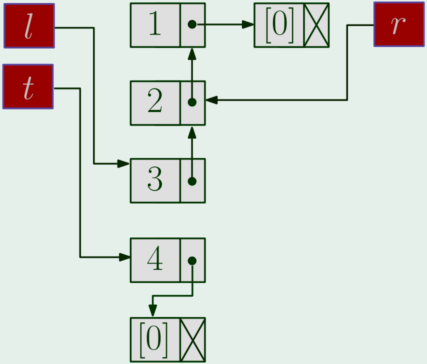
```
list rev(list l) {
  list *r;
  list *t;
  while (l->next != null){
    t = l->next;
    l->next = r;
    r = l;
    l = t;
  }
  return r;
}
```



Reversing Well-Formed Lists [Zanardini '08]

Reversing the list

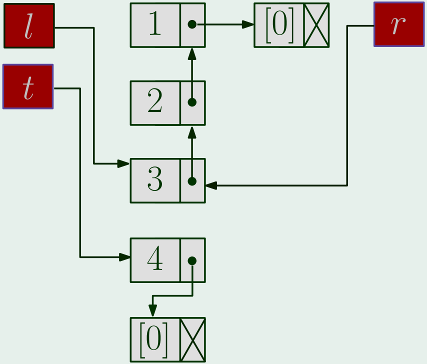
```
list rev(list l) {
  list *r;
  list *t;
  while (l->next != null){
    t = l->next;
    l->next = r;
    r = l;
    l = t;
  }
  return r;
}
```



REVERSING WELL-FORMED LISTS [Zanardini '08]

Reversing the list

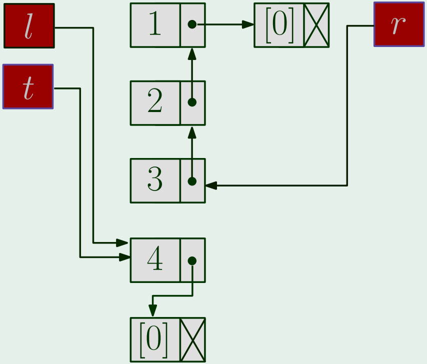
```
list rev(list l) {
  list *r;
  list *t;
  while (l->next != null){
    t = l->next;
    l->next = r;
    r = l;
    l = t;
  }
  return r;
}
```



REVERSING WELL-FORMED LISTS [Zanardini '08]

Reversing the list

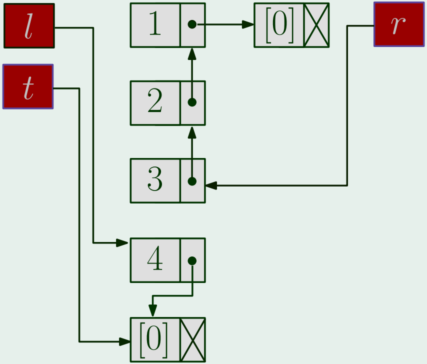
```
list rev(list l) {
  list *r;
  list *t;
  while (l->next != null){
    t = l->next;
    l->next = r;
    r = l;
    l = t;
  }
  return r;
}
```



REVERSING WELL-FORMED LISTS [Zanardini '08]

Reversing the list

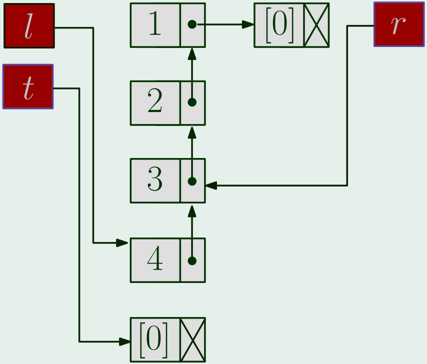
```
list rev(list l) {
  list *r;
  list *t;
  while (l->next != null){
    t = l->next;
    l->next = r;
    r = l;
    l = t;
  }
  return r;
}
```



REVERSING WELL-FORMED LISTS [Zanardini '08]

Reversing the list

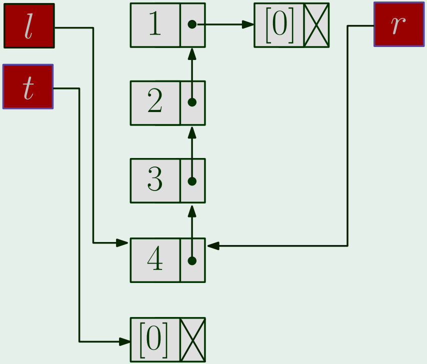
```
list rev(list l) {
  list *r;
  list *t;
  while (l->next != null){
    t = l->next;
    l->next = r;
    r = l;
    l = t;
  }
  return r;
}
```



Reversing Well-Formed Lists [Zanardini '08]

Reversing the list

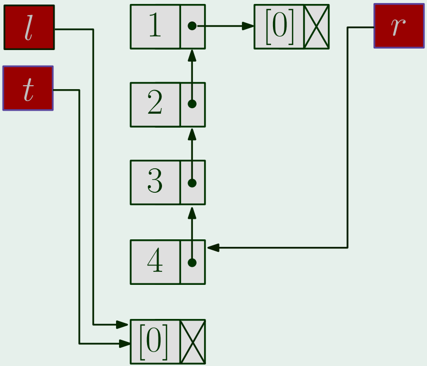
```
list rev(list l) {
  list *r;
  list *t;
  while (l->next != null){
    t = l->next;
    l->next = r;
    r = l;
    l = t;
  }
  return r;
}
```



Reversing Well-Formed Lists [Zanardini '08]

Reversing the list

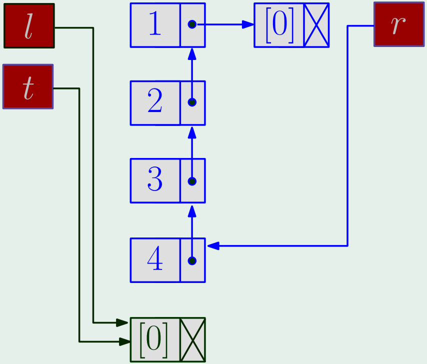
```
list rev(list l) {
  list *r;
  list *t;
  while (l->next != null){
    t = l->next;
    l->next = r;
    r = l;
    l = t;
  }
  return r;
}
```



REVERSING WELL-FORMED LISTS [Zanardini '08]

Reversing the list

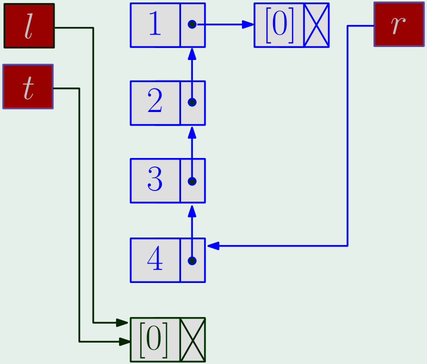
```
list rev(list l) {
  list *r;
  list *t;
  while (l->next != null){
    t = l->next;
    l->next = r;
    r = l;
    l = t;
  }
  return r;
}
```



REVERSING WELL-FORMED LISTS [Zanardini '08]

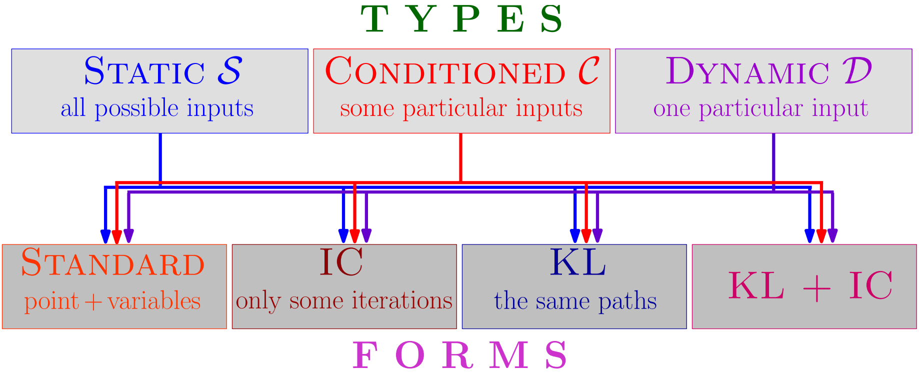
Reversing the list

```
list rev(list l) {
  list *r;
  list *t;
  while (l->next != null){
    t = l->next;
    l->next = r;
    r = l;
    l = t;
  }
  return r;
}
```



⇒ if r is well-formed before while, it is well-formed after while as well

Formal Framework [Binkley at al. '06]



Formal Framework [Binkley at al. '06]

$$\langle \sqsubseteq, \mathcal{E} \rangle$$

Formal Framework [Binkley at al. '06]

SEMANTIC CONSTRAINTS

maps slicing criteria to semantic equivalence relations

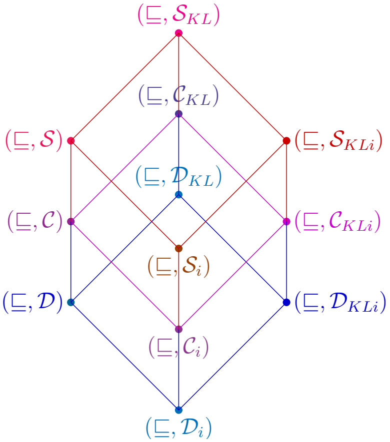
$\langle \sqsubseteq, \mathcal{E} \rangle$

TRADITIONAL SYNTACTIC ORDERING

$$Q \sqsubseteq P \Leftrightarrow \mathcal{F}(Q) \subseteq \mathcal{F}(P)$$

$\mathcal{F}(P)$ maps l to c iff P contains statement c at line l

Formal Framework [Binkley at al. '06]



Hierarchy of Existing Forms of Slicing

C
R
I
T
E
R
I
O
N

VARIABLES OF INTEREST: V

STATIC	CONDITIONED	DYNAMIC
all possible inputs	some particular inputs	one particular input

C
R
I
T
E
R
I
O
N

VARIABLES OF INTEREST: V

STATIC	CONDITIONED	DYNAMIC
all possible inputs	some particular inputs	one particular input
$\mathcal{I} = \mathbb{M}$	$\varnothing \subset \mathcal{I} \subseteq \mathbb{M}$	$\lvert\mathcal{I}\rvert = 1$

CRITERION

- INPUTS OF INTEREST: $\mathcal{I}$
- VARIABLES OF INTEREST: V

C
R
I
T
E
R
I
O
N

Inputs of Interest: $\mathcal{I}$

Variables of Interest: V

Iteration Count only particular iterations

Non-Iteration Count all iterations

CRITERION

- INPUTS OF INTEREST: $\mathcal{I}$
- VARIABLES OF INTEREST: V
- OCCURRENCES OF INTEREST: $\mathcal{O}$

$\mathcal{O} = n \times \wp(\mathbb{N})$

ITERATION COUNT only particular iterations

NON-ITERATION COUNT all iterations

$\mathcal{O} = n \times \mathbb{N}$

C
R
I
T
E
R
I
O
N

INPUTS OF INTEREST: $\mathcal{I}$

VARIABLES OF INTEREST: V

OCCURRENCES OF INTEREST: $\mathcal{O}$

KL	NON - KL
the same paths	even different paths

CRITERION

- INPUTS OF INTEREST: $\mathcal{I}$
- VARIABLES OF INTEREST: V
- OCCURRENCES OF INTEREST: $\mathcal{O}$
- PATH REQUIREMENT $\mathcal{L}$

$\mathcal{L} = true$

KL
the same paths

$\mathcal{L} = false$

NON - KL
even different paths

STATIC
all possible inputs
$\mathcal{I} = \mathbb{M}$

CONDITIONED
some particular inputs
$\varnothing \subset \mathcal{I} \subseteq \mathbb{M}$

DYNAMIC
one particular input
$|\mathcal{I}| = 1$

CRITERION

- **INPUTS OF INTEREST:** $\mathcal{I}$
- **VARIABLES OF INTEREST:** V
- **OCCURRENCES OF INTEREST:** $\mathcal{O}$
- **PATH REQUIREMENT** $\mathcal{L}$

$\mathcal{O} = n \times \wp(\mathbb{N})$

ITERATION COUNT
only particular iterations

NON-ITERATION COUNT all iterations

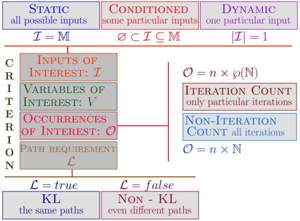
$\mathcal{O} = n \times \mathbb{N}$

$\mathcal{L} = true$

KL
the same paths

$\mathcal{L} = false$

NON - KL
even different paths



Generalized Slicing Criterion

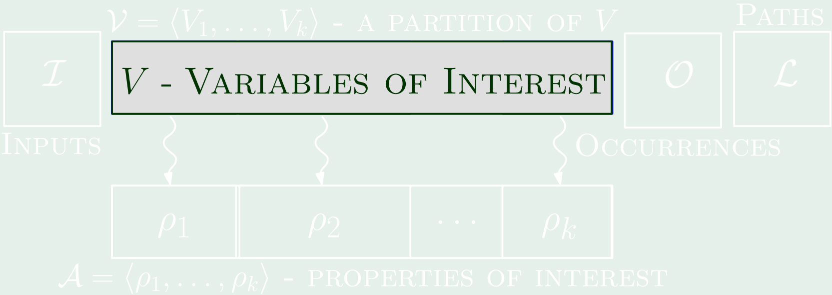
$$\mathcal{C} = \langle \mathcal{I}, \mathrm{V}, \mathcal{O}, \mathcal{L} \rangle,$$

where

- $\mathcal{I} \subseteq \mathbb{M}$ - set of INPUTS of interest,
- V - set of VARIABLES of interest,
- $\mathcal{O} \in n \times \wp(\mathbb{N})$ - set of OCCURRENCES of interest,
- $\mathcal{L} \in \{true, false\}$ - determines a KL form.

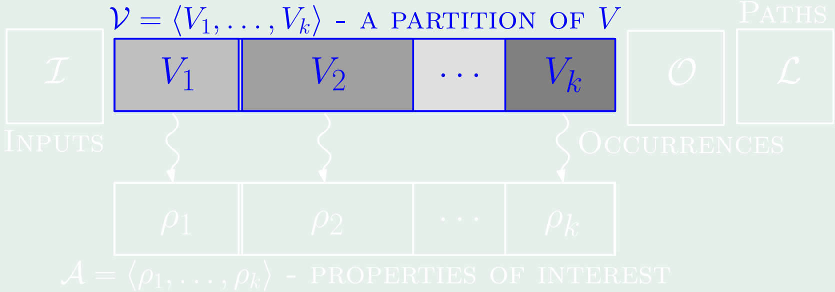
ABSTRACTING GENERALIZED SLICING CRITERION

Generalized abstract criterion



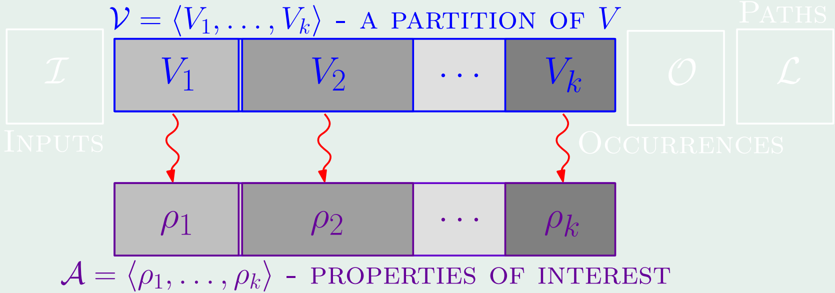
ABSTRACTING GENERALIZED SLICING CRITERION

Generalized abstract criterion



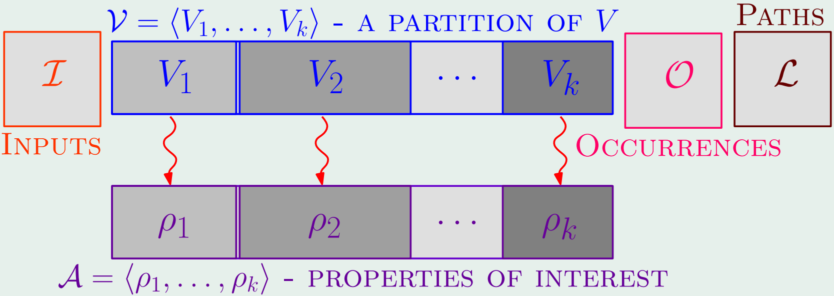
Abstracting generalized slicing criterion

Generalized abstract criterion



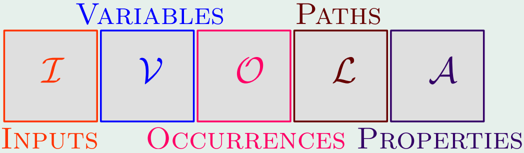
ABSTRACTING GENERALIZED SLICING CRITERION

Generalized abstract criterion



Abstracting generalized slicing criterion

Generalized abstract criterion



Example

$$Var = \{x_1, x_2, x_3, x_4\} \qquad V = \{x_1, x_2, x_3\}$$

Properties of interest: Sign^2 of $x_1 \times x_2$ and Par of x_3

$$\Rightarrow \mathcal{V} = \langle \{x_1, x_2\}, \{x_3\} \rangle \qquad \mathcal{A} = \langle \text{Sign}^2, \text{Par} \rangle$$

$$\text{Sign}^2(x, y) = \begin{cases} \text{POS} & \text{if } x * y > 0 \\ 0 & \text{if } x * y = 0 \\ \text{NEG} & \text{otherwise} \end{cases} \qquad \text{Par}(x) = \begin{cases} \text{EVEN} & \text{if } x \text{ is even} \\ \text{ODD} & \text{otherwise} \end{cases}$$

System States and Traces

```
begin
   read(n);
   i:=1;
   s:=0;
   p:=1;
   while (i<=n) do
   begin
      s:=s+i;
      p:=p*i;
      i:=i+1;
   end;
   write(s);
   write(p);
end;
```

$$\sigma = \{n \leftarrow 2\}$$

SYSTEM STATES AND TRACES

```
1   begin
2     read(n);
3     i:=1;
4     s:=0;
5     p:=1;
6     while (i<=n) do
7     begin
8       s:=s+i;
9       p:=p*i;
10      i:=i+1;
11    end;
12    write(s);
13    write(p);
14  end;
```

$$\sigma = \{n \leftarrow 2\}$$

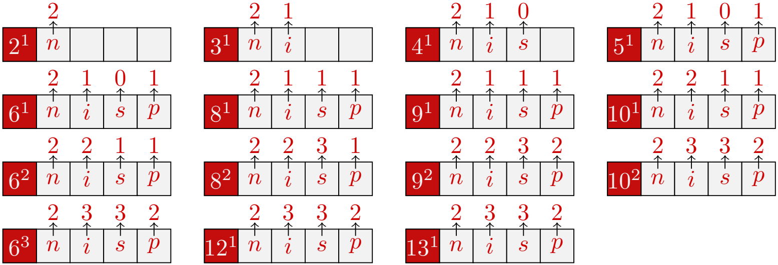
State	n	i	s	p
2^1	2			
3^1	2	1		
4^1	2	1	0	
5^1	2	1	0	1
6^1	2	1	0	1
8^1	2	1	1	1
9^1	2	1	1	1
10^1	2	2	1	1
6^2	2	2	1	1
8^2	2	2	3	1
9^2	2	2	3	2
10^2	2	3	3	2
6^3	2	3	3	2
12^1	2	3	3	2
13^1	2	3	3	2

SYSTEM STATES AND TRACES

L - ADDITIONAL POINTS OF INTERESTED

$$Proj'^{\alpha}_{(\mathcal{V},\mathcal{O},L,\mathcal{A})}(n,k,\sigma) \stackrel{\text{def}}{=} \begin{cases} (n, \sigma \upharpoonright^{\alpha} \mathcal{V}) & \text{if } (n,k) \in \mathcal{O} \\ (n, \sigma \upharpoonright^{\alpha} \varnothing) & \text{if } (n,k) \notin \mathcal{O} \wedge n \in L \\ \lambda & \text{otherwise} \end{cases}$$

$$\mathcal{V} = \langle \{i\}, \{s\} \rangle,\ \mathcal{O} = \{8\} \times \mathbb{N},\ L = \{6\},\ \mathcal{A} = \langle \textsc{Sign}, \textsc{Par} \rangle$$

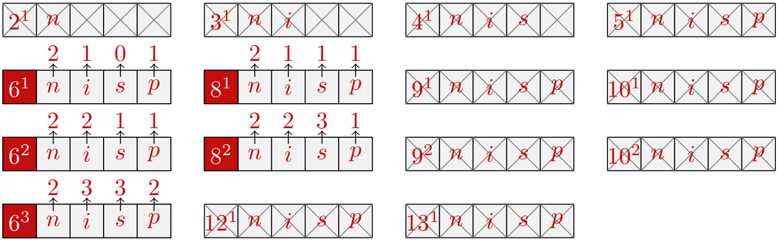


SYSTEM STATES AND TRACES

L - ADDITIONAL POINTS OF INTERESTED

$$Proj'^{\alpha}_{(\mathcal{V},\mathcal{O},L,\mathcal{A})}(n,k,\sigma) \stackrel{\text{def}}{=} \begin{cases} (n, \sigma \restriction^{\alpha} \mathcal{V}) & \text{if } (n,k) \in \mathcal{O} \\ (n, \sigma \restriction^{\alpha} \varnothing) & \text{if } (n,k) \notin \mathcal{O} \wedge n \in L \\ \lambda & \text{otherwise} \end{cases}$$

$$\mathcal{V} = \langle \{i\}, \{s\} \rangle, \ \mathcal{O} = \{8\} \times \mathbb{N}, \ L = \{6\}, \ \mathcal{A} = \langle \text{SIGN}, \text{PAR} \rangle$$

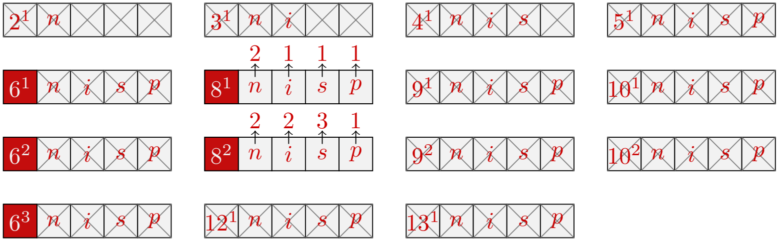


System States and Traces

L - Additional Points of Interested

$$Proj'^{\alpha}_{(\mathcal{V},\mathcal{O},L,\mathcal{A})}(n,k,\sigma) \stackrel{\text{def}}{=} \begin{cases} (n, \sigma \upharpoonright^{\alpha} \mathcal{V}) & \text{if } (n,k) \in \mathcal{O} \\ (n, \sigma \upharpoonright^{\alpha} \varnothing) & \text{if } (n,k) \notin \mathcal{O} \wedge n \in L \\ \lambda & \text{otherwise} \end{cases}$$

$$\mathcal{V} = \langle \{i\}, \{s\} \rangle,\ \mathcal{O} = \{8\} \times \mathbb{N},\ L = \{6\},\ \mathcal{A} = \langle \text{Sign}, \text{Par} \rangle$$

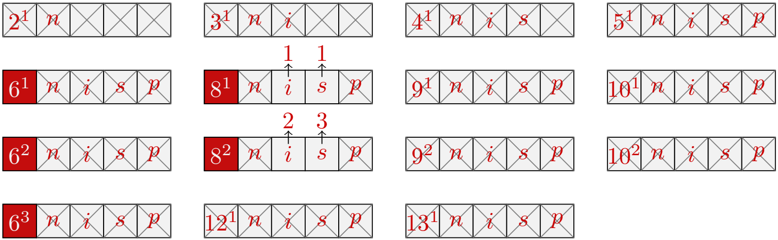


System States and Traces

L - Additional Points of Interested

$$Proj'^{\alpha}_{(\mathcal{V},\mathcal{O},L,\mathcal{A})}(n,k,\sigma) \stackrel{\text{def}}{=} \begin{cases} (n, \sigma \restriction^{\alpha} \mathcal{V}) & \text{if } (n,k) \in \mathcal{O} \\ (n, \sigma \restriction^{\alpha} \varnothing) & \text{if } (n,k) \notin \mathcal{O} \wedge n \in L \\ \lambda & \text{otherwise} \end{cases}$$

$$\mathcal{V} = \langle \{i\}, \{s\} \rangle, \mathcal{O} = \{8\} \times \mathbb{N}, L = \{6\}, \mathcal{A} = \langle \text{SIGN}, \text{PAR} \rangle$$

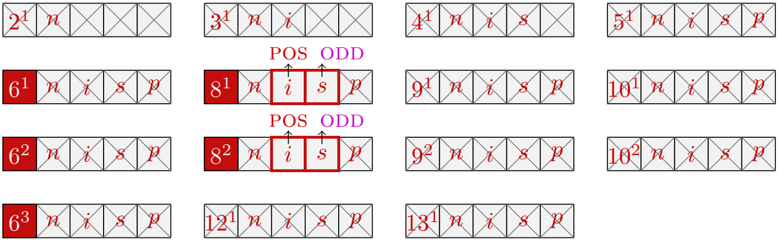


SYSTEM STATES AND TRACES

L - ADDITIONAL POINTS OF INTERESTED

$$Proj'^{\alpha}_{(\mathcal{V},\mathcal{O},L,\mathcal{A})}(n,k,\sigma) \stackrel{\text{def}}{=} \begin{cases} (n, \sigma \upharpoonright^{\alpha} \mathcal{V}) & \text{if } (n,k) \in \mathcal{O} \\ (n, \sigma \upharpoonright^{\alpha} \varnothing) & \text{if } (n,k) \notin \mathcal{O} \wedge n \in L \\ \lambda & \text{otherwise} \end{cases}$$

$$\mathcal{V} = \langle \{i\}, \{s\} \rangle, \ \mathcal{O} = \{8\} \times \mathbb{N}, \ L = \{6\}, \ \mathcal{A} = \langle \text{SIGN}, \text{PAR} \rangle$$



Abstract Unified Equivalence

- P, Q - executable programs,
- l_P, l_Q - sets of line numbers of P and Q
- $\mathcal{C}_{\mathcal{A}} = \langle \mathcal{I}, \mathcal{V}, \mathcal{O}, \mathcal{L}, \mathcal{A} \rangle$ - abstract criterion
- $L_{\mathcal{L}}(P, Q) = \mathcal{L} \ ? \ l_P \cap l_Q \ : \ \varnothing$
- P is Abstract Equivalent to Q ($P \ \ \mathcal{U}^{\mathcal{A}}(\mathcal{I}, \mathcal{V}, \mathcal{O}, L_{\mathcal{L}}, \mathcal{A}) \ \ Q$) iff

$$\forall \sigma \in \mathcal{I}. Proj^{\alpha}_{(\mathcal{V}, \mathcal{O}, L_{\mathcal{L}}, \mathcal{A})}(T^{\sigma}_{P}) = Proj^{\alpha}_{(\mathcal{V}, \mathcal{O}, L_{\mathcal{L}}, \mathcal{A})}(T^{\sigma}_{Q})$$

EXTENDED FRAMEWORK

Semantic Constraint

$$\mathcal{E}_{\mathcal{A}} \stackrel{\text{def}}{=} \lambda(\mathcal{I}, \mathcal{V}, \mathcal{O}, \mathcal{L}, \mathcal{A}).\, \mathcal{U}^{\mathcal{A}}(\mathcal{I}, \mathcal{V}, \mathcal{O}, L_{\mathcal{L}}, \mathcal{A})$$

$\langle \sqsubseteq, \mathcal{E}_{\mathcal{A}} \rangle$ - Representation of Abstract Forms of Slicing

EXTENDED FRAMEWORK

Semantic Constraint

$$\mathcal{E}_{\mathcal{A}} \stackrel{\text{def}}{=} \lambda(\mathcal{I}, \mathcal{V}, \mathcal{O}, \mathcal{L}, \mathcal{A}).\, \mathcal{U}^{\mathcal{A}}(\mathcal{I}, \mathcal{V}, \mathcal{O}, L_{\mathcal{L}}, \mathcal{A})$$

$\langle \sqsubseteq, \mathcal{E}_{\mathcal{A}} \rangle$ - Representation of Abstract Forms of Slicing

$\Updownarrow$

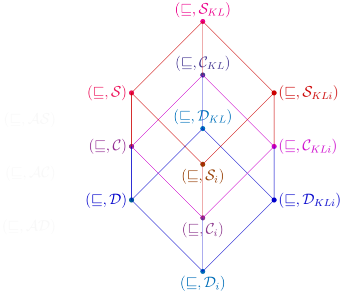
We have inserted Abstract Slicing in Formal Framework

$\Downarrow$ - Extended Theory

Enriched Hierarchy

Extended Framework

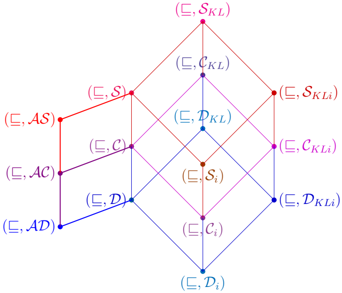
$\langle \sqsubseteq, \mathcal{E}_{\mathcal{A}} \rangle$ - Representation of Abstract Forms of Slicing
Enriched Hierarchy



EXTENDED FRAMEWORK

$\langle \sqsubseteq, \mathcal{E}_{\mathcal{A}} \rangle$ - Representation of Abstract Forms of Slicing

Enriched Hierarchy



SLICING VS DEPENDENCIES

Slicing

...extracts from programs the statements which are *relevant* for a given behaviour.

Dependency

...defines what *relevant* means.

Slicing vs dependencies

Slicing

...extracts from programs the statements which are *relevant* for a given behaviour.

Dependency

...defines what *relevant* means.

Example

$$\text{SYNTACTIC DEF-REF}: \begin{cases} x := y + 2z \\ \textbf{x} \text{ depends on } \textbf{y} \text{ and on } \textbf{z} \end{cases}$$

SLICING VS DEPENDENCIES

Slicing

...extracts from programs the statements which are *relevant* for a given behaviour.

Dependency

...defines what *relevant* means.

Example

SYNTACTIC DEF-REF :

$x := y + 2z$
x depends on **y** and on **z**

$x := z + y - y$
x depends on **y** and on **z**

SLICING VS DEPENDENCIES

Slicing

...extracts from programs the statements which are *relevant* for a given behaviour.

Dependency

...defines what *relevant* means.

Example

SEMANTIC : $\begin{cases} x := z + y - y \\ \textbf{x} \text{ depends on } \textbf{z} \text{ but it does NOT depend on } \textbf{y} \end{cases}$

SLICING VS DEPENDENCIES

Slicing

...extracts from programs the statements which are *relevant* for a given behaviour.

Dependency

...defines what *relevant* means.

Example

SEMANTIC :

$x := z + y - y$
x depends on **z** but it does NOT depend on **y**

$x := 2y$
x depends on **y**

SLICING VS DEPENDENCIES

Slicing

...extracts from programs the statements which are *relevant* for a given behaviour.

Dependency

...defines what *relevant* means.

Example

ABSTRACT SEMANTIC (PARITY) : $\begin{cases} x := 2y \\ \textbf{x} \text{ does NOT depend on } \textbf{y} \end{cases}$

SLICING VS DEPENDENCIES

Slicing

...extracts from programs the statements which are *relevant* for a given behaviour.

Dependency

...defines what *relevant* means.

Example

ABSTRACT SEMANTIC (PARITY) :

- $x := 2y$ — **x** does NOT depend on **y**
- $x := 2y + z$ — **x** depends on **z**

SLICING BY PRUNING PDG

Program Dependency Graphs (PDG) are defined by two kind of edges (s_1, s_2):

Control Flow Edge

s_1 represents a control predicate and s_2 represents a component of the program immediately nested within the predicate s_1;

Flow Dependence Edge

s_1 defines a variable x which is used in s_2
i.e., $x \in \mathbf{def}(s_1) \cap \mathbf{ref}(s_2)$,
and x is not further defined between s_1 and s_2;

SLICING BY PRUNING PDG

Program Dependency Graphs (PDG) are defined by two kind of edges (s_1, s_2):

Control Flow Edge

s_1 represents a control predicate and s_2 represents a component of the program immediately nested within the predicate s_1;

Flow Dependence Edge

s_1 defines a variable x which is used in s_2
i.e., $x \in \mathbf{def}(s_1) \cap \mathbf{ref}(s_2)$,
and x is not further defined between s_1 and s_2;

$\Downarrow$

Flow dependence edges = DIRECT FLOWS=DEF-REF dependencies
Control flow edges = INDIRECT FLOWS

PRUNING DEPENDENCIES

Kind of dependencies

- Data dependencies (Assignments);
- Control dependencies (Control structures)

PRUNING DEPENDENCIES

Kind of dependencies

- Data dependencies (Assignments) $\Rightarrow$ Direct flows;
- Control dependencies (Control structures) $\Rightarrow$ Indirect flows

PRUNING DEPENDENCIES

Kind of dependencies

- Data dependencies (Assignments) $\Rightarrow$ Direct flows;
- Control dependencies (Control structures) $\Rightarrow$ Indirect flows

We propose a PRUNING of *data dependencies*!

$\Downarrow$

STILL WE LOSE SOMETHING ABOUT CONTROL DEPENDENCIES!

PRUNING DEPENDENCIES

Kind of dependencies

- Data dependencies (Assignments) $\Rightarrow$ Direct flows;
- Control dependencies (Control structures) $\Rightarrow$ Indirect flows

Example

$$\textbf{if } (y + 2x \; mod \, 2) == 0 \textbf{ then } w := 0 \textbf{ else } w := 0$$

$\Rightarrow$ The guard DOES NOT DEPEND on x: OK

$\Rightarrow$ The variable w DOES NOT DEPEND on y: NO!

DERIVING ABSTRACT DEPENDENCIES

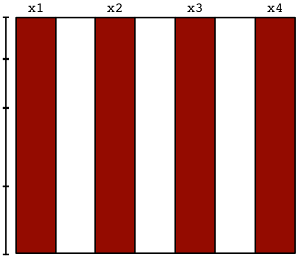
Systematic way to go through the (variables and states)-space

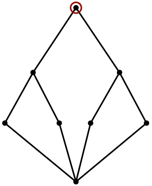
- Incrementally find the set X of variables which are enough to determine the value of e
- X determines e if any change to other variables can be ignored (needs to go into the state space)

DERIVING ABSTRACT DEPENDENCIES

Systematic way to go through the (variables and states)-space

- Incrementally find the set X of variables which are enough to determine the value of e
- X determines e if any change to other variables can be ignored (needs to go into the state space)

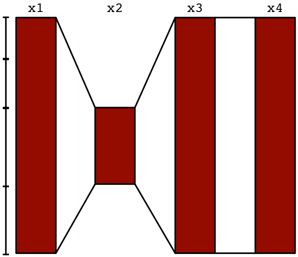


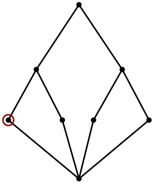


DERIVING ABSTRACT DEPENDENCIES

Systematic way to go through the (variables and states)-space

- Incrementally find the set *X* of variables which are enough to determine the value of *e*
- *X* determines *e* if any change to other variables can be ignored (needs to go into the state space)

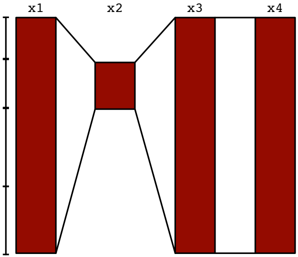


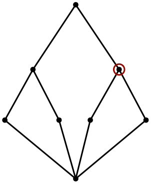


DERIVING ABSTRACT DEPENDENCIES

Systematic way to go through the (variables and states)-space

- Incrementally find the set X of variables which are enough to determine the value of e
- X determines e if any change to other variables can be ignored (needs to go into the state space)

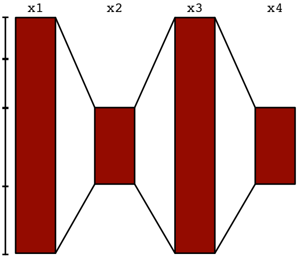


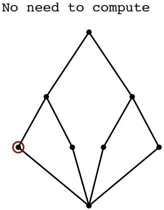


DERIVING ABSTRACT DEPENDENCIES

Systematic way to go through the (variables and states)-space

- Incrementally find the set *X* of variables which are enough to determine the value of *e*
- *X* determines *e* if any change to other variables can be ignored (needs to go into the state space)

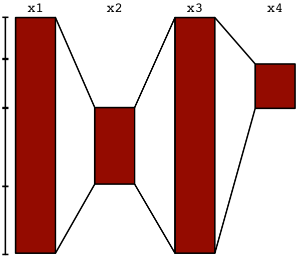


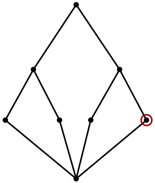


Deriving abstract dependencies

Systematic way to go through the (variables and states)-space

- Incrementally find the set *X* of variables which are enough to determine the value of *e*
- *X* determines *e* if any change to other variables can be ignored (needs to go into the state space)





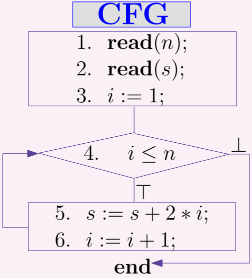
Slicing by abstracting CFG

- Start from a static slice of a program
- Derive an abstraction ρ from $\mathcal{C}_{\mathcal{A}}$ and construct abstract states using ρ
- Determine an abstract state graph ASG
- Abstract Slice corresponds to a pruned ASG

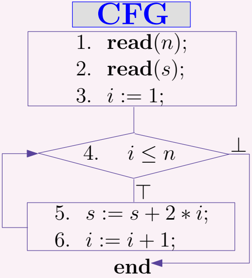
Example

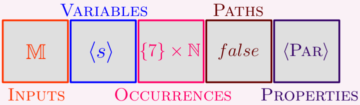
```
read(n);
read(s);
i := 1;
while (i<=n) do
   s := s + 2*i;
   i := i+1;
od
```

Example



Example

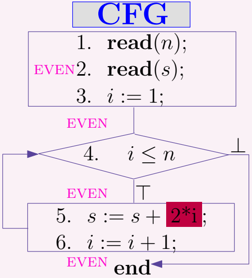


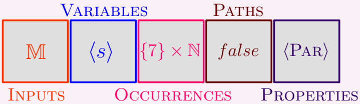


$\mathbb{M}$- all possible inputs

$$\text{PAR}(x) = \begin{cases} \text{EVEN} & \text{if } x \equiv_2 0 \\ \text{ODD} & \text{otherwise} \end{cases}$$

Example

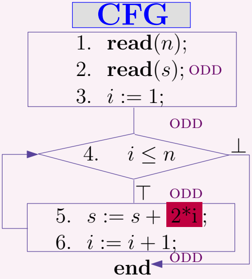


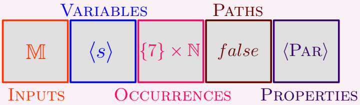


$\mathbb{M}$- all possible inputs

$$\text{PAR}(x) = \begin{cases} \text{EVEN} & \text{if } x \equiv_2 0 \\ \text{ODD} & \text{if } x \equiv_2 0 \end{cases}$$

Example

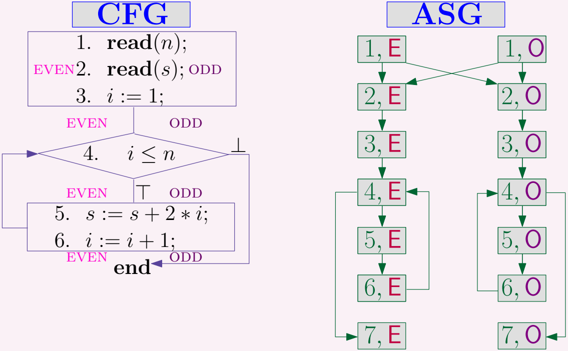




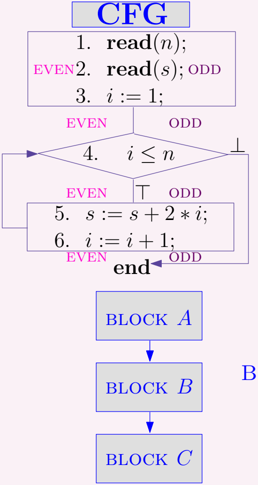
$\mathbb{M}$- all possible inputs

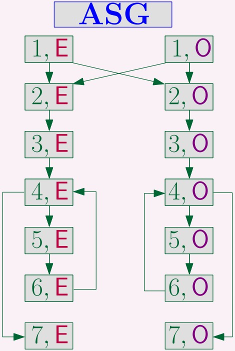
$$\text{PAR}(x) = \begin{cases} \text{EVEN} & \text{if } x \equiv_2 0 \\ \text{ODD} & \text{if } x \equiv_2 0 \end{cases}$$

Example


CFG
1. read(n);
EVEN 2. read(s); ODD
3. i := 1;
EVEN
ODD
4. i ≤ n
⊥
EVEN
⊤
ODD
5. s := s + 2 * i;
6. i := i + 1;
EVEN
ODD
end
ASG
1, E
1, O
2, E
2, O
3, E
3, O
4, E
4, O
5, E
5, O
6, E
6, O
7, E
7, O

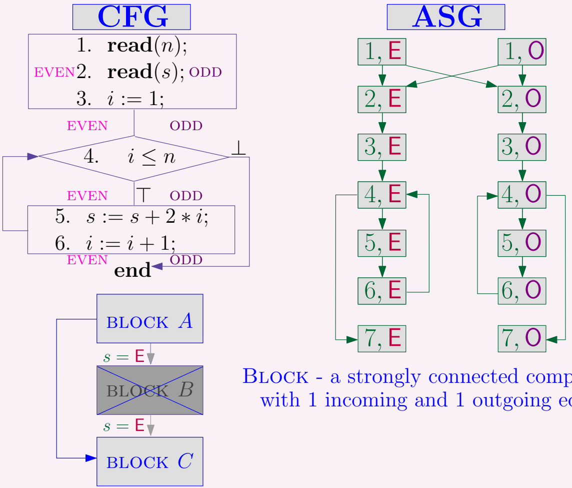
Example





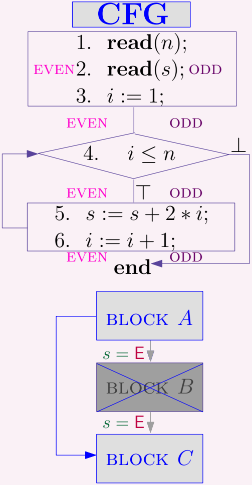
BLOCK - a strongly connected component with 1 incoming and 1 outgoing edge

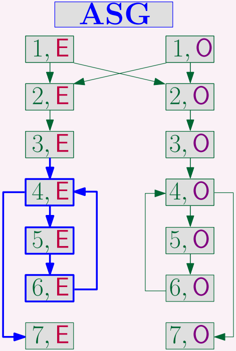
Example



BLOCK - a strongly connected component with 1 incoming and 1 outgoing edge

Example



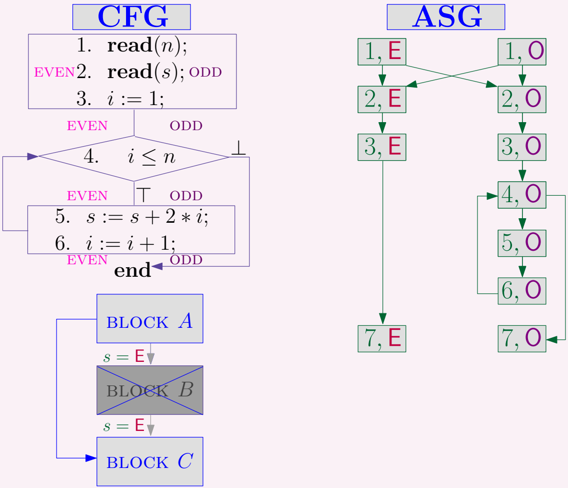


BLOCK ⟨4, E⟩, ⟨5, E⟩, ⟨6, E⟩

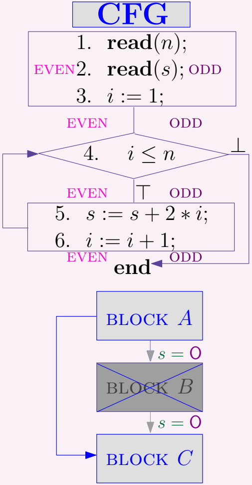
Input: $s =$ E Output: $s =$ E

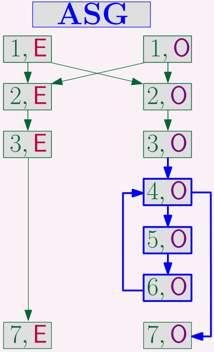
$\Rightarrow$ we can remove it

Example


CFG
1. read(n);
EVEN 2. read(s); ODD
3. i := 1;
EVEN
ODD
4. i ≤ n
⊥
⊤
EVEN
ODD
5. s := s + 2 * i;
6. i := i + 1;
EVEN
ODD
end
BLOCK A
s = E
BLOCK B
s = E
BLOCK C
ASG
1, E
1, O
2, E
2, O
3, E
3, O
4, O
5, O
6, O
7, E
7, O

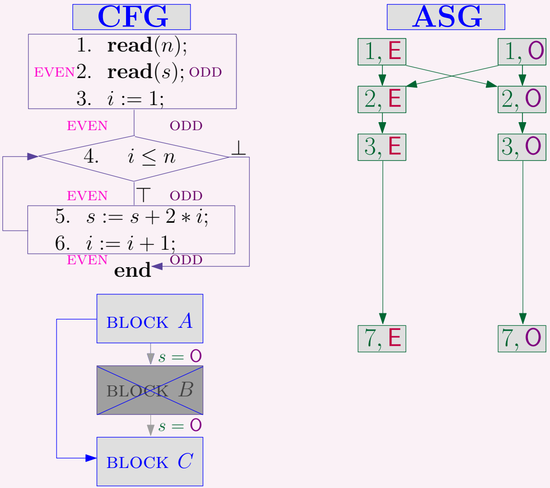
Example



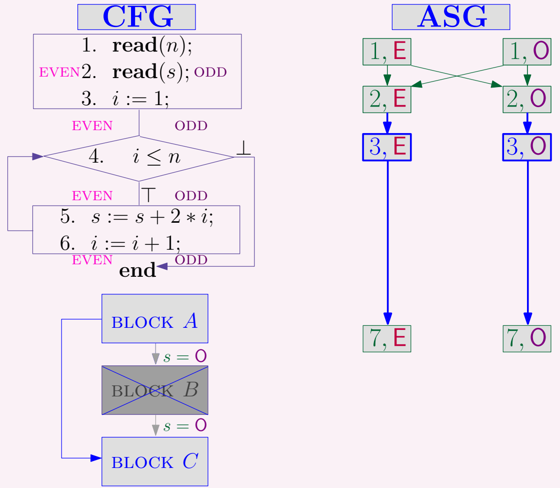


BLOCK $\langle 4, O\rangle, \langle 5, O\rangle, \langle 6, O\rangle$
Input: $s = O$ Output: $s = O$
$\Rightarrow$ we can remove it

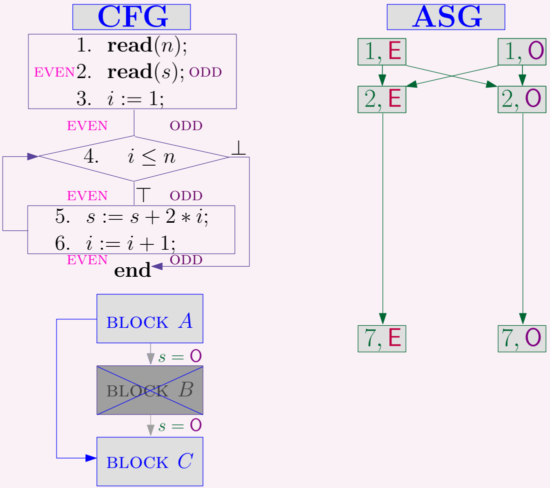
Example


CFG
1. read(n);
EVEN 2. read(s); ODD
3. i := 1;
EVEN ODD
4. i ≤ n
⊥
EVEN ⊤ ODD
5. s := s + 2 * i;
6. i := i + 1;
EVEN ODD
end
BLOCK A
s = O
BLOCK B
s = O
BLOCK C
ASG
1, E
1, O
2, E
2, O
3, E
3, O
7, E
7, O

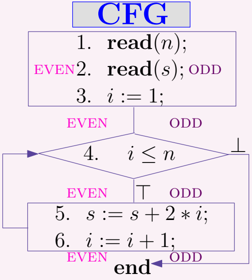
Example

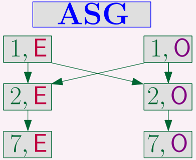

CFG
1. read(n);
EVEN 2. read(s); ODD
3. i := 1;
EVEN ODD
4. i ≤ n
⊥
EVEN ⊤ ODD
5. s := s + 2 ∗ i;
6. i := i + 1;
EVEN ODD
end
BLOCK A
s = O
BLOCK B
s = O
BLOCK C
ASG
1, E
1, O
2, E
2, O
3, E
3, O
7, E
7, O

Example


CFG
1. read(n);
EVEN 2. read(s); ODD
3. i := 1;
EVEN
ODD
4. i ≤ n
⊥
EVEN
⊤
ODD
5. s := s + 2 * i;
6. i := i + 1;
EVEN
ODD
end
BLOCK A
s = O
BLOCK B
s = O
BLOCK C
ASG
1, E
1, O
2, E
2, O
7, E
7, O

Example





1. **read**(n);
2. **read**(s);
7. **end**

CONCLUSIONS

Putting all together

- Generalized Slicing Criteria (Traditional and Abstract versions)
- Extension of Unified Formal Framework
- Formal definition of Abstract Program Slicing
- Semantic and constructive characterization of abstract dependencies
- First steps towards an implementation of abstract program slicing

Conclusions

Limitations

- If the property used for the construction of ASG is too much abstract, the Simple Approach returns the static slice
- This approach cannot be used for the extraction of dynamic and conditional slices: Extended Approach is one possible refinement of this algorithm
- Still a lot of work to do for obtaining a real implementation
- Also the semantic and constructive characterization of abstract dependencies is still far from its use in a real implementation of abstract slicing

Conclusions

Ideas for the Future

- Improvement and implementation of proposed algorithm(s)
- Obfuscation and Watermarking vs. Abstract Slicing
- Abstract slicing for malware detection