The Semantics of Abstract Program Slicing

Damiano Zanardini

CLIP, Technical University of Madrid

SCAM'08, Beijing, September 28th, 2008

Introduction

The topic

To slice programs with respect to some *properties* of their values, not the values themselves

Motivations

- a slice can be too big for practical purposes
- we are often interested in properties of data: e.g., why a given reference is null at a given program point

The outcome

Possibly smaller slices, which are sound with respect to the property of interest

Tools and Methodology

Abstract Interpretation

- to model properties as abstract domains
- to deal with static analysis issues

Quite a theoretical focus

The *technical* machinery consists of a *rule system*, static analysis for *invariance*, and an *algorithm schema*

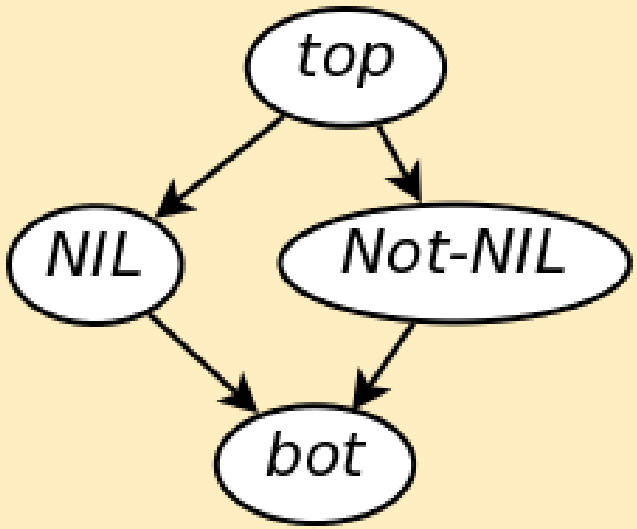
:-) the framework is proven to be *sound*

:-(nothing implemented so far, no focus on efficiency

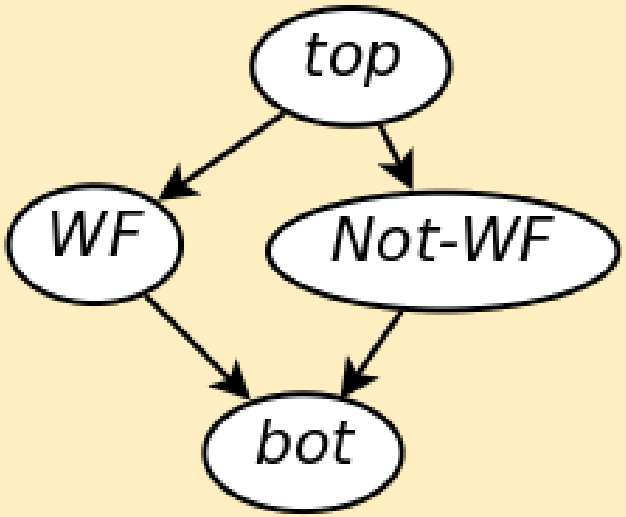
Example

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}

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

Example

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** { $res := list2$ }

$\mathcal{A}_{\rho_{nil}}(res)$?

Slicing criterion

The focus is on why *res* is null (or not) at the end

Example

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** { $res := list2$ }

$\mathcal{A}_{\rho_{\mathrm{WF}}}(list2)?$
$\uparrow$
$\mathcal{A}_{\rho_{nil}}(res)?$

Slicing criterion

The focus is on why *res* is null (or not) at the end

Example

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 $(notLast(list1))$ {
  $tmp := list1.next;$
  $list1.next := list2;$ (does nothing!)
  $list2 := list1;$
  $list1 := tmp;$
}
if $(nil(list2) \vee illFormed(list2))$ {
  $res := nil$ } **else** { $res := list2$ }

$\mathcal{A}_{\rho_{\mathrm{WF}}}(list2)?$
$\uparrow$
$\mathcal{A}_{\rho_{nil}}(res)?$

Slicing criterion

The focus is on why *res* is null (or not) at the end

Example

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 $(notLast(list1))$ {	
$tmp := list1.next;$	$\uparrow$
$list1.next := list2;$	(does nothing!)
$list2 := list1;$	$\uparrow$
$list1 := tmp;$	
}	$\mathcal{A}_{\rho_{\mathrm{WF}}}(list2)?$
if $(nil(list2) \vee illFormed(list2))$ {	$\uparrow$
$res := nil$ } **else** { $res := list2$ }	$\mathcal{A}_{\rho_{nil}}(res)?$

Slicing criterion

The focus is on why *res* is null (or not) at the end

Example

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** { $res := list2$ }

What's up

What happened?

We could remove a considerable part of the code because the reason for well-formedness of *list2* is elsewhere

When a command can be removed

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

These requirements can be given a *semantic* characterization

My questions (i.e., *your* questions)

Going into practice: how can we do it?

- decide when commands are *invariant* on the abstract property
- propagate *questions* backwards (i.e., implement the rule system)
- infer data *dependencies* at the abstract level
- implement the algorithm

What is this all about?

Is developing such a framework, and keeping an eye on theory-related issues

- desirable
- useful
- of any interest