Shape analysis via three-valued logical structures

Summary by Ori Lahav. January 2, 2013.

This summary presents an important and interesting example of abstract interpretation for (statically) verifying heap intensive programs (programs that manipulate the heap allocated storage).

1 Logical background

In this section, we provide the necessary definitions and theorems in logic, on which the described method consists. We adapt and simplify all general notions to our purposes.

1.1 Logical structures

A (relational) *signature* is a set of relation symbols, each of which is associated with some arity.

A (*logical) structure* M for a signature σ consists of:

- A non-empty set of elements, called *domain*, and denoted by U_M.
- An interpretation function, denoted by I_M, assigning to each relation symbol p in σ of arity n, a function $I_M[p]: {U_M}^n \to \{0,1\}$.

All signatures that we use below include a binary relation symbol, denoted by "=". This symbol has a special status: all structures should interpret it as the identity relation over U_M, i.e., $I_M[=] = \lambda u_1, u_2. \begin{cases} 0 & u_1 \neq u_2 \\ 1 & u_1 = u_2 \end{cases}$.

1.2 First-order logic with transitive closure

We use a usual first-order formal language (with ¬,∧,∨,∀,∃), augmented with a transitive reflexive closure operator denoted by TC. For a given signature σ, formulas in this language over σ are built as usual using ¬,∧,∨,∀,∃, where TC is used as following:

- If φ is a formula with two free variables a and b, then $[TC_{a,b}\,\varphi](c,d)$ is a formula for every two variables c and d.

To define the truth-value of a given formula (for a given structure), we need first to introduce *assignments.* An *assignment* for a structure M is a function assigning an element from U_M to every variable of the language. Now, the value (in $\{0,1\}$) assigned by a structure M and an assignment α to a formula φ is recursively defined as usual, and denoted by $value(M,\alpha,\varphi)$:

- $value\big(M,\alpha,p(a_1,\dots,a_n)\big) = I_M[p](\alpha(a_1),\dots,\alpha(a_n))$ for every n-ary relation symbol p and variables $a_1,\dots,a_n$
- $value(M,\alpha,\neg\varphi) = 1 - value(M,\alpha,\varphi)$
- $value(M,\alpha,\varphi\wedge\psi) = \sqcap\{value(M,\alpha,\varphi), value(M,\alpha,\psi)\}$
- $value(M,\alpha,\varphi\vee\psi) = \sqcup\{value(M,\alpha,\varphi), value(M,\alpha,\psi)\}$

- $value(M,\alpha,\forall x.\varphi) =$
  $\sqcap\{value(M,\alpha',\varphi)|\alpha' \text{ (possibly) differs from } \alpha \text{ only in its value for } x\}$
- $value(M,\alpha,\exists x.\varphi) =$
  $\sqcup\{value(M,\alpha',\varphi)|\alpha' \text{ (possibly) differs from } \alpha \text{ only in its value for } x\}$

Here, $\sqcap$ is standing for min, and $\sqcup$ is standing for max.

In order to define $value\big(M,\alpha,[TC_{a,b}\varphi](c,d)\big)$ we first define $value\big(M,\alpha,[TC_{a,b}\varphi],u_1,\dots,u_n\big)$ for every $u_1,\dots,u_n \in U_M$ (the intended meaning is that $u_1,\dots,u_n$ is a path using φ):

- $value\big(M,\alpha,[TC_{a,b}\varphi],u_1,\dots,u_n\big) =$
  $\sqcap\{value(M,\alpha',\varphi(a,b))|\alpha'(a) = u_i \text{ and } \alpha'(b) = u_{i+1} \text{ for some } 1 \le \mathrm{i} \le \mathrm{n}-1\}$

And now we can define $value\big(M,\alpha,[TC_{a,b}\varphi](c,d)\big)$ as follows:

- $value\big(M,\alpha,[TC_{a,b}\varphi](c,d)\big) =$
  $\sqcup\left\{value\left(M,\alpha,[TC_{a,b}\varphi],\alpha(c),u_1,\dots,u_n,\alpha(d)\right)\middle|u_1,\dots,u_n \in U_M\right\}$

In other words, $value\big(M,\alpha,[TC_{a,b}\varphi](c,d)\big) = 1$ iff the pair $(\alpha(c),\alpha(d))$ is in the transitive reflexive closure of the relation assigned to p in M. To make this evident, we will use a syntactic sugar and write $p^*(c,d)$ instead of $[TC_{a,b}\varphi](c,d)$. Similarly, we write $p^+(c,d)$ instead of $[TC_{a,b}\varphi](c,d) \wedge \neg(\mathrm{c} = d)$.

Finally, note that $value(M,\alpha,\varphi) = value(M,\alpha',\varphi)$ if α and α' agree on all free variables of φ. In other words, only the values assigned by α to the free variables of φ are material. Thus, we are able to use a simplified notation: If $a_1,\dots,a_n$ are the free variables of a formula φ and $u_1,\dots,u_n \in U_M$, we will denote by $value(M,[a_1 \to u_1,\dots,a_n \to u_n],\varphi)$ the value assigned to φ by M and *any* assignment α satisfying $\alpha(a_i) = u_i$.

1.3 Three-valued structures

In three-valued logics we have three truth-values (instead of two), and we denote them by 0,1 and ½. Intuitively, ½ is standing for "maybe" (or: "unknown"). We shall use an information (partial) order on $\{0,1,½\}$, denoted by $\sqsubseteq$, and defined by the set of pairs $\{(0,0),(0,½),(1,1),(1,½),(½,½)\}$ (namely, ½ is the maximum, and 0,1 are not comparable). $\sqcup$ and $\sqcap$ will denote respectively the join and meet operations for $\sqsubseteq$.

A three-valued structure is defined exactly as a usual structure, except for the fact that the interpretation of the relation symbols is three-valued. Thus, for every n-ary relation symbol p, its interpretation $I_M[p]$ in a three-valued structure M is a function from n-tuples of elements of U_M to $\{0,1,½\}$. Note that we do not require here anything from $I_M[=]$. Given a three-valued structure M and an assignment α (for M), the value assigned to every formula (in the language of first-order logic with transitive closure) is recursively defined using exactly the same definitions used above for usual structures (obviously, $\sqcup$ and $\sqcap$ have a different meaning now).

1.3.1 Bounded three-valued structures

A three-valued structure S for a signature σ is called *bounded* if for every two distinct elements $u_1, u_2 \in U_S$, there exists some unary relation symbol $p \in \sigma$, such that

$$(I_S[p](u_1) = 0 \text{ and } I_S[p](u_2) = 1) \text{ or } (I_S[p](u_1) = 1 \text{ and } I_S[p](u_2) = 0)$$

Note that a bounded three-valued structure for a finite σ has at most $2^{|\sigma|}$ elements in its domain. Hence, there is a finite number (up to isomorphism) of bounded three-valued structures for every finite signature σ.

1.4 Embeddings

An *embedding* of a (two-valued) structure M into a three-valued structure S is a surjective function F from U_M to U_S such that the following holds: For every n -ary relation symbol p and $u_1, \dots, u_n \in U_M$, $I_M[p](u_1, \dots, u_n) \sqsubseteq I_S[p](F(u_1), \dots, F(u_n))$. We write $S \leq M$ if there exists an embedding of M into S. (An embedding of a three-valued structure T into a three-valued structure S is defined similarly).

Note that if we have an embedding F of a structure M into three-valued structure S and an assignment α for M, then the composition $F \circ \alpha$ is an assignment for S. Also, by definition, we have the following properties for every n-ary relation symbol p and assignment α for M: $value\big(M, \alpha, p(x_1, \dots, x_n)\big) \sqsubseteq value\big(S, F \circ \alpha, p(x_1, \dots, x_n)\big)$.

It is possible to show that this connection is preserved by the logical operations in our language and prove the following embedding theorem:

Embedding Theorem: If we have an embedding F of a structure M into three-valued structure S, then for every formula φ and assignment α for M: $value(M, \alpha, \varphi) \sqsubseteq value(S, F \circ \alpha, \varphi)$.

Note that we do not know anything about $value(M, \alpha, \varphi)$ when $value(S, F \circ \alpha, \varphi) = ½$.

1.5 Canonical embedding

We present an example of an embedding of a structure M into a bounded three-valued structure S, called *the canonical embedding*. Let M be a structure for a signature σ. First, define the equivalence relation $\sim_M$ on U_M that is induced by M :

$$u_1 \sim_M u_2 \text{ iff } I_M[p](u_1) = I_M[p](u_1) \text{ for every unary relation symbol } p \text{ in } \sigma$$

Now, define a three-valued structure S (for the same σ) by:

- U_S is the quotient set of U_M under $\sim_M$.
  In other words, U_S consists of all equivalence classes of $\sim_M$.
- For every n-ary relation symbol p in σ:
  $I_S[p](q_1, \dots, q_n) = \sqcup\{I_M[p](u_1, \dots, u_n) \,|\, u_1 \in q_1, \dots, u_n \in q_1\}$.

Clearly, S is bounded. Now, the function F assigning for each $u \in U_M$ its equivalence class (under $\sim_M$) is a surjective function from U_M to U_S. In addition, for every n -ary relation symbol p and $u_1, \dots, u_n \in U_M$, $I_M[p](u_1, \dots, u_n) \sqsubseteq I_S[p](F(u_1), \dots, F(u_n))$. Using the

embedding theorem above, it follows that for every formula φ and assignment α for M: If $value(S, F \circ \alpha, \varphi) \in \{0,1\}$ then $value(M, \alpha, \varphi) = value(S, F \circ \alpha, \varphi)$.

2 Representing the heap using logical structures

At each time point during the run of a program, the shape of the dynamically allocated data structure can be described by a structure M. The domain U_M consists of the nodes of the allocated memory. The signature σ depends on the nature of the program. For simplicity of presentation, we consider programs that manipulate linked lists (but it should be straightforward to handle other types of dynamically allocated data structures). In this case our signature consist of:

1. Unary relation symbols, one for each variable that appears in the program. We will use the names of the variables for these symbols as well. The intended meaning of $x[a]$ is "x points at $\alpha(a)$" (i.e. the value of x is the address of $\alpha(a)$).
2. A Binary relation symbol, denoted by n. The intended meaning of $n(a,b)$ is "$\alpha(b)$ is the next element of $\alpha(a)$ in the list".

For example, the structure defined by $U_M = \{u, v, w\}$, $I_M[head](u) = 1$, $I_M[head](v) = I_M[head](w) = 0, I_M[tail](u) = I_M[tail](v) = 0, I_M[tail](w) = 1$, and $I_M[n](u,v) = I_M[n](v,w) = 1$ (and $I_M[n]$ is zero otherwise) represents a case where we have a linked list with three items, the stack variable $head$ points at the head of the list, and the stack variable $tail$ points at its tail.

2.1 Describing properties of the program

Using the formal language of first-order logic with transitive closure, it is possible to define several interesting properties of dynamically allocated data structures. These may be the goals for verification. Here are few examples:

1. **No garbage.** Suppose that our program has two variables x, y. Then, to have no garbage means that every node is accessible either from x or from y. This can be expressed by the formula $\forall a. \exists b. (x(b) \vee y(b)) \wedge n^*(b,a)$. Clearly, we can have a similar formula if we have more variables.
2. **Acyclic list.** If we want to express the fact that there is no cyclic list, we can use the formula: $\forall a. \neg n^+(a,a)$.
3. **Acyclic list from x.** If we want to express the fact that there is no cyclic list accessible from the variable x, we can use the formula: $\forall a, b. x(a) \wedge n^*(a,b) \rightarrow \neg n^+(b,b)$.

2.2 Instrumentations

In some applications, and also to increase precision (see Section 4.1 below), it is required to add several more relation symbols to the signature, intending to capture different properties of the dynamically allocated memory. We call these additions "instrumentations". We present some instrumentations, that turned out to be useful in several applications:

- Order. We can add a binary relation symbol in_order, where the intended meaning of $in\_order(a,b)$ is that the data stored in node $\alpha(a)$ is less than or equal to the

data in $\alpha(b)$. This (or a similar instrumentation) is clearly needed to describe and analyze correctness of sorting programs.

- Cyclicity. For every variable x, one can have a nullary (0-ary) relation c_x, which expresses the fact that some cycle is accessible from x. Clearly, it can be calculated from the structure we already have (the relations x and n). However, having this as a primitive relation may increase the precision of the method described below.
- Heap sharing. A node is *shared* if two different nodes are pointing at it. To keep this data, we can add a unary relation is_shared. The intended meaning of $is\_shared(a)$ is that $\alpha(a)$ is shared. Again, it can be calculated from the structure we already have (using e.g. the formula $\exists b, c. n(b,a) \wedge n(c,a) \wedge b \neq c$), but having it as a primitive relation may increase the precision of the method described below.
- Reachability. A node u is reachable from a node v if there exists a path (using the next pointers) from v to u. To keep this data, we can add a binary relation $reachable$, and the intended meaning $reachable(a,b)$ is that $\alpha(b)$ is reachable from $\alpha(a)$. A unary relation expressing reachability from a certain variable $reachable_x$ can be also added, and the intended meaning of $reachable_x(a)$ is that $\alpha(a)$ is reachable from x. While both $reachable(a,b)$ and $reachable_x(a)$ can be calculated from the structure we already have (using e.g. the formulas $n^*(a,b)$ and $\exists w. x(w) \wedge n^*(w,a)$), having them as primitives may again increase the precision of the method described below.

2.3 Effect of statements (concrete semantics rules)

Now that we described how to represent the shape of the dynamically allocated data structure using a (logical) structure, it is also possible to interpret the statements of a program as functions from (logical) structures to (logical) structures. Each statement cmd is associated with a function $\mathrm{Sem}^c_{\mathrm{cmd}}$ from structures to structures (the superscript c is standing for "concrete"). These functions, sometimes called *concrete semantics rules*, will be used below in the definition of the semantics of programs. In the next table we provide examples of $\mathrm{Sem}^c_{\mathrm{cmd}}$ for several statements. We denote by M the input structure, and by N the output structure. To have short descriptions of $\mathrm{Sem}^c_{\mathrm{cmd}}$, we only describe the indigents of N which differ from those of M:

Cmd	$\mathrm{Sem}^c_{\mathrm{cmd}}$
$x := \mathbf{NULL}$	$I_N[x] = \lambda u. 0$
$x := \mathbf{malloc}$	$U_N = U_M \uplus \{v\}$ (where v is a new element), $I_N[x] = \lambda u. \begin{cases} 0 & u \neq v \\ 1 & u = v \end{cases}$, and for each other n-ary relation symbol p, $I_N[p](u_1,\ldots,u_n) = 0$ if one of the u_i's is v, and $I_N[p](u_1,\ldots,u_n) = I_M[p](u_1,\ldots,u_n)$ otherwise.
$x := y$	$I_N[x] = \lambda u. value(M, [a \to u], y(a))$
$x := y \to \mathbf{next}$	$I_N[x] = \lambda u. value(M, [b \to u], \exists a. y(a) \wedge n(a,b))$
$x \to \mathbf{next} := y$	$I_N[n] = \lambda u, w. value(M, [a \to u, b \to w], (n(a,b) \wedge \neg x(a)) \vee (x(a) \wedge y(b)))$

Many statements involving the data itself (rather than the shape of the memory) do not affect the structure at all (e.g. a statement like "$x \to data := 5$").

Note that in general, for each statement cmd, $\mathrm{Sem}^c_{\mathrm{cmd}}$ consists of a set of expressions of the form $\lambda u_1, \ldots, u_n. value(M, [a_1 \to u_1, \ldots, a_n \to u_n], \varphi)$ one for each n-ary relation symbol of σ (with the exception of "$x := malloc$", that requires a special treatment).

Also note that when σ include some instrumentations (like those described above), then each of which should be handled as well in the concrete semantics rules. This is usually straightforward. For example, if we have $is\_shared(a)$ in our signature, then the rule for "$x \to next := y$" should also include the following:

$I_N[is_{shared}] = \lambda u.value(M,[a \to u],$
$is\_shared(a) \wedge \exists b,c.\neg(b = c) \wedge n(b,a) \wedge n(c,a) \wedge \neg x(b) \wedge \neg x(c)) \vee \big(t(a) \wedge \exists b.n(b,a) \wedge \neg x(b)\big)\big)$

2.3.1 Example

Consider the structure M defined by:

$$U_M = \{w,v\}, \quad I_M[x](w) = 1 \quad I_M[x](v) = 0$$

$$I_M[n](v,v) = 0 \quad I_M[n](v,w) = 0 \quad I_M[n](w,v) = 1 \quad I_M[n](w,w) = 0$$

We have: $\lambda u.value\big(M,[b \to u], \exists a.\, x(a) \wedge n(a,b)\big) = \begin{cases} 0 & u = w \\ 1 & u = v \end{cases}$.

Thus, after applying the rule of the statement "$x := x \to next$" we obtain the following structure $N = \mathrm{Sem}^c_{x:=x\to next}(M)$:

$$U_N = \{u,v\}, \quad I_N[x](u) = 0 \quad I_N[x](v) = 1$$

$$I_N[n](v,v) = 0 \quad I_N[n](v,u) = 0 \quad I_N[n](u,v) = 1 \quad I_N[n](u,u) = 0$$

3 Applying abstract interpretation

In this section, we describe how to apply abstract interpretation to verify correctness of heap intensive programs during compilation time. The tools of the previous sections will become handy.

First, given a program $Prog$, one fixes a signature σ which will be used to describe the shape of the dynamically allocated data structure during runs of $Prog$. For programs that manipulate linked lists it is possible to use the signature suggested above (possibly, with some instrumentations). Note that the number of unary relation symbols in σ depends on the number of variables used in $Prog$.

Now, as usual in abstract interpretation, we have a concrete semantics of $Prog$, and a sound (but not complete) abstract semantics of $Prog$. More precisely, we have a concrete domain $\mathcal{C}$, an abstract domain $\mathcal{A}$, and a concretization function $\gamma: \mathcal{A} \to \mathcal{C}$. Then, the concrete semantics Sem^c of $Prog$ is a function from $Prog \to \mathcal{C}$ to $Prog \to \mathcal{C}$ (here we identify $Prog$ with its set of program points). Similarly, the abstract semantics Sem^a of $Prog$ is a function from $Prog \to A$ to $Prog \to A$. The method itself would calculate the least fixed point of Sem^a (possibly using the "chaotic iteration" method), and use it to (statically) verify the required properties. It will be computable since the abstract domain will be finite. The results will be correct, since we will have a sound abstraction, namely: for every mapping $G \in Prog \to \mathcal{A}$ and $q \in Prog$, we should have $\mathrm{Sem}^c\big(\lambda q \in Prog.\gamma\big(G(q)\big)\big)(q) \subseteq \gamma\big(\mathrm{Sem}^a(G)(q)\big)$.

3.1 Concrete and abstract domains

The **concrete domain** consists of all sets of structures over σ. We denote this domain by $\mathcal{C}$. We use $\subseteq$ (set inclusion) as a partial order on $\mathcal{C}$.

$\mathrm{Sem}^{\mathrm{c}}$, the concrete semantics of $Prog$, is defined as follows: given a mapping $\mathrm{F} \in Prog \to \mathcal{C}$, and a program point p_1, $\mathrm{Sem}^{\mathrm{c}}(\mathrm{F})(p_1)$ is the set of structures obtained by applying the concrete semantics rules (see Section 2.3) on every element in $\mathrm{F}(p_1)$.

The **abstract domain** consists of all sets of bounded *three-valued* structures over σ. We denote this domain by $\mathcal{A}$. This is a finite number, and thus the abstract domain is finite. As a partial order on $\mathcal{A}$, we use $\preccurlyeq$, defined by: $A_1 \preccurlyeq A_2$ iff for every $S_1 \in A_1$ there exists some $S_2 \in A_2$ such that $S_1 \leq S_2$.

$\mathrm{Sem}^{\mathrm{a}}$, the abstract semantics of $Prog$, is defined similarly to $\mathrm{Sem}^{\mathrm{c}}$ above, but instead of using the concrete semantics rules, we use *abstract* semantics rules, called also: *transformers*. For each statement cmd, we have a transformer $\mathrm{Sem}^{\mathrm{a}}_{\mathrm{cmd}}$, which is a mapping from bounded three-valued structures to bounded three-valued structures. These transformers are defined exactly like the concrete semantic rules, but here the values of the formulas are calculated in a three-valued structure. The statement "$x := malloc$" again requires a special treatment, a new element should be added to the domain and the interpretation function should be changed as in the corresponding concrete semantic rule.

3.1.1 Example

Consider the three-valued structure S defined by:

$$U_S = \{u, v\}, \quad I_S[x](u) = 1 \quad I_S[x](v) = 0$$

$$I_S[n](v,v) = ½ \quad I_S[n](v,u) = 0 \quad I_S[n](u,v) = ½ \quad I_S[n](u,u) = 0$$

The transformer of the statement "$x := x \to next$" updates $I_S[x]$, and sets: $I_T[x] = \lambda u.\, value\,(S, [b \to u], \exists a.\, x(a) \wedge n(a,b))$ (where $T = \mathrm{Sem}^{\mathrm{a}}_{x:=x\to next}(S)$ is the output three-valued structure). Now:

- $value\big(S, [b \to u], \exists a.\, x(a) \wedge n(a,b)\big) = value\big(S, [a \to u, b \to u], x(a) \wedge n(a,b)\big) \sqcup value\big(S, [a \to v, b \to u], x(a) \wedge n(a,b)\big) = 0 \sqcup 0 = 0$
- $value\big(S, [b \to v], \exists a.\, x(a) \wedge n(a,b)\big) = value\big(S, [a \to u, b \to v], x(a) \wedge n(a,b)\big) \sqcup value\big(S, [a \to v, b \to v], x(a) \wedge n(a,b)\big) = ½ \sqcup 0 = ½$

Thus, we obtain the three-valued structure T degined by:

$$U_T = \{u, v\}, \quad I_T[x](u) = 0 \quad I_T[x](v) = ½$$

$$I_T[n](v,v) = 0 \quad I_T[n](v,u) = 0 \quad I_T[n](u,v) = ½ \quad I_T[n](u,u) = ½$$

3.1.2 Concretization and soundness

The concretization function $\gamma: \mathcal{A} \to \mathcal{C}$ is defined by: $\gamma(A) = \{M | \exists S \in A.\, S \leq M\}$. It is easy to see that γ is monotone, i.e. if $A_1 \preccurlyeq A_2$ then $\gamma(A_1) \subseteq \gamma(A_2)$. The corresponding abstraction function is $\alpha: \mathcal{C} \to \mathcal{A}$ is defined by: $\alpha(C) = \Cap \{A | \forall M \in C.\, \exists S \in A.\, S \leq M\}$, where $\Cap$ is the meet operation with respect to $\preccurlyeq$.

It remains to prove the soundness of the proposed abstraction. As usual in abstract interpretation, the soundness of the full abstract semantics follows from the local soundness of the transformers. Thus, we should prove that for every statement cmd and every $A \in \mathcal{A}$, we have:

$$\{\mathrm{Sem}^{\mathrm{c}}_{\mathrm{cmd}}(M) | M \in \gamma(A)\} \subseteq \gamma(\{\mathrm{Sem}^{\mathrm{a}}_{\mathrm{cmd}}(\mathrm{S}) | S \in A\})$$

Using the definition of γ, this is equivalent to:

> For every structure M: if there exists some $S \in A$ such that $S \leq M$ then $\mathrm{Sem}^{\mathrm{a}}_{\mathrm{cmd}}(\mathrm{S}) \leq \mathrm{Sem}^{\mathrm{c}}_{\mathrm{cmd}}(M)$ for some $S \in A$.

Thus, obviously it suffices to show that $S \leq M$ implies that $\mathrm{Sem}^{\mathrm{a}}_{\mathrm{cmd}}(\mathrm{S}) \leq \mathrm{Sem}^{\mathrm{c}}_{\mathrm{cmd}}(M)$. This would be a simple corollary of the embedding theorem:

Suppose that $S \leq M$. Thus there exists a surjective function F from U_M to U_S such that for every n -ary relation symbol p and $u_1, \ldots, u_n \in U_M$, we have $I_M[p](u_1, \ldots, u_n) \sqsubseteq I_S[p](F(u_1), \ldots, F(u_n))$. Note that $\mathrm{Sem}^{\mathrm{c}}_{\mathrm{cmd}}(M)$ is a structure N obtained by setting for every n -ary relation symbol p, $I_N[p] = \lambda u_1, \ldots, u_n. value(M, [a_1 \to u_1, \ldots, a_n \to u_n], \varphi)$, where $a_1, .., a_n$ are some variables and φ is some formula. Similarly, according to the definition of the transformers, $\mathrm{Sem}^{\mathrm{a}}_{\mathrm{cmd}}(\mathrm{S})$ is a three-valued structure T is obtained by setting for every n -ary relation symbol p $I_T[p] = \lambda u_1, \ldots, u_n. value(S, [a_1 \to u_1, \ldots, a_n \to u_n], \varphi)$, using the same variables $a_1, .., a_n$ and formula φ that were used for defining $I_N[p]$. Now, the embedding theorem implies that we always have: $value(M, \alpha, \varphi) \sqsubseteq value(S, F \circ \alpha, \varphi)$. In other words, for every n -ary relation symbol p we have $I_N[p](u_1, \ldots, u_n) \sqsubseteq I_T[p](F(u_1), \ldots, F(u_n))$. Also, since $U_N = U_M$ and $U_S = U_T$, F is also a surjective function F from U_N to U_T. It follows that the same function F is an embedding of $N = \mathrm{Sem}^{\mathrm{c}}_{\mathrm{cmd}}(M)$ into $T = \mathrm{Sem}^{\mathrm{a}}_{\mathrm{cmd}}(\mathrm{S})$. Therefore, $\mathrm{Sem}^{\mathrm{a}}_{\mathrm{cmd}}(\mathrm{S}) \leq \mathrm{Sem}^{\mathrm{c}}_{\mathrm{cmd}}(M)$. (Note that the case of statement "$x := malloc$" requires a special treatment in this proof. We skip this detail here).

4 Increasing precision

The method described above is sound, but for many applications it is not precise enough. In this section we describe several possible heuristic additions, that retain soundness and may increase the precision.

4.1 Instrumentation

As we hinted before, it is possible to increase the precision of the method by using some instrumentations, i.e. by adding some more relation symbols to the signature of the structures that represent the shape of the dynamically allocated data structure. For example, consider a linked list of three elements, where the variable x points at the head of the list. Without instrumentations such a list is represented by the following structure M:

$$U_M = \{u, v, w\}, \quad I_M[x](u) = 1 \quad I_M[x](v) = 0 \quad I_M[x](w) = 0$$

$$I_M[n](u, v) = I_M[n](v, w) = 1 \quad \text{and all other values of } I_M[n] \text{ are zero}$$

The abstraction of M (formally, of $\{M\}$) includes, in particular, the three-valued structure S obtained from M using the canonical embedding:

$$U_S = \{\{u\},\{v,w\}\}, \quad I_S[x](\{u\}) = 1 \quad I_S[x](\{v,w\}) = 0$$

$$I_S[n](\{u\},\{v,w\}) = I_S[n](\{v,w\},\{v,w\}) = ½ \qquad I_S[n](\{v,w\},\{u\}) = I_S[n](\{u\},\{u\}) = 0$$

Now, if we now try to verify the fact that there is no garbage using the formula $\forall a.\exists b.x(b) \wedge n^*(b,a)$ we would have a "false-alarm" since it evaluates to ½ in S, while in M it evaluates to 1. However, let us add an instrumentation in the form of a unary relation symbol $reachable_x(a)$ with the intended meaning that $\alpha(a)$ is reachable from x (α here is an assignment). M would now additionally include:

$$I_M[reachable_x](u) = 1 \quad I_M[reachable_x](v) = 1 \quad I_M[reachable_x](w) = 1$$

And in S we have:

$$I_S[reachable_x](\{u\}) = 1 \quad I_S[reachable_x](\{v,w\}) = 1$$

In fact, in any three-valued structure S such that $S \leq M$, we will have $I_S[reachable_x](e) = 1$ for every $e \in U_S$. Consequently, we are now able to verify that there is no garbage, since the value assigned by every three-valued strucre in the abstraction of $\{M\}$ to the formula $\forall a.reachable_x(a)$ is 1.

To conclude, by explicitly storing in the structures values of certain additional formulas, we can make finer distinctions and improve the precision of the method.

4.2 Semantic reductions

A *semantic reduction* is an operation on an abstract domain element that returns another abstract domain element which is more precise, but has the same concretization. In other words, it is a function $O: \mathcal{A} \to \mathcal{A}$, such that $O(A) \preccurlyeq A$ but $\gamma(O(A)) = \gamma(A)$ for every $A \in \mathcal{A}$. The application of such a function before and/or after applying the transformer is obviously sound, and may increase precision. We describe two possible such reductions.

4.2.1 Focus

Some transformers operate "better" when the value of a certain formula φ is definite (i.e. either 0 or 1). In the calculations of these transformers on some three-valued structure S, it is possible to apply a semantic reduction called *foucs*, which will first replace S by a set of three-valued structures, each of which has an embedding into S and a definite value for φ. We present a simple example (and omit many other details):

Consider the three-valued structure S:

$$U_S = \{u,v\}, \qquad I_S[y](u) = 1 \quad I_S[y](v) = 0 \quad I_S[x](u) = 1 \quad I_S[x](v) = 0$$

$$I_S[n](v,v) = 0 \quad I_S[n](v,u) = 0 \quad I_S[n](u,v) = ½ \quad I_S[n](u,u) = ½$$

Now, suppose that the transformer of "$x := x \to next$". This transformer updates $I_S[x]$ using the formula $\lambda u.value(S,[b \to u],\exists a.x(a) \wedge n(a,b))$. Since $value(S,[b \to$

$v], \exists a.\, x(a) \wedge n(a,b)\big) = value\big(S, [a \to u, b \to v], x(a) \wedge n(a,b)\big) \sqcup value\big(S, [a \to v, b \to v], x(a) \wedge n(a,b)\big) = ½ \sqcup 0 = ½$, we obtain the following structure T:

$$U_T = \{u,v\}, \qquad I_T[y](u) = 1 \quad I_T[y](v) = 0 \quad I_T[x](u) = 0 \quad I_T[x](v) = ½$$

$$I_T[n](v,v) = 0 \quad I_T[n](v,u) = 0 \quad I_T[n](u,v) = ½ \quad I_T[n](u,u) = ½$$

Thus, we do not capture the fact that after this opertation, $y \to next$ points to the same location as x. To have a more precise transformer, we can apply *focus*, and make $value\big(S, [b \to v], \exists a.\, x(a) \wedge n(a,b)\big)$ to have a definite value. This means that we first replace S with a set of three-valued structures in which this value is definite, and each of them has an embedding into S. In this case, it is possible to take the following structures:

1. $U_{S_1} = \{u,v\},\ I_{S_1}[y](u) = 1 \quad I_{S_1}[y](v) = 0 \quad I_{S_1}[x](u) = 1 \quad I_{S_1}[x](v) = 0$
   $I_{S_1}[n](v,v) = 0 \quad I_{S_1}[n](v,u) = 0 \quad I_{S_1}[n](u,v) = 0 \quad I_{S_1}[n](u,u) = ½$
2. $U_{S_2} = \{u,v\},\ I_{S_2}[y](u) = 1 \quad I_{S_2}[y](v) = 0 \quad I_{S_2}[x](u) = 1 \quad I_{S_2}[x](v) = 0$
   $I_{S_2}[n](v,v) = 0 \quad I_{S_2}[n](v,u) = 0 \quad I_{S_2}[n](u,v) = 1 \quad I_{S_2}[n](u,u) = ½$

Now, after applying the transformer, we obtain the set:

1. $U_{T_1} = \{u,v\},\ I_{T_1}[y](u) = 1 \quad I_{T_1}[y](v) = 0 \quad I_{T_1}[x](u) = 0 \quad I_{T_1}[x](v) = 0$
   $I_{T_1}[n](v,v) = 0 \quad I_{T_1}[n](v,u) = 0 \quad I_{T_1}[n](u,v) = 0 \quad I_{T_1}[n](u,u) = ½$
2. $U_{T_2} = \{u,v\},\ I_{T_2}[y](u) = 1 \quad I_{T_2}[y](v) = 0 \quad I_{T_2}[x](u) = 0 \quad I_{T_2}[x](v) = 1$
   $I_{T_2}[n](v,v) = 0 \quad I_{T_1}[n](v,u) = 0 \quad I_{T_2}[n](u,v) = 1 \quad I_{T_2}[n](u,u) = ½$

In each of these structures, we can verify that $y \to next$ points to the same location as x.

4.2.2 Coercion

Coercion is another sort of semantic reduction that can be also applied before and/or after applying the transformer (it is also possible to have it together with focus). Here, the idea is that some global invariant of programs can be exploited. By using this information (which usually depends on the programming language), in many cases we can increase the precision. For example, usually one program variable cannot point at more than one location. In other words, the formula $\forall a,b.\, x(a) \wedge x(b) \to a = b$ should always evaluate to 1. By applying a constraint solver it is possible to replace three-valued structures in which this formula is evaluated to ½ by sets of structures (that have an embedding into the original structures) in which this formula is evaluated to 1. Soundness is retained under the assumption that this global invariant always holds. We omit the details here.

4.3 Implementation and further reading

The method described above is implemented in a system called TVLA (three-valued logic analysis), available at: http://www.cs.tau.ac.il/~tvla/ . Many more details, examples, and illustrations are available at:

1. http://www.cs.tau.ac.il/~msagiv/courses/pa12-13/shapeI.ppt
2. Sagiv, Mooly, Thomas Reps, and Reinhard Wilhelm. "Parametric shape analysis via 3-valued logic." ACM Transactions on Programming Languages and Systems (TOPLAS) 24.3 (2002): 217-298.