Abstract Program Slicing

Damiano Zanardini

CLIP, Universidad Politécnica de Madrid

PLID'08, Valencia, July 15th, 2008

Overview

An intuition

some basic ideas taken from published (to appear) work [SCAM'08]

Overview

An intuition

some basic ideas taken from published (to appear) work [SCAM'08]

The big issue

a brief discussion on practicality, usefulness, etc.

Overview

An intuition

some basic ideas taken from published (to appear) work [SCAM'08]

The big issue

a brief discussion on practicality, usefulness, etc.

After all, it's a workshop

future work, extensions, ideas

Introduction

Program slicing: $\mathcal{P}^S$ is the slice of $\mathcal{P}$ with respect to the criterion S

Introduction

Program slicing: $\mathcal{P}^S$ is the slice of $\mathcal{P}$ with respect to the criterion S

Observations

- S (the value of a set of vars) can be too restrictive, since the interest may be on some *data property* (e.g., the *nullity* of references)
- slices are sometimes too big for practical use (*debugging*, *program understanding*)
- dealing with properties *relaxes* the notion of dependency (abstract dependency, ANI, etc.)

Introduction

Program slicing: $\mathcal{P}^S$ is the slice of $\mathcal{P}$ with respect to the criterion S

Observations

- S (the value of a set of vars) can be too restrictive, since the interest may be on some *data property* (e.g., the *nullity* of references)
- slices are sometimes too big for practical use (*debugging*, *program understanding*)
- dealing with properties *relaxes* the notion of dependency (abstract dependency, ANI, etc.)

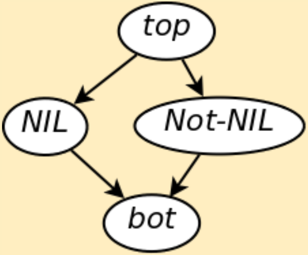
Slicing at the abstract level

might be successful in modelling interesting tasks and, at the same time, obtaining smaller slices

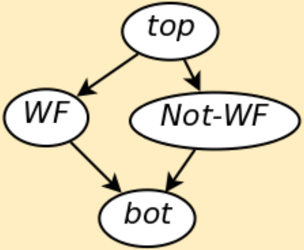
An example [SCAM'08]

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 interested are represented by abstract domains for *nullity* and *well-formedness*:



ρ_{nil}



ρ_{WF}

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

An example [SCAM'08]

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

```
$list1 := a_1;$
$list2 := a_2;$
while ($notLast(list1)$) {
  $tmp := list1.next;$
  $list1.next := list2;$
  $list2 := list1;$
  $list1 := tmp;$
}
if ($nil(list2) \vee illFormed(list2)$) {
  $res := nil$ } else { $res := list2$ }
```

An example [SCAM'08]

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

```
list1 := $a_1$;
list2 := $a_2$;
while (notLast(list1)) {
  tmp := list1.next;
  list1.next := list2;
  list2 := list1;
  list1 := tmp;
}
if (nil(list2) $\vee$ illFormed(list2)) {
  res := nil } else { res := list2 }
```

An example [SCAM'08]

Append-reverse: $\langle 1, 2, 3, 4, [0]\rangle ++ \langle 5, 6, [0]\rangle = \langle 4, 3, 2, 1, 5, 6, [0]\rangle$

```
list1 := $a_1$;
list2 := $a_2$;
while (notLast(list1)) {
  tmp := list1.next;
  list1.next := list2;
  list2 := list1;
  list1 := tmp;
}
if (nil(list2) $\vee$ illFormed(list2)) {
  res := nil } else { res := list2 }
```

An example [SCAM'08]

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

```
$list1 := a_1$;
$list2 := a_2$;
while ($notLast(list1)$) {
  $tmp := list1.next$;
  $list1.next := list2$;
  $list2 := list1$;
  $list1 := tmp$;
}
if ($nil(list2) \vee illFormed(list2)$) {
  $res := nil$ } else { $res := list2$ }
```

An example [SCAM'08]

Append-reverse: $\langle 1,2,3,4,[0]\rangle ++ \langle 5,6,[0]\rangle = \langle 4,3,2,1,5,6,[0]\rangle$

```
$list1 := a_1;$
$list2 := a_2;$
while ($notLast(list1)$) {
  $tmp := list1.next;$
  $list1.next := list2;$
  $list2 := list1;$
  $list1 := tmp;$
}
if ($nil(list2) \vee illFormed(list2)$) {
  $res := nil$ } else { $res := list2$ }
```

An example [SCAM'08]

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

```
$list1 := a_1$;
$list2 := a_2$;
while ($notLast(list1)$) {
  $tmp := list1.next$;
  $list1.next := list2$;
  $list2 := list1$;
  $list1 := tmp$;
}
if ($nil(list2) \vee illFormed(list2)$) {
  $res := nil$ } else { $res := list2$ }
```

An example [SCAM'08]

Append-reverse: $\langle 1,2,3,4,[0]\rangle ++ \langle 5,6,[0]\rangle = \langle 4,3,2,1,5,6,[0]\rangle$

```
$list1 := a_1$;
$list2 := a_2$;
while ($notLast(list1)$) {
  $tmp := list1.next$;
  $list1.next := list2$;
  $list2 := list1$;
  $list1 := tmp$;
}
if ($nil(list2) \vee illFormed(list2)$) {
  $res := nil$ } else { $res := list2$ }
```

An example [SCAM'08]

Append-reverse: $\langle 1,2,3,4,[0]\rangle ++ \langle 5,6,[0]\rangle = \langle 4,3,2,1,5,6,[0]\rangle$

```
$list1 := a_1;$
$list2 := a_2;$
while $(notLast(list1))$ {
  $tmp := list1.next;$
  $list1.next := list2;$
  $list2 := list1;$
  $list1 := tmp;$
}
if $(nil(list2) \lor illFormed(list2))$ {
  $res := nil$ } else { $res := list2$ }          $\mathcal{A}_{\rho_{nil}}(res)$
```

An example [SCAM'08]

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

```
$list1 := a_1$;
$list2 := a_2$;
while ($notLast(list1)$) {
  $tmp := list1.next$;
  $list1.next := list2$;
  $list2 := list1$;
  $list1 := tmp$;
}                                                    $\mathcal{A}_{\rho_{\mathrm{WF}}}(list2)$
if ($nil(list2) \vee illFormed(list2)$) {
  $res := nil$ } else { $res := list2$ }             $\mathcal{A}_{\rho_{nil}}(res)$
```

An example [SCAM'08]

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

```
$list1 := a_1$;
$list2 := a_2$;
while ($notLast(list1)$) {
  $tmp := list1.next$;
  $list1.next := list2$;                                  …
  $list2 := list1$;
  $list1 := tmp$;
}                                                         $\mathcal{A}_{\rho_{\mathrm{WF}}}(list2)$
if ($nil(list2) \vee illFormed(list2)$) {
  $res := nil$ } else { $res := list2$ }                  $\mathcal{A}_{\rho_{nil}}(res)$
```

An example [SCAM'08]

Append-reverse: $\langle 1, 2, 3, 4, [0]\rangle ++ \langle 5, 6, [0]\rangle = \langle 4, 3, 2, 1, 5, 6, [0]\rangle$

```
$list1 := a_1;$                                                   $any$
$list2 := a_2;$                                                   $\mathcal{A}_{\rho_{\mathrm{WF}}}(list2)$
while ($notLast(list1)$) {
  $tmp := list1.next;$
  $list1.next := list2;$                                          $\ldots$
  $list2 := list1;$
  $list1 := tmp;$
}                                                                 $\mathcal{A}_{\rho_{\mathrm{WF}}}(list2)$
if ($nil(list2) \vee illFormed(list2)$) {
  $res := nil$ } else { $res := list2$ }                          $\mathcal{A}_{\rho_{nil}}(res)$
```

An example [SCAM'08]

Append-reverse: $\langle 1,2,3,4,[0]\rangle ++ \langle 5,6,[0]\rangle = \langle 4,3,2,1,5,6,[0]\rangle$

```
$list2 := a_2$;


if ($nil(list2) \vee illFormed(list2)$) {
  $res := nil$ } else { $res := list2$ }
```

Basic ideas

When a command can be removed

- it *preserves* some *property* (as $x := x + 2$ preserves the *parity* of x)
- such property was obtained by *propagating* the slicing criterion backwards *from the end of the program* (WLOG) (e.g., in the example, the final nullity of *res* is propagated backwards to the well-formedness of *list2*)

A semantic characterization of these requirements can be given

Basic ideas

Abstract semantics and slicing criteria

- an abstract semantics $[\![ ]\!]^{\#}_{S}$ is *induced* by the slicing criterion S, which specifies the property to be preserved by the slicing

$$\forall\sigma.\ [\![P]\!]^{\#}_{S}(\sigma) = [\![P^{S}]\!]^{\#}_{S}(\sigma)$$

- criterions are expressed as *agreements* between two states: $\mathcal{A}(\sigma_1, \sigma_2)$ means that both states behave the same w.r.t. the abstract property specified by $\mathcal{A}$
- S requires the output of the program and of the slice to agree on $\mathcal{A}$:

$$\mathcal{A}\left([\![\mathcal{P}]\!](\sigma), [\![\mathcal{P}^{S}]\!](\sigma)\right)$$

Basic ideas

Which kind of machinery is needed, in practice

- a way to compute when a command does not make a difference (is *invariant*) on the abstract property

  $$\mathcal{A}([\![C]\!](\sigma),\ \sigma)$$

  saying that $\mathcal{A}$ is invariant on C means that C cannot be distinguisehd from *skip* as regards the property of interest
- the computation of such results must be *sound*, i.e., the invariance must be *guaranteed*

Basic ideas

Which kind of machinery is needed, in practice

- how criteria (properties) are propagated backwards through the code, i.e., given $\mathcal{A}'$ after C, find the best $\mathcal{A}$ before C such that

$$\forall \sigma_1, \sigma_2. \quad \mathcal{A}(\sigma_1, \sigma_2) \;\Rightarrow\; \mathcal{A}'(\llbracket C \rrbracket(\sigma_1), \llbracket C \rrbracket(\sigma_2))$$

- these are example (Hoare-like) *rules* for propagating agreements through a program

$$\frac{\{\mathcal{A}\}\ C\ \{\mathcal{A}'\} \quad \{\mathcal{A}'\}\ C'\ \{\mathcal{A}''\}}{\{\mathcal{A}\}\ C\ ;\ C'\ \{\mathcal{A}''\}}\ \textsc{A-concat}$$

$$\frac{\{\mathcal{A} \sqcap \mathcal{A}_b\}\ C_w\ \{\mathcal{A} \sqcap \mathcal{A}_b\}}{\{\mathcal{A} \sqcap \mathcal{A}_b\}\ \mathit{while}\ (b)\ \mathit{do}\ C_w\ \{\mathcal{A} \sqcap \mathcal{A}_b\}}\ \textsc{A-while}$$

Basic ideas

Which kind of machinery is needed, in practice

- when a piece of data *depends* on other data at the abstract level
  - to track the flow of information in the case of *assignments*
  - *syntax* is not enough, not even a reasonable approximation: the difference between *concrete* and *abstract* lies in the *semantics*
  - e.g., the sign of xy^2 only depends on x
- in this direction:
  - *binary* domains (null/non-null, zero/non-zero, etc.)?
  - in this case, *independence* of (a property of) an expression from (a property of) a variable means that knowing the answer to *questions* about x is not needed to answer about the result
  - e.g., if answering to *is x null?* is irrelevant to the question *is e positive?*, then the sign of e does not depend on the nullity of x
  - using BDDs would be practical?

Trying to think general

What we have

the presented version of slicing reasons about abstractions (properties) of the (final) value of variables

- in one sense, $[\![]\!]^{\#}_{S}$ is an abstraction of the *denotational semantics*

Trying to think general

What we have

the presented version of slicing reasons about abstractions (properties) of the (final) value of variables

- in one sense, $[\![]\!]^{\#}_{S}$ is an abstraction of the *denotational semantics*

What we (fore)see

yet, what if *non-denotational* information is included in the abstraction?

- e.g., suppose $[\![]\!]^{\#}_{S}$ be an abstraction of the *trace semantics*
- this may allow to reason about the *history* of the computation

Preserving program properties

Possibilities

this approach could possibly allow reasoning about complex, functional properties of a program

- *termination*: $\mathcal{P}^S$ $\{$must$|$can$|$cannot$\}$ terminate iff $\mathcal{P}$ $\{$must$|$can$|$cannot$\}$ terminate
- *information flow*: x $\{$must$|$can$|$cannot$\}$ flow to y in $\mathcal{P}^S$ iff it $\{$must$|$can$|$cannot$\}$ flow to y in $\mathcal{P}$

Preserving program properties

Possibilities

this approach could possibly allow reasoning about complex, functional properties of a program

- *termination*: $\mathcal{P}^S$ $\{$must$|$can$|$cannot$\}$ terminate iff $\mathcal{P}$ $\{$must$|$can$|$cannot$\}$ terminate
- *information flow*: x $\{$must$|$can$|$cannot$\}$ flow to y in $\mathcal{P}^S$ iff it $\{$must$|$can$|$cannot$\}$ flow to y in $\mathcal{P}$

A general framework

- in all cases, the problem is to compute invariance, to propagate properties, to deal with abstract dependence
- clearly, formalizing and manipulating the property of interest can be quite tricky

Preserving program properties

An example

suppose h *cannot* flow to l while l'' *must* flow to l

$$l \;:=\; h{+}l''$$

$$\ldots$$

$$if\ (b)\ \ \{\ l \;:=\; l'{+}1\ \}\ else\ \ \{\ l \;:=\; l'{+}l''\ \}$$

$$\ldots$$

- in this case, removing the conditional would break the first requirement, since there *could* be a flow from h to l
- of course, the absence of *must-flow* requirements would imply that the empty program is always a correct slice (unless *self-flows* are forbidden)

Related work

To be completed...

- ANI (Giacobazzi & Mastroeni)
- abstract dependencies (Rival, Mastroeni & Zanardini)
- symbolic execution (King)
- conditioned slicing (Canfora et al.)
- logic for information flow (Amtoft & Banerjee)
- *invariance on commands*?
- ...

References

MZ08

I. Mastroeni and D. Zanardini. **Data Dependencies and Program Slicing: from Syntax to Abstract Semantics**. In *Proc. ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM)*, 2008.

Zan08

D. Zanardini. **The Semantics of Abstract Program Slicing**. In *Proc. International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2008. To appear.