

Shape Analysis – part II

Dror Weiss

Embedding and Canonical abstraction

Embedding

Embedding is a method of representing information about large logical structures in smaller ones in a conservative manner.

A logical structure B can be embedded onto a structure S via an onto function f (we note it as $B \sqsubseteq^f S$) if the basic relations (predicates) are preserved:

$$p^B(u_1, \dots, u_k) \sqsubseteq p^S(f(u_1), \dots, f(u_k))$$

All the values of the predicates in B are contained in the values of the predicated in S.

We call S a *tight embedding* of B with respect to f when S does not lose unnecessary information: $p^S(u^{\#}_1, \dots, u^{\#}_k) = \sqcup \{p^B(u_1, \dots, u_k) \mid f(u_1) = u^{\#}_1, \dots, f(u_k) = u^{\#}_k\}$

Canonical abstraction is tight embedding by its definition. When embed the concrete structure that has 2-value logic into the abstract structure that has 3-value logic, the abstract value of a predicate will only be 1/2 in cases where it can be both 0 and 1 in the concrete structure.

In the canonical abstraction used in shape analysis, we group individuals iff they have exactly the same values in all unary predicates. Suppose we have n unary predicates we may have up to 2^n groups of individuals. The abstraction is expensive, however, this is still more compact than the concrete heap-structure which has unbounded number of individuals.

The embedding theorem

When $B \sqsubseteq^f S$ that is $p^B(u_1, \dots, u_k) \sqsubseteq p^S(f(u_1), \dots, f(u_k))$ then every formula φ is preserved:

If $\llbracket \varphi \rrbracket = 1$ in S, then $\llbracket \varphi \rrbracket = 1$ in B

If $\llbracket \varphi \rrbracket = 0$ in S, then $\llbracket \varphi \rrbracket = 0$ in B

If $\llbracket \varphi \rrbracket = \frac{1}{2}$ in S, then $\llbracket \varphi \rrbracket$ can be either 0 or 1 in B

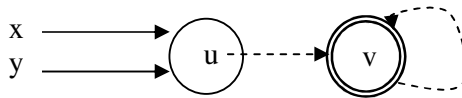
The significance of this theorem is that when we define correct concrete semantic the analysis will be sound. The conservative nature on the predicates defined in the embedding makes all the formulas conservative as well. That is correct because the operators and quantifiers are monotone - we make mistakes only in the conservative direction.

Improving precision

Canonical abstraction is not precise enough

Let's consider the following example:

We have a linked list with x and y point to its head. The abstraction of the situation will look like this:



Or more precisely, we have 2 individuals:

Node	$x(\text{node})$	$y(\text{node})$
u	1	1
v	0	0

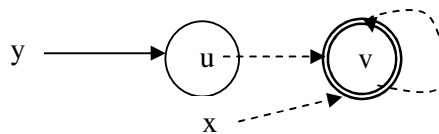
And the next relation:

From	To	Next(from,To)
v	v	1/2
v	u	0
u	v	1/2
u	u	0

Now we apply the statement $x = x \rightarrow next$.

The logical semantic of this statement is $ix(v) = \exists w: x(w) \wedge next(w, v)$.

For the node u the value of x will be 0 because no node may point to it. For the node v the value of x will be 1/2 because u has x predicate and may (1/2) point to v . The abstraction we get after performing the statement looks like this:



We still have 2 individuals:

Node	$x(\text{node})$	$y(\text{node})$
u	0	1
v	1/2	0

And the next relation is now:

From	To	Next(from,To)
v	v	1/2
v	u	0
u	v	1/2
u	u	0

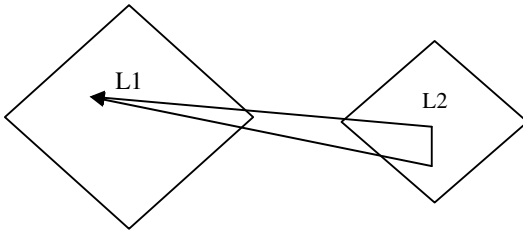
The abstraction we got is sound, but is not very precise – it does not represent the fact that now $y \rightarrow next$ points to the same heap location as x . Since we only considered the unary predicates in the canonical abstraction when grouping the individuals, we lost relational operation that may be important.

Semantic reduction

Semantic reduction is an operation that is done on an abstract domain element that returns another abstract domain element that is lower in the lattice, but represents the same concrete set of states. In other words, semantic reduction moves to a more precise abstract state. More formally, given a Galois connection $(L_1, \alpha, \gamma, L_2)$ an operation $op: L_2 \rightarrow L_2$ is a semantic reduction iff:

- $\forall l \in L_2: op(l) \sqsubseteq l$
- $\gamma(op(l)) = \gamma(l)$

Visual illustration:



The lower element in L2 is the semantic reduction of the upper element, but they represent the same concrete states.

The Focus Operation

The focus operation is a type of semantic reduction that increases the precision of the abstraction. The focus is a formula we want to represent more precisely in the abstraction. After focus on a formula is performed, the formula will have a value of 0 or 1 for each individual, but never 1/2. The focus operation can be seen as a partial concretization of the abstract state.

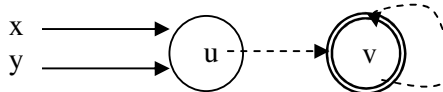
More formally focus is a function $\text{Focus}: Formula \rightarrow (P(3 - Struct) \leftrightarrow P(3 - Struct))$ s.t. for every formula φ :

- $\text{Focus}(\varphi)(X)$ yields a structure in which φ evaluates to a definite value (0 or 1) in all assignments
- $\text{Focus}(\varphi)$ is a semantic reduction

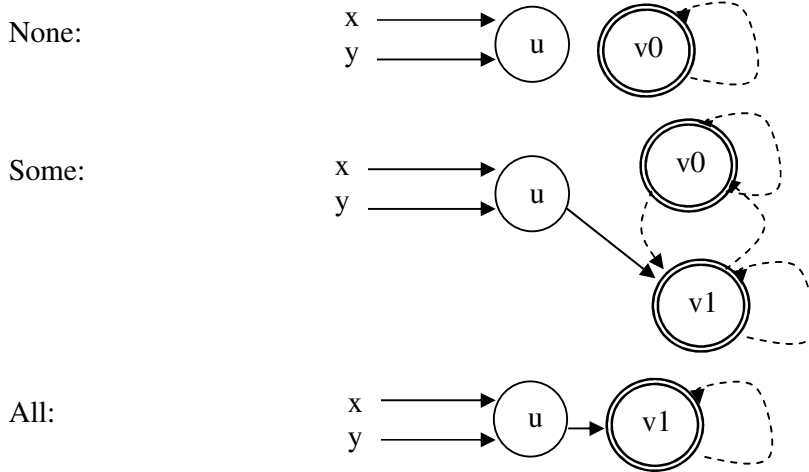
The focus operation can be applied on the abstract state before or after performing abstract transfer functions.

Let's consider the previous example, now with focus on $x \rightarrow next$ applied before the transfer function.

Before focus is applied the summary node has 1/2 on the $x \rightarrow next$ formula.



We perform the focus and split the summary node to groups according to the value they have for $x \rightarrow next$. We get 3 possible structures for the cases that none, some or all of the concrete locations which the summary node represents have $x \rightarrow next$:

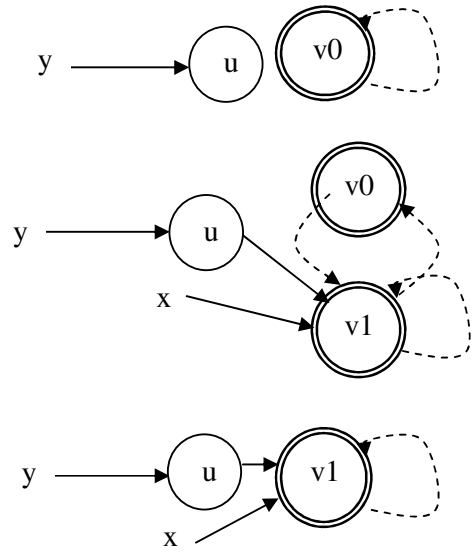


Case	Node	x(node)	y(node)	x->n
None	u	1	1	0
	v0	0	0	0
Some	u	1	1	0
	v1	0	0	1
	v0	0	0	0
All	u	1	1	0
	v1	0	0	1

Here's the next relation for all the cases. We had to adjust the next predicate to make the focused formula $x \rightarrow next$ 1 or 0 for v1 and v0 respectively. Each formula which is focused induces update formulas for predicates used in it.

Case	From	To	next(from,To)
None	u	u	0
	u	v0	0
	v0	u	0
	v0	v0	1/2
Some	u	u	0
	u	v0	0
	u	v1	1
	v0	u	0
	v0	v0	1/2
	v0	v1	1/2
	v1	u	0
	v1	v0	1/2
	v1	v1	1/2
	All	u	u
u		v1	1
v1		u	0
v1		v1	1/2

Now we apply the statement $x = x \rightarrow next$:
 =they have for $x \rightarrow next$:



Case	Node	$x(node)$	$y(node)$
None	u	0	1
	v0	0	0
Some	u	0	1
	v0	0	0
	v1	1	0
All	u	0	1
	v1	1	0

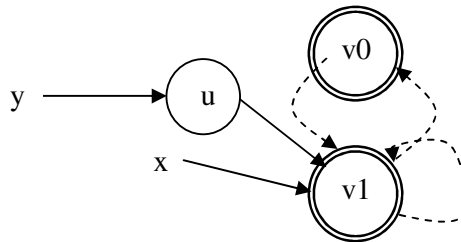
The next predicate hasn't changed because the statement semantic doesn't update it.

Now we use the canonical abstraction again to get the abstract element for this structure:

Node	x(node)	y(node)
u	0	1
v0	0	0
v1	1	0

From	To	next(from,To)
u	u	0
u	v0	0
u	v1	1
v0	u	0
v0	v0	1/2
v0	v1	1/2
v1	u	0
v1	v0	1/2
v1	v1	1/2

Or graphically:



We finally got a structure that reflects the fact that x points to the element next to y. because y(u) is 1, x(v1) is 1 and next(u,v1) is 1. We still see that this structure is not as optimal as it could be – v1 is a summary node while it is clear that it must be a single heap location, since it is pointed by x and the next of u.

Coercion operation

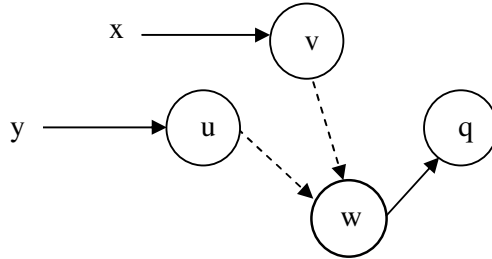
As we've seen in the recent example, the result we got wasn't as precise as possible. We didn't exploit our knowledge of heap pointers to eliminate infeasible states and get a more accurate structure. The idea of Coercion is to formulate our knowledge of the problem domain into constraints. We use the constraints to reduce the set of possible concrete states – this is also a kind of semantic reduction. Constraints represent interrelation between predicates. Additional instrumentation may be required to create more constraints. A constraint solver is applied to find additional structure properties. Examples:

- Equality (two locations pointed by same stack pointer are the same location)– for each unary predicate x: $x(v1) \wedge x(v2) \rightarrow eq(v1, v2)$
- Equality (two locations pointed by same heap pointer are the same location) - $n(v, v1) \wedge n(v, v2) \rightarrow eq(v1, v2)$

- Heap sharing relation (v is pointed by two non-equal heap locations) -
 $n(v1, v) \wedge n(v2, v) \wedge \neg eq(v1, v2) \leftrightarrow is(v)$

The constraint solver can use the constraints to derive additional relations on the structure.

Consider the following case:



and let's consider the following constraints:

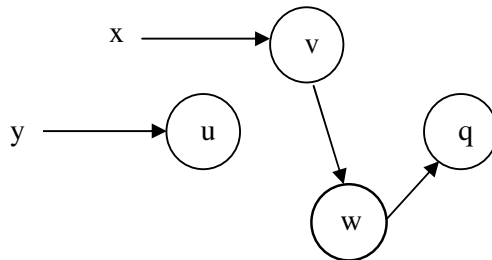
1. $n^*(v, q)$
2. $\neg is(w)$

The constraint solver can deduce –

$$n^*(v, q) \Rightarrow n(v, w)$$

$$n(v, w) \wedge \neg is(w) \Rightarrow \neg n(u, w)$$

So we can simplify the structure to:



Example – instrumentation of linked list traversal

Consider the following program which traverses the pointer x along a linked list with no cycles.

1. $x=y$
2. while ($x \neq \text{NULL}$)
3. $x=x \rightarrow \text{next}$

We want to analyze the structure of the heap when the program executes. In particular, we want our analysis to capture the fact that within the loop, x divides the list to 2 parts:

1. Locations reachable from y but not from x
2. Locations reachable from both y and x

The instrumentation we need is the reachability from variable via predicate:

$$r[n, y](v) = \exists w: y(w) \wedge n^*(w, v)$$

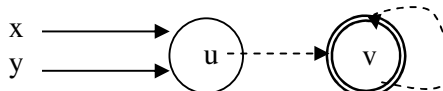
We instrument 2 new unary predicates rx and ry to represent reachability from x and y respectively.

We also would like to exploit the knowledge that the list is non-cyclic so we instrument the heap-sharing relation is :

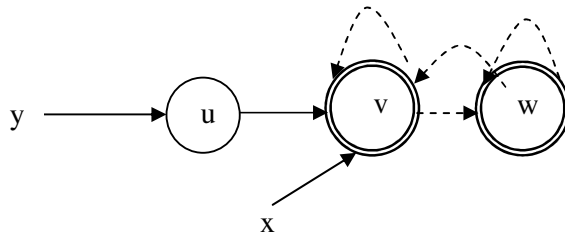
$$\text{next}(v1, v) \wedge \text{next}(v2, v) \wedge \neg eq(v1, v2) \leftrightarrow is(v)$$

$$ry(v) \rightarrow \neg is(v)$$

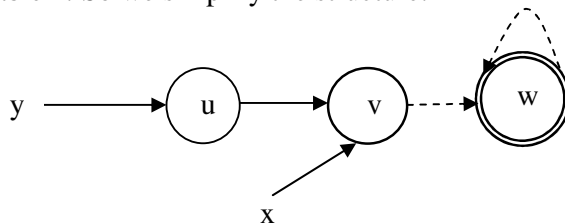
When the loop starts the structure is:



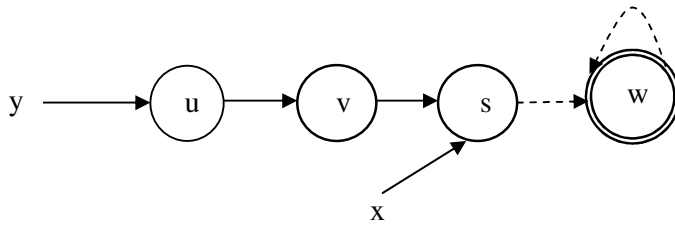
With the help of the focus operation (on $x \rightarrow \text{next}$) shown earlier in this document we perform the promotion of x in the loop once and reach the structure:



We can now increase precision by means of coercion: v is reachable from y so it cannot be shared, but it is pointed from u – it cannot be pointed by any other heap-location. Moreover, v must be a single location (not summary node) since it is pointed from variable x . So we simplify the structure:



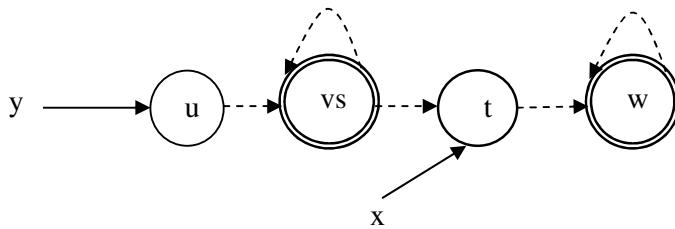
Now we compute the structure after another iteration:



There are a few things to note on how we got this structure:

1. Summary node w from the previous state was split into s and w because of the focus on $x \rightarrow \text{next}$.
2. Location v was not merged with w because $rx(v)=0$ but $rx(w)=1$
3. We apply coercion to use the non-cyclic nature of the list in order to eliminate next relations from w to s and from s to itself
4. v is not a summary node because it is pointed by $u \rightarrow \text{next}$

Now one more iteration:



Here's what happened:

1. Summary node w from the previous state was split into t and w because of the focus on $x \rightarrow \text{next}$
2. Location v was merged with s into the summary node vs because they had identical unary predicates.
3. Location vs not merged with w because $rx(vs)=0$ but $rx(w)=1$

This structure will stay stable if we apply the $x=x \rightarrow \text{next}$ again. We captured a structure with 4 nodes:

1. The node u is the head of the list
2. The node t represent a location somewhere in the middle of the list, from which there is no reachability to the head of the list and to a set of elements noted by vs
3. The node vs represents elements between the head of the list and t (reachable from y but not from x)
4. The summary node w represents all list locations after t (reachable from both x and y).

With just 4 nodes we managed to capture some information (that may be useful/interesting) regarding the heap structure.