Program Analysis **December 27, 2015**

Lecture 10: Shape Analysis

Lecturer: Mooly Sagiv *Scribe: Gal Dudovitch, Daniela Rabkin*

10.1 Shape Analysis - Reminder

The process of shape analysis is used to statically determine properties about a program's dynamically allocated memory. It may be used to generate warnings about unwanted states that may arise during the execution of a program, or consequently verify and prove that they will not exist.

For example, the following questions could be answered by shape analysis:

- Does a variable p point to a shared memory?
- Does a variable p point to an allocated element every time p is dereferenced?
- Does a variable p point to an acyclic list?
- Does a variable p point to a doubly-linked list?
- Can a procedure introduce a memory-leak?

As one can see, answering some of these questions may allow us to enhance the safety and performance of our program. For example, if the answer for the first question is "no", we could skip the use of synchronization elements. We could also use the properties we discover about the memory to assert a block of memory is eventually freed (and only once). This will also provide us with helpful information assisting our program's garbage collection (if we use such a mechanism). Shape analysis is a very powerful tool, however, running the analysis on large programs can take a significant amount of time, and therefore it is not widely used.

10.2 Logical Structures (Labeled Graphs)

The analysis defines a set of relation symbols that are used to describe the variables' properties:

- Nullary relation symbols - $p_0() \rightarrow \{0, 1\}$
- Unary relation symbols - $p_1(v) \rightarrow \{0, 1\}$
- Binary relation symbols - $p_2(u, v) \rightarrow \{0, 1\}$

In addition, we use first order logic with transitive closure (FO^{TC} over TC, $\forall, \exists, \neg, \vee, \wedge$) to describe the invariants we must check during the analysis.

The analysis only stores tables containing a set of individuals (nodes) U, and the properties described above.

10.3 Representing Stores (Memory States) as Logical Structures

The logical structures described above are used throughout the analysis to represent the memory states of the program and its variables. This is usually done as follows:

- Memory locations are the set of individuals (nodes) U.
- Program variables are described by unary relations (e.g. x(v)=1 means variable x points to the individual v)
- Fields are described by binary relations (e.g. n(u1,u2)=1 means that the next field of u1 points to u2).

Theorem 10.1 *Represantation as Graph and Tables*

$U = \{u_1,u_2,u_3,u_4,u_5\}$ $x = \{u_1\}$, $p = \{u_3\}$ $n = \{< u_1, u_2 >, < u_2, u_3 >, < u_3, u_4 >, < u_4, u_5 >\}$

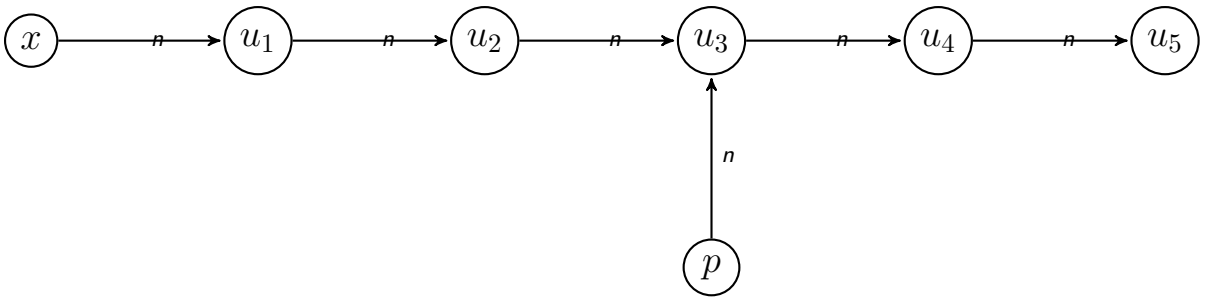
Table 10.1: Unary properties - Variables

	x	p
u1	**1**	0
u2	0	0
u3	0	**1**
u4	0	0
u5	0	0

Table 10.2: Binary properties - Fields

n	u1	u2	u3	u4	u5
u1	0	**1**	0	0	0
u2	0	0	**1**	0	0
u3	0	0	0	**1**	0
u4	0	0	0	0	**1**
u5	0	0	0	0	0

Graph Representation:



10.4 Concrete Interpretation Rules

As seen above, throughout the analysis we store a set of tables representing the relations of the memory locations at each state. When dynamically going over the program's execution, the concrete state of the

program may change from one executed statement to another (i.e. the values in the tables may change at each state). The following table shows several examples of how a line of code may change the value of a given unary or binary relation (note that a tagged property means the new value, e.g. x'(v) is the updated value of x after running the current line of code, while x(v) is its value before running the current line of code):

Statement	Update formula	Explanation
x =NULL	x'(v)= 0	For every node v, the property x of v will be false (i.e. 0).
x= malloc()	x'(v) = IsNew(v)	IsNew is a TVLA operation. It would return 1 if memory was allocated for node v, otherwise 0.
x=y	x'(v)= y(v)	For every node v, the new value of x of v is the same as the value of y of v.
x=y →next	x'(v)= ∃w: y(w) ∧ n(w, v)	For every node v, x of v is true (i.e. 1) iff there exists a node w pointed to by y, and the n-field of w is v.
x →next=y	n'(v, w) = (x(v) ? y(w) : n(v, w))	This line changes the value of the binary property defined by n: For every pair (v, w), if v is not pointed to by x then it remains unchanged (i.e. with the value of n(v, w)), however if v is pointed to by x, then the value n(v, w) becomes the same as y(w) (i.e. true iff w is pointed to by y).

Figure 10.1: Concrete Interpretation Rules.

10.5 Abstract Interpretation

The former example represents a concrete state updates. We would now like to use abstract interpretation to perform the analysis. As always, the transformation from concrete to abstract states may cause a loss of information. While in the concrete state every predicate could only be true or false, in the abstract interpretation, due to the loss of information, we might have a third state which would represent "don't-know". We therefore use Kleene's 3-valued logic for extracting information from the abstract value:

Table 10.3: Kleene's 3-Valued Logic - And Operation

AND	True (1)	Unknown ($\frac{1}{2}$)	False (0)
True (1)	1	$\frac{1}{2}$	0
Unknown ($\frac{1}{2}$)	$\frac{1}{2}$	$\frac{1}{2}$	0
False (0)	0	0	0

As mentioned above, a value of $\frac{1}{2}$ simply means we don't know whether the predicate should be evaluated to true or false. The logic is actually a join semi-lattice where $\frac{1}{2}$ functions as top, i.e. $0 \sqcup 1 = \frac{1}{2}$.

The canonical abstraction function (β) divides the nodes of the program into classes, based on the values of their unary relations. i.e. every two or more elements whose unary predicates are evaluated to the same

Table 10.4: Kleene's 3-Valued Logic - Or Operation

AND	True (1)	Unknown ($\frac{1}{2}$)	False (0)
True (1)	1	1	1
Unknown ($\frac{1}{2}$)	1	$\frac{1}{2}$	$\frac{1}{2}$
False (0)	1	$\frac{1}{2}$	0

values fall into the same class, and are represented in the graph by one summary node (the node is a summary node if it may represent more than one concrete value).

The relations between the abstract elements are evaluated as follows:
$p^S(u'_1, ..., u'_k) = \bigsqcup\{p^B(u_1, ..., u_k) | f(u_1) = u'_1, ..., f(u_k) = u'_k\}$

Remember that we are using a 3-valued logic, so the resulted value may be in 0,1,$\frac{1}{2}$. Also note that if A is the number of unary predicates, then we may have as many as 2^A abstract classes. This number could of course be very large; however, in practice, if we run the analysis on a single procedure, this number will usually be reasonably small.

Theorem 10.2 *Abstract transformation The following shows the transformation from a concrete state representing a linked list with 4 elements, to an abstract state representing (among other things) the same list:*

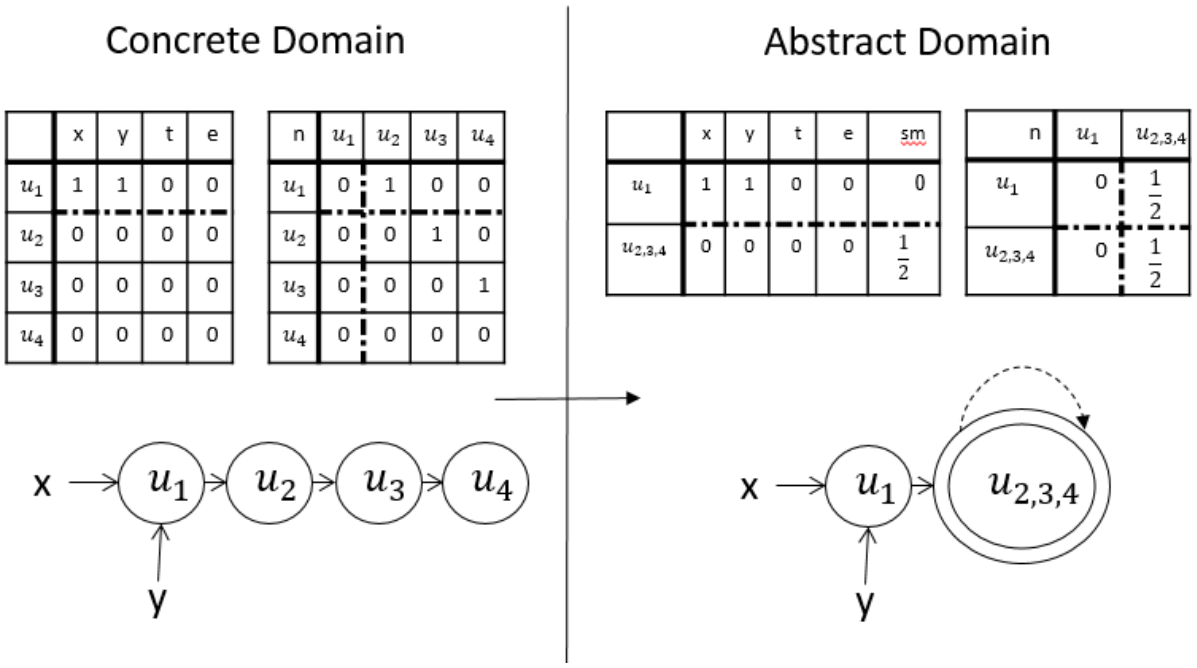


Figure 10.2: Concrete to Abstract transformation.

As we can see, the concrete state has a list with 4 elements, whose head is pointed to by both x and y. In the abstract state, we see that the canonical abstraction created two classes: node u_1 is the member of the class whose elements are pointed to by both x and y, and nodes u_2, u_3, u_4 are members of the class whose elements are not pointed to by any variable (i.e. all their unary predicates evaluate to 0). Note that by looking at the unary predicates' table in the concrete state, it is easy to see that theoretically speaking, if we have 4 variables

(unary predicates), we could have as many as 16 (2^4) different classes. Also, we can see that the relations between the nodes are evaluated as follows, using the join operation:

- $n(u_1, u_1) = 0$, *because in the concrete case* $n(u_1, u_1) = 0$.
- $n(u_1, u_{2,3,4}) = \frac{1}{2}$, *because:* $n(u_1, u_{2,3,4}) = n(u_1, u_2) \sqcup n(u_1, u_3) \sqcup n(u_1, u_4) = 1 \sqcup 0 \sqcup 0 = \frac{1}{2}$.
- $n(u_{2,3,4}, u_1) = n(u_2, u_1) \sqcup n(u_3, u_1) \sqcup n(u_4, u_1) = 0 \sqcup 0 \sqcup 0 = 0$.
- $n(u_{2,3,4}, u_{2,3,4}) = n(u_2, u_2) \sqcup n(u_2, u_3) \sqcup n(u_2, u_4) \sqcup n(u_3, u_2) \sqcup n(u_3, u_3) \sqcup n(u_3, u_4) \sqcup n(u_4, u_2) \sqcup n(u_4, u_3) \sqcup n(u_4, u_4) = 0 \sqcup 1 \sqcup 0 \sqcup 0 \sqcup 0 \sqcup 1 \sqcup 0 \sqcup 0 \sqcup 0 = \frac{1}{2}$.

Also note that the abstract representation contains a new column "sm", which stands for "summary". This column specifies whether the given class is a summary node (i.e. may represent 2 or more concrete nodes) or not (i.e., represents just one concrete node). For technical reasons, this column may contain only 0 or $\frac{1}{2}$ *(where* $\frac{1}{2}$ *means it is a summary node).*

As you may notice, the abstract interpretation is of course potentially less precise than the concrete one. There are other concrete lists that may be represented by the same abstract list shown above. However, we can conservatively test invariants we wish to verify on the abstract representation and if the verification succeeds, we are guaranteed that the same property would hold on the concrete state as well (Soundness). We may, however, get warnings which would not be true.

The abstract interpretation can help us finish the iteration where the concrete interpretation may continue indefinitely: consider the list creation example given in class:

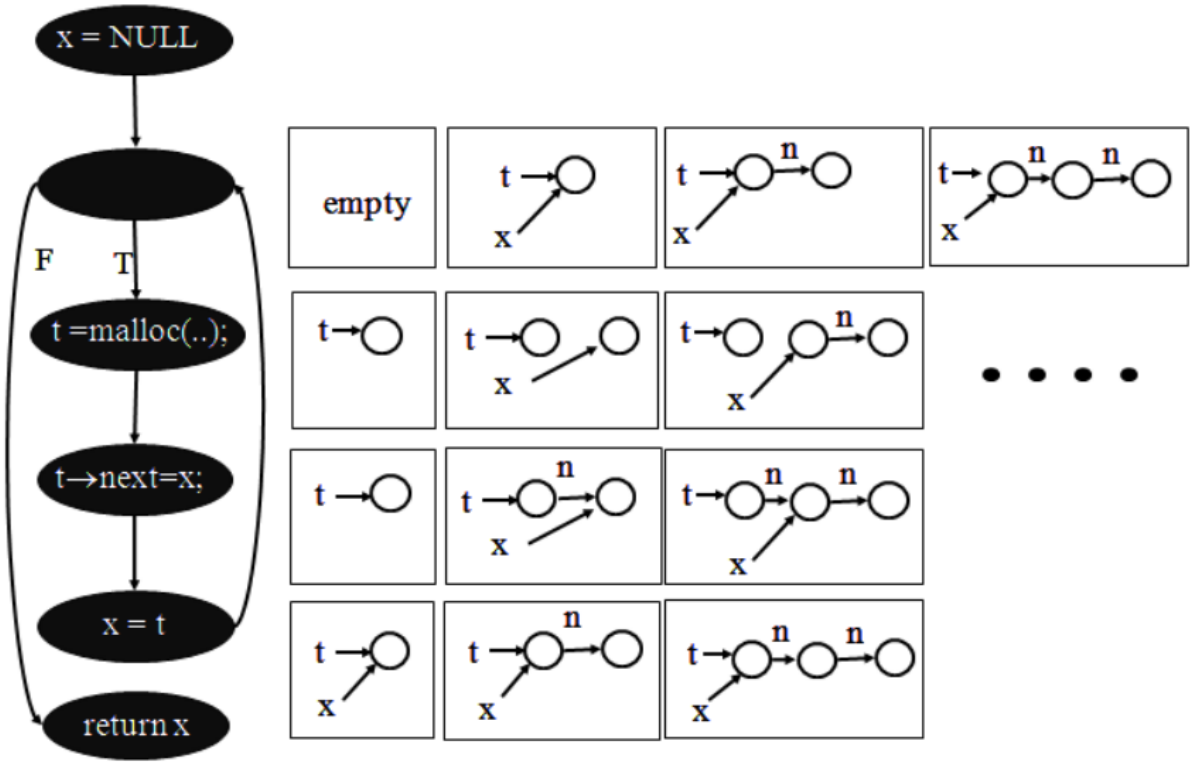


Figure 10.3: List Creation Example, Concrete Representation.

We would like to verify that:

- *The code given in the example does not leak memory.*
- *The list created contains no cycles.*

This figure represents the concrete states for the code. We can see that we would have to continue indefinitely as we never get two subsequent states that are equal to one another for the same configuration node. However, if we used the abstract interpretation, we would get the following:

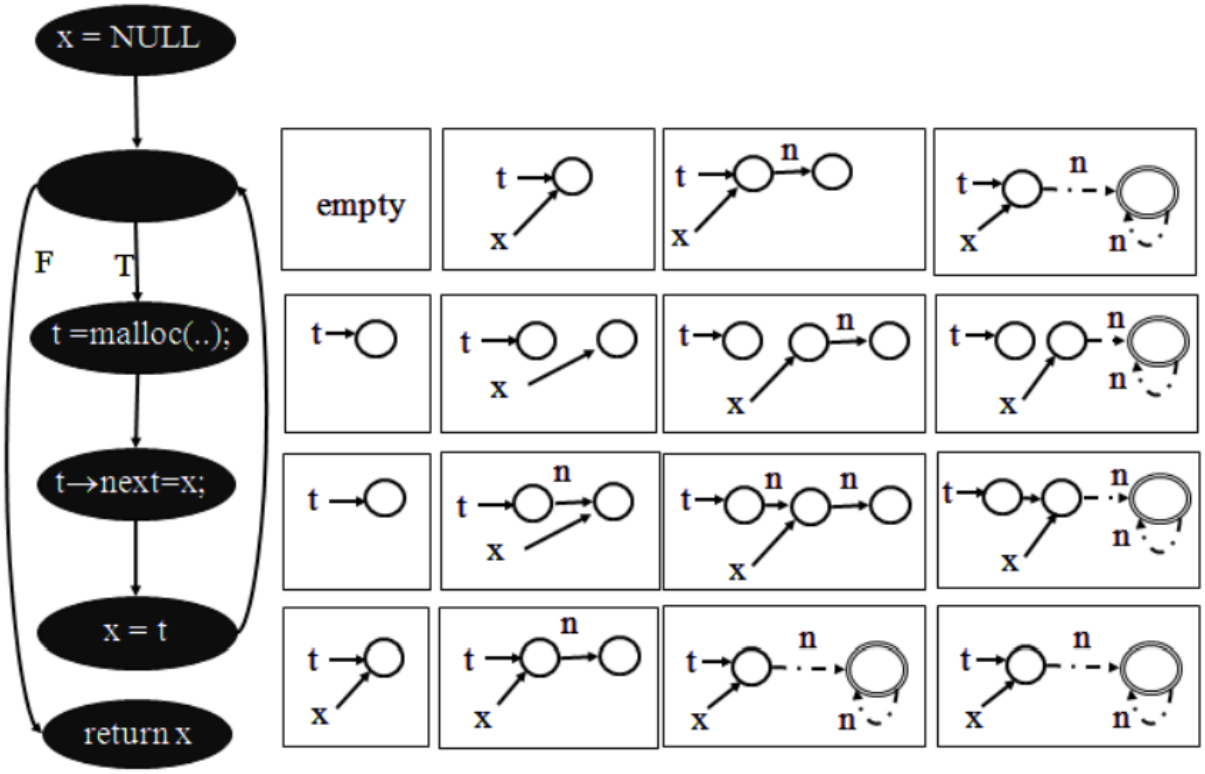


Figure 10.4: List Creation Example, Abstract Representation.

As we can see, the last two states for the last configuration node ($x = t$) are the same, and therefore we stop the iteration. Make note of the summary node, representing two or more concrete nodes (i.e. the list may be 3 or more elements long). Note that given the final state above, we would have to generate warning regarding both questions we have asked (as we may have memory leaks, because the nodes of the summary node may not be pointed to at all, and we may have cycles, as nodes within the summary node may create a cycle with one another). We will see how to resolve these issues shortly.

10.6 Global Invariants

We may define other properties which may be interesting for our analysis. Such properties are represented as unary (or nullary) predicates, and can be defined using first order logic. Let us consider a few examples:

10.6.1 Cyclicity Relation (Nullary)

This relation is intended to check whether there exist cycles in the list, and it is defined as follows: $c[x]() = \exists v_1, v_2 : x(v_1) \wedge n^*(v_1, v_2) \wedge n^+(v_2, v_2)$ *This property checks whether at any given point we have some nodes* v_1 *and* v_2 *(which may actually be the same node as well) such that* v_1 *is pointed to by* x *(i.e. it is the head of the list), there is a path of some length from* v_1 *to* v_2*, and there is a path of length at least 1 from* v_2 *to itself. As long as this property is false, we can guarantee that there are no cycles in the list. For a list with no cycles, the following are the concrete representation and its corresponding abstract representation:*

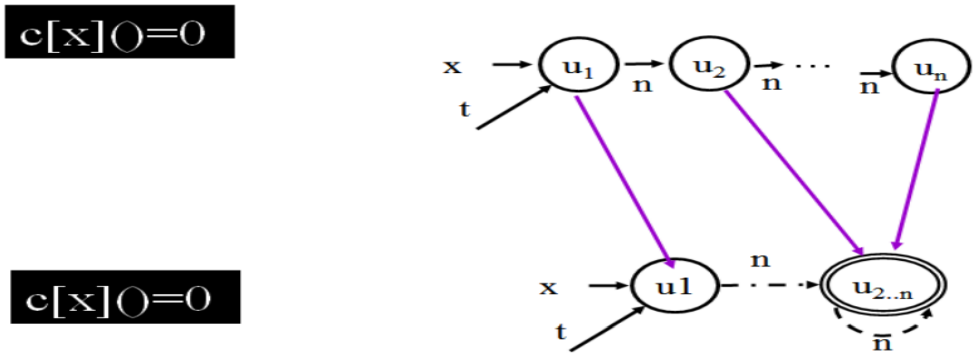


Figure 10.5: Cyclicity Relation with no cycles.

And for a list that does contain a cycle:

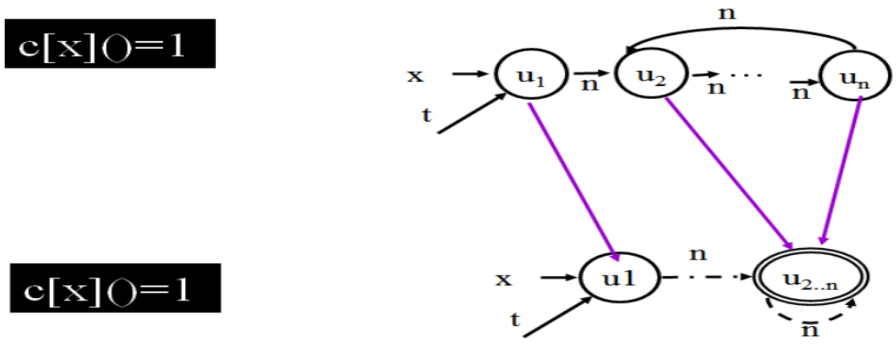


Figure 10.6: Cyclicity Relation with cycles.

Although the graph representation looks the same, we keep track of the property we defined and thus we can tell whether there is a cycle or not. We will later see how the values of properties are kept and updated.

10.6.2 Heap Sharing Relation (Unary)

Another property we may define is heap sharing. This property is unary, and checks for each node if there are (at least) two different heap objects that point to it. It is defined as follows:

$is(v) = \exists v_1, v_2 : n(v_1, v) \wedge n(v_2, v) \wedge v1 \neq v2$

In other words – a node v is heap-shared if it has two nodes v_1 and v_2 pointing to it, and these nodes v_1 and v_2 are not actually the same node. Note that this local property can help us determine whether a list has cycles or not (even though its definition seemingly has nothing to do with cycles):

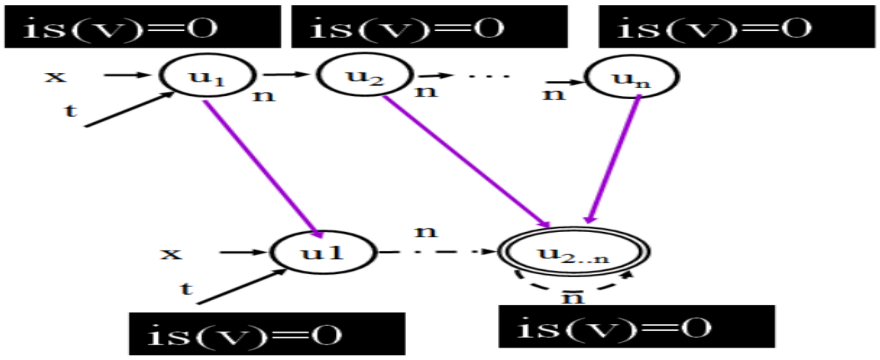


Figure 10.7: Heap Sharing Relation with no cycles.

As we can see, each node v has its own is(v) value. In the case above, for the concrete representation, they are all 0 (note that although u_1 is pointed to from both x and t – these are not heap variables, therefore $is(u_1) = 0$ and not 1). When transforming into the abstract representation, as before, we receive two classes: u_1 and $u_{2...n}$. Each of these nodes receives a value of is(v)=0 as well (as it is simply the join of the corresponding unary properties in the concrete representation).

However, if our list did contain a cycle, we would get:

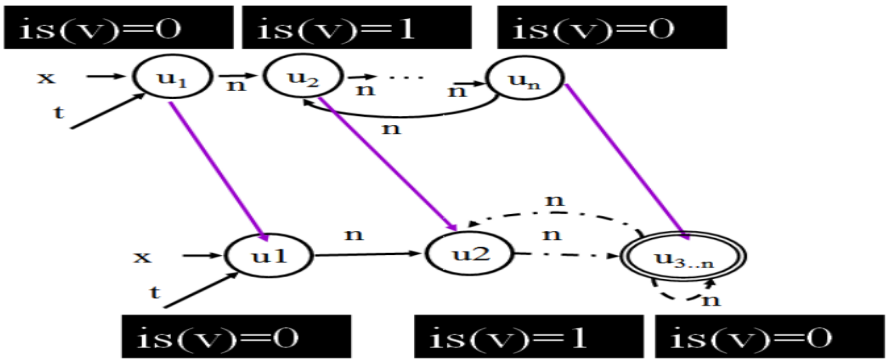


Figure 10.8: Heap Sharing Relation with Cycles.

As we can see, node u_2 in the concrete representation has a value of is(v)=1. Therefore, when transforming into the abstract representation, we now obtain three classes: class u_1 whose elements have x(v)=1, t(v)=1, is(v)=0; class u_2 whose elements have x(v)=0, t(v)=0, is(v)=1; and class $u_{3...n}$ whose elements have x(v)=0, t(v)=0, is(v)=0. In the previous example we did not have a cycle, therefore we obtained only two classes. Now, node u_2 has the property is(v)=1 (unlike the nodes $u_3, u_4, ..., u_n$ which have is(v)=0), thus we receive a third class. As a result, by looking at the abstract representation alone, we know that node u_2 is shared among two heap variables – one of which is u_1 and the other is one of $\{u_3, u_4, ..., u_n\}$ (which are represented

by just one summary node). Therefore, we can conclude that this list potentially has a cycle, and the analysis would produce a warning. Note that this property can completely separate between lists that have cycles and lists that don't (so in the previous example of a list with no cycles, our analysis would know not to produce such a warning).

10.6.3 Reachability Relation (Binary)

Lastly, let us consider an example for a binary property – reachability. This property defines for any two nodes v_1 and v_2 whether there is a path of some length from v_1 to v_2 or not. It is defined as follows:

$t[n](v_1, v_2) = n^*(v_1, v_2)$

The transformation from concrete to abstract representation looks as follows:

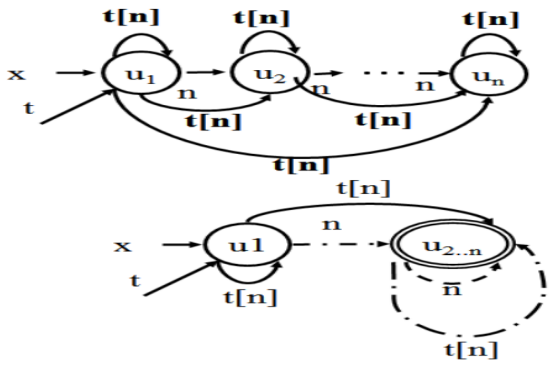


Figure 10.9: Reachability Relation.

As before, the property is evaluated for the abstract case simply as a 3-valued join between the values of the corresponding concrete case. Therefore we get:

- $t[n](u_1, u_1) = 1$, *because the value of* $t[n](u_1, u_1)$ *in the concrete representation is 1.*
- $t[n](u_1, u_{2...n}) = t[n](u_1, u_2) \sqcup = t[n](u_1, u_3) \sqcup ... \sqcup t[n](u_1, u_n) = 1 \sqcup 1 \sqcup ... \sqcup 1 = 1.$
- $t[n](u_{2...n}, u_1) = t[n](u_2, u_1) \sqcup = t[n](u_3, u_1) \sqcup ... \sqcup t[n](u_n, u_1) = 0 \sqcup 0 \sqcup ... \sqcup 0 = 0.$
- $t[n](u_{2...n}, u_{2...n}) = \bigsqcup_{i \in \{2...n\}, j \in \{2...n\}} .t[n](u_i, u_j) = \frac{1}{2}.$

We may use this property with list segments. Consider the following example:

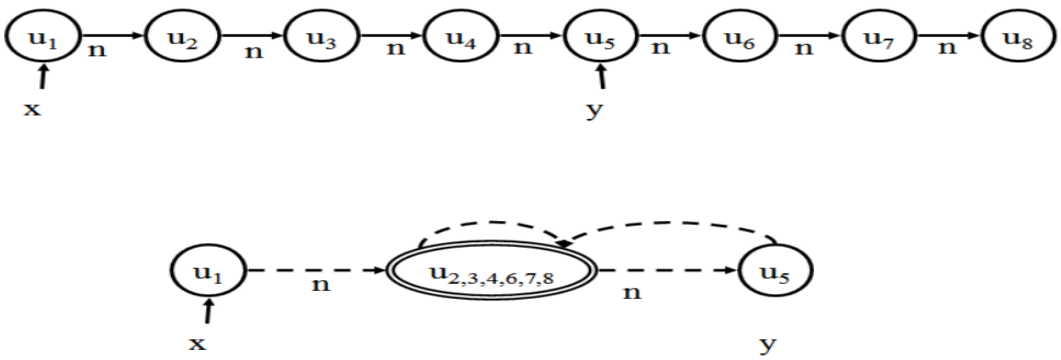


Figure 10.10: List Segments without Reachability.

As we can see, the abstract representation of the list has a few issues: even though there are no cycles in the concrete case, the abstract representation suggests that there might be. Let us now use a property similar (yet slightly changed) to the reachability property we just saw.

Let us define: $r[n, y](v) = w : y(w) \wedge n^*(w, v)$. *If we add this property to our analysis, we now get:*

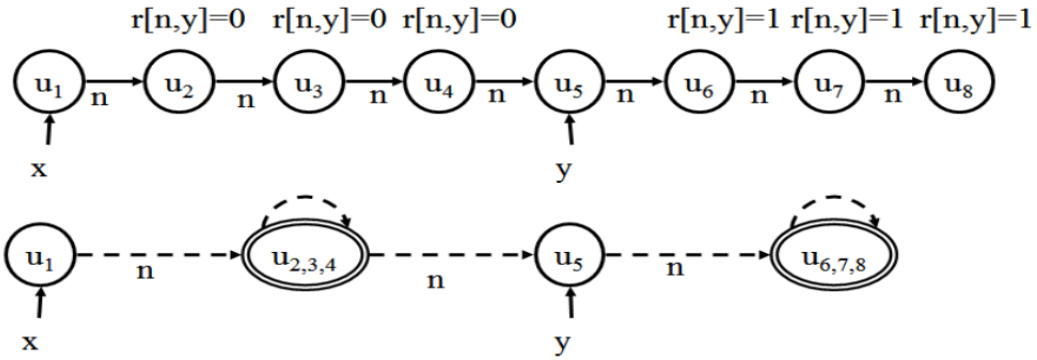


Figure 10.11: List Segments with Reachability.

As we can see, the nodes u_2, u_3, u_4 *now conform a class separate than* u_6, u_7, u_8 *due to the difference in their r[n, y] property. This solves the cycle problem seen above.*

10.7 Concrete Interpretation Rules

We saw several examples of how a transformation can be made from the concrete case to the abstract case when global invariants are used. However, when running an abstract analysis, we do not perform such a transformation but rather keep updating the configuration node's state. We saw in table 1 above how we update the unary properties for a given line of code. We can similarly define how to update the values of all relevant properties (including the user-defined global invariants). Consider the following example for the heap-sharing relation described above:

Statement	Update formula	Difference from original Table.
x =NULL	x'(v)= 0	No difference – the heap sharing property remains unchanged.
x= malloc()	x'(v) = IsNew(v) $is'(v) = is(v) \wedge \neg$IsNew(v)	If the object was shared before, and the IsNew operation was not called on the current node v, it will remain shared. Otherwise it is not (i.e. a newly allocated block of memory would always be unshared)
x=y	x'(v)= y(v)	No difference
x=y →next	x'(v)= ∃w: y(w) ∧ n(w, v)	No difference
x →next=y	n'(v, w) = $\neg x(v) \wedge n(v, w)$ $is'(v) = is(v) \wedge$ $\exists v_1, v_2 : n(v_1, v) \wedge \neg x(v_1)$ $\wedge\, n(v_2, v) \wedge \neg x(v_2)$ $\wedge\, \neg eq(v_1, v_2)$	Note that in this example we set x→next to null, while in table 1 we set it to some variable y. In this example, node v is shared iff two different nodes (which are not the node pointed to by x) point to v, and the node was marked as shared before the execution of this statement. This means that a node that was not previously shared will always remain unshared, but a node which was shared before may either be shared or unshared after this line.

Figure 10.12: Concrete Implementation rules + is-shared property.

Previously the user of TVLA had to enter these update formulas himself. In the newer version, TVLA can calculate it based on the definition of the property and the update formula for the other predicates.

10.8 Instrumentation and Embedding

We have seen several examples for instrumentation – transforming a concrete structure B of individuals (nodes) U^B and properties P^B, to an abstract structure S of individuals $U^S = \{f(u) | u \in U^B\}$ and properties $P^S = P^B \cup \{sm\}$ so that every two individuals $u_1, u_2 \in U^B$ are mapped to the same individual $(f(u_1) = f(u_2))$ if and only if they give the same result for every unary property in P^B.

In the abstract structure, the unary properties are easy to define – they give the same result as the concrete individuals that were mapped to them (recall that all the concrete individuals that are mapped to the same abstract individual have the same result on every unary property). The other properties (nullary, binary or k-ary) are defined by:

$p^S(u'_1, ..., u'_k) = \bigsqcup\{p^B(u_1, ..., u_k) | f(u_1) = u'_1, ..., f(u_k) = u'_k\}$

The instrumentation created a structure S which is a "tight-embedding" of the structure B. We say that a structure B can be embedded into a structure S via a surjective function $f : U^B \to U^S$ if all the properties are preserved (some information may be lost, but we won't get contradictions):

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

(for every k-ary property $p^B \in P^B$, its corresponding property $p^S \in P^S$, and any k individuals in U^B).

By "tight-embedding" we mean that the value of the abstract property is the least upper bound:

$p^S(u'_1, ..., u'_k) = \bigsqcup\{p^B(u_1, ..., u_k) | f(u_1) = u'_1, ..., f(u_k) = u'_k\}$

Canonical Abstraction, we described, is a tight embedding.

Notice that the "can be embedded" and "tight embedding" relations are also defined if B is a 3- valued logical structure, thus creating a partial-order $\sqsubseteq^f$ between all abstract and concrete structures.

Furthermore, we added the new property "sm" (summary node) defined by:

$$sm(u') = \begin{cases} 0, & \text{if only a single } u \text{ is mapped to } u' \\ 1/2, & \text{otherwise} \end{cases}$$

We noticed that the number of individuals in the abstract structure is finite and limited by $3^{|P^B|}$, so our analysis memory and time requirements are limited too (due to the lattice's finite depth). On the other hand, we pay for this by losing information: our abstract structure may also represent other concrete structures which cannot occur at runtime.

10.9 Embedding Theorem

During the instrumentation, we defined the values of the abstract-structure's properties so they'll preserve the values of the concrete-structure properties. For example, the concrete oneway-reachability property:
$owr[n](u_1, u_2) = [[n^+(v, w) \wedge \neg n^+(w, v)]]_{[v \to u_1, w \to u_2]}$
will be defined in the abstract structure as:
$owr^S(u_1, u_2) = \bigsqcup\{owr[n_B](u_1, u_2) | f(u_1) = u'_1, f(u_2) = u'_2\}$
This raises the question whether the abstract definition of owr preserves the result of the formula that defined it. According to the Embedding Theorem that we will soon prove, the evaluation of any FO^{TC} formula is preserved under the instrumentation. By FO^{TC} formula we mean that the formula can be constructed of:

- *The structure's atom properties.*
- *The first order-logic usage of: $\forall, \exists, \neg, \wedge, \vee$.*
- *The Transitive Closure, which we use to mark with '+', such as in n^+.*

In our previous example of owr, using the Embedding Theorem we get:

$\forall u_1, u_2 \in U^B : [[n^+_B(v, w) \wedge \neg n^+_B(w, v)]]_{[v \to f(u_1), w \to f(u_2)]}$

Or in other words: $owr[n_B](u_1, u_2) \sqsubseteq owr[n_S](f(u_1), f(u_2))$
$owr[n_B](u_1, u_2) \sqsubseteq owr[n_S](u'_1, u'_2)$ *where* $f(u_1) = u'_1, f(u_2) = u'_2$

Therefore, from the definition of the "least upper bound":
$owr^S(u'_1, u'_2) = \bigsqcup\{owr[n_B](u_1, u_2) | f(u_1) = u'_1, f(u_2) = u'_2\} \sqsubseteq owr[n_S](u'_1, u'_2)$. *This means that owr^S preserves the formula, and may even be more precise. Using the Embedding Theorem, it is easy to see how we can prove this for any k-ary property that is defined by formula – the new property will preserve the evaluation of the formula in the abstract structure.*

Here we have to note that a formula that includes equality, such as "v=w" is not preserved "as-is". If we look at the different concrete individuals u_1, u_2 that are mapped to the same abstract summary node individual $u' = f(u_1) = f(u_2)$, we will obviously get a contradiction:

- $[[v = w]]_{v \to u_1, w \to u_2} = 0$

- $[[v = w]]_{v \to f(u_1), w \to f(u_2)} = 1$

In order to fix this, such formulas are translated to the abstract world as: $v = w \wedge \neg sm(v)$

- *If two different abstract individuals are compared, the result will be 0, as with any 2 concrete individuals that are mapped to 2 different abstract individuals.*
- *If a non-summary individual is compared to itself the result will be 1, as with the only concrete individual that was mapped to it.*
- *If a summary individual is compared to itself the result will be* $\frac{1}{2}$*, which means that we don't know what result will be in the concrete structure.*

Let's look at a structure B which can be embedded into a structure S by a surjective $f : U^B \to U^S$ *(denoted as* $B \sqsubseteq^f S$*), and let* ϕ *be some formula with the free variables* $v_1, ..., v_k$*. By De Morgan laws we can assume WLOG that* ϕ *is constructed only by* $\wedge, \neg, \exists$ *and TC of smaller formulas, or that* ϕ *is an atomic formula.*

It can be shown by induction that for any assignment Z of the free variables $v_1, ..., v_k$ *to some individuals* $u_1, ..., u_k \in U^B$ *(respectively) we get* $[[\phi]]_Z \sqsubseteq [[\phi']]_{foZ}$ *where:*

- foZ *is the assignment that maps the free variables* $v_1, ..., v_k$ *to* $f(u_1), .., f(u_k)$ *(respectively)*
- ϕ' *is the same formula as* ϕ*, except for the terms of the form* $(v_i = v_j)$ *which are replaced with* $v_i = v_j \wedge \neg sm(v_i)$

All the evaluations are done under the 3-valued logic. The theorem is also true in the sub-case when B is a 2-values structure (a concrete structure).

In order to prove that $[[\phi]]_Z \sqsubset [[\phi']]_{foZ}$*, we have to show that:*

- *if* $[[\phi']]_{foZ} = 1$ *then* $[[\phi]]_Z = 1$
- *if* $[[\phi']]_{foZ} = 0$ *then* $[[\phi]]_Z = 0$
- *if* $[[\phi']]_{foZ} = 1/2$ *the claim is true (nothing to prove in this case).*

10.10 Transformers

10.10.1 Best Transformer

As we mentioned before, instead of maintaining all the possible concrete structures in every code line in the shape analysis, we only maintain a collection possible of abstract structures, knowing that every possible concrete structure is represented by the abstract structure that it can be embedded to. Suppose that we have a collection of abstract structures for some code line. We now have to transform them into a new abstract collection, which will represent (by embedding) all the concrete structures after the code line has been executed. Obviously, the most accurate transformer would be to look at all the concrete structures represented by the current abstract collection, execute the code line (the concrete transformation) on each one, and find their abstract representations. However, this "Best Transformer" is not feasible – the number of concrete structures represented by an abstract structure can be infinite (for example, in the case of a linked list, which can represent any linked list of any length).

10.10.2 Kleene Transformer

We have seen before the "Concrete Interpretation Rules", which tell us how to evaluate the relations in the new concrete structure, based on the current concrete structure and the code line to execute. For example:

Statement	Update Formula	Explanation
x = y->next	x'(v) = ∃w: y(w) ∧ n(w, v)	For every node v, x of v is true (i.e. 1) iff there exists a node w pointed to by y, and there is a path (of length 1) from w to v.

The Kleene Transformer simply evaluates the update-formula on the abstract structure, in a 3- valued logic. According to the Embedding Theorem, if we have a concrete structure that can be embedded into an abstract structure $B_{before} \sqsubseteq^f S_{before}$ (f maps the concrete individuals to their abstract individuals) then evaluations of such a formula ϕ will obey:

$[[\phi]]_Z^{B_{before}} \sqsubseteq [[\phi']]_{foZ}^{S_{before}}$

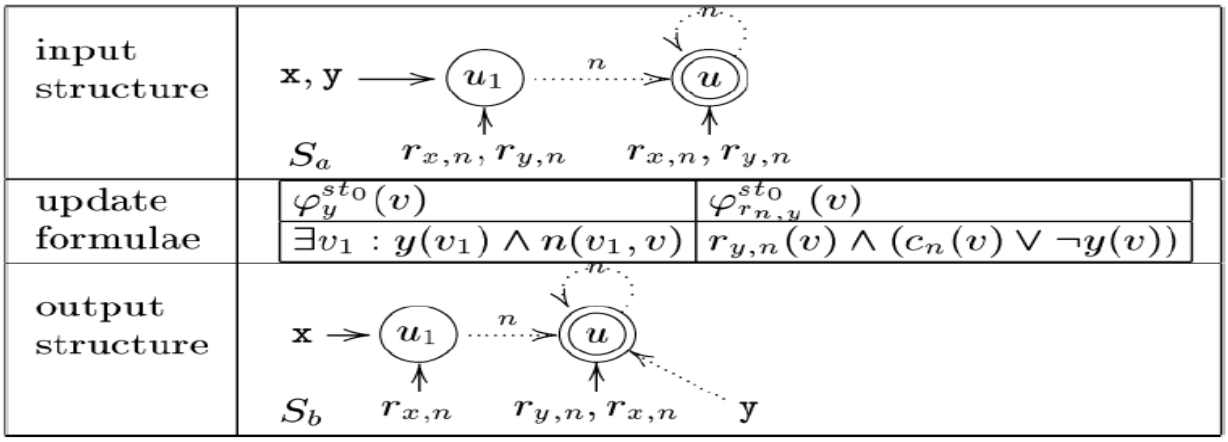
Therefore, every nullary, unary, binary or k-ary property $p(u_1, ..., u_k)$ will be preserved:

$p^{B_{after}}(u_1, ..., u_k) \sqsubseteq p^{S_{after}}(f(u_1), ..., f(u_k))$

This means that B_{after} can be embedded into S_{after}, so with this transformation the new abstract structures for sure represent all the required concrete structures.

However, this technique may lose important information, as the new properties may resolve in $\frac{1}{2}$.

For example, let's look at the simple execution of the statement $y = y \to n$, on an abstract structure where y points to the first node of a linked list.



As we can see, we lost the information about the node that is pointed by y (we can still conclude that it is not null because there are reachable nodes).

10.10.3 Focusing and Semantic Reduction

We want to have a transformation which will be more accurate than the Kleene transformation, and will also be feasible. We do that by picking an indecisive property in our abstract structure (a property that

evaluates to $\frac{1}{2}$), and from all the abstract structures that can be embedded into our structure, we find all the maximal abstract structures in which this property is decisive: all the maximal abstract structures in which this property evaluates to 1 and all the maximal abstract structures in which this property evaluates to 0.

It is easy to see that every concrete structure that can be embedded into the original abstract structure can be embedded into one of the new abstract structures, because the properties of concrete structures are always decisive. Furthermore, because of the transitivity of the embedding, every concrete structure that can be embedded into one of the new abstract structure can be also embedded into the original structure. Therefore, the original abstract structure and the collection of new abstract structures represent the same set of concrete structures.

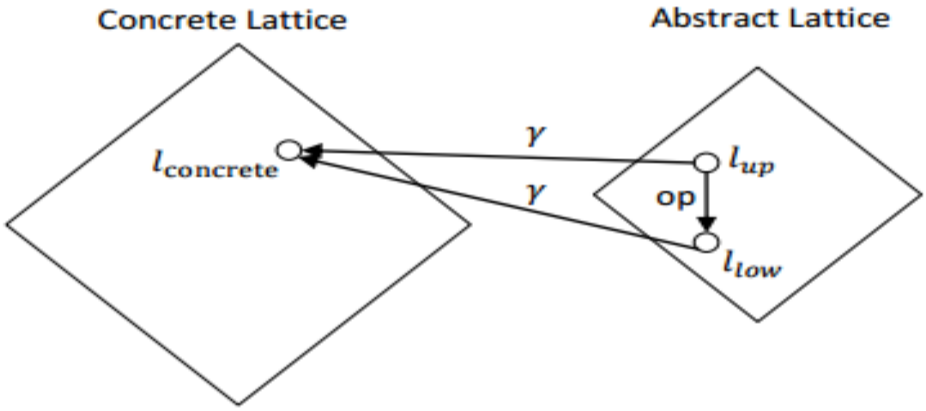


Figure 10.13: Example for a semantic reduction operation op: $op(l_{up}) = l_{low} \sqsubset l_{up}$ and $\gamma(l_{up}) = \gamma(l_{low})$

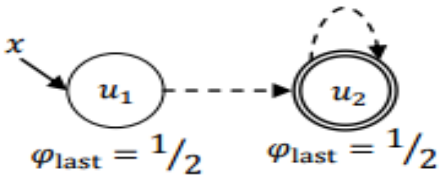
This technique is called Focusing. The idea of taking an element in our lattice (the original abstract structure) and converting it to a more precise element (the collection of new abstract structures), in a way that preserves their transformation to the concrete world, is called Semantic Reduction.

The property that we focus on does not necessarily have to be one of the properties that we track during the analysis. It can also be defined by a new formula. After focusing, we can apply the Kleene Transformer on every new abstract structure, and then "unfocus" by forgetting about the new property and non-maximal abstract structures - after removing the property, we can ignore structures that can be embedded into other structures in the collection. In case the removed property was unary, we must also merge individuals that differed from each other only in that property, and have identical values in all the other properties.

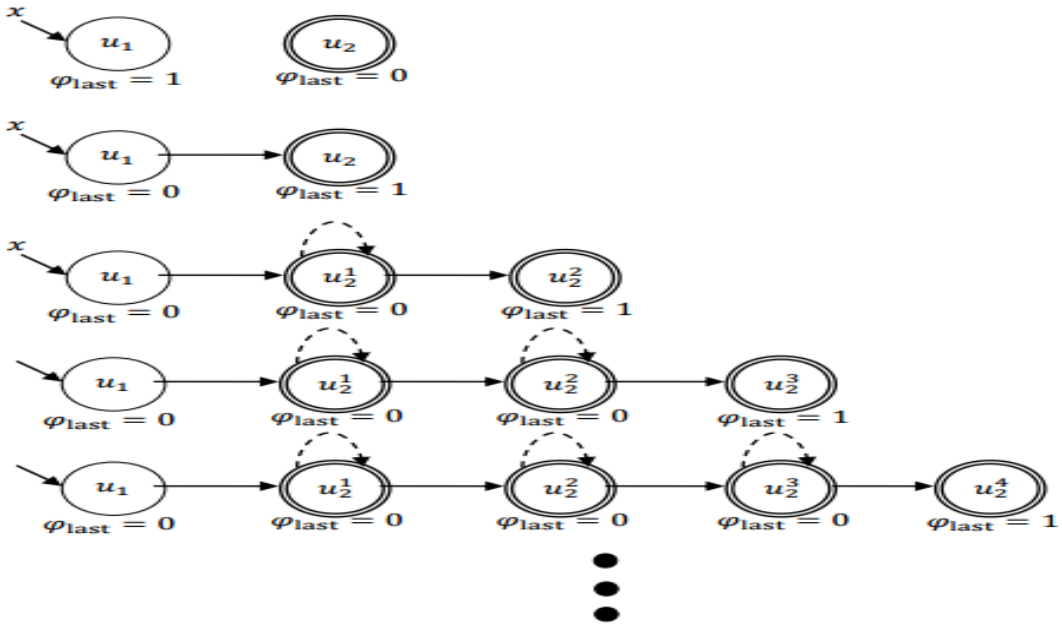
Not all formulas can be used in the focusing process. For example, let's look at the formula:

$\phi_{last}(v) \equiv \forall v_1 : \neg n(v, v_1)$

And the abstract structure for a linked list pointed by x:



In u_1, ϕ_{last} gets an indecisive value because $n(u_1, u_2) = 1/2$. In u_2 we also get an indecisive value because $n(u_2, u_2) = 1/2$. Let's look at some abstract structures that can be embedded into our structure, and have decisive values for ϕ_{last}.



All of the structures above can be embedded into the original structure. They also represent different concrete structures: a list of length m can be embedded only to the structure number m. Therefore, they cannot be embedded into one another. Finally, and most important, any union of any couple of these structures will create an abstract structure with an individual that all of its outgoing edges are $\frac{1}{2}$'s, and therefore it will have an indecisive $\phi_{last} = 1/2$. This means that all of the abstract structures do not have common maximal "focused" abstract structures (in fact, these structures are maximal, but we won't prove it here). The Focus transformation will generate an infinite number of abstract structures, and therefore it is not feasible.

Luckily, we have a smart way of choosing our focus formulas for statements of the form $lhs = rhs$, as we can see in figure 10.14 (next page).

The idea behind this table is:

- In the "read" part (rhs), we want to be decisive on which individual the pointer that we read points to, or in other words we want to know decisively whether it points or doesn't point to every node. For

example, the statement $x == NULL$ and its focus formula $x(v)$, or the statement $x = t \to n$ (reading from $t \to n$), and its focus formula $\exists v_1 : t(v_1) \wedge n(v_1, v)$

- In the "write" part (rhs), we want to be decisive on which individual is going to change. For example in $x = NULL$, no nodes are going to change, but in $x \to n = t$, the node pointed by x is going to change (its "next" attribute will change), so we want to be decisive on the formula $x(v)$.

st	Focus Formulae
`x = NULL`	$\emptyset$
`x = t`	$\{t(v)\}$
`x = t->n`	$\{\exists v_1 : t(v_1) \wedge n(v_1, v)\}$
`x->n = t`	$\{x(v), t(v)\}$
`x = malloc()`	$\emptyset$
`x == NULL`	$\{x(v)\}$
`x != NULL`	$\{x(v)\}$
`x == t`	$\{x(v), t(v)\}$
`x != t`	$\{x(v), t(v)\}$
`UninterpretedCondition`	$\emptyset$

Figure 10.14: The target formulas for focus, for statements and conditions of a program that uses type List

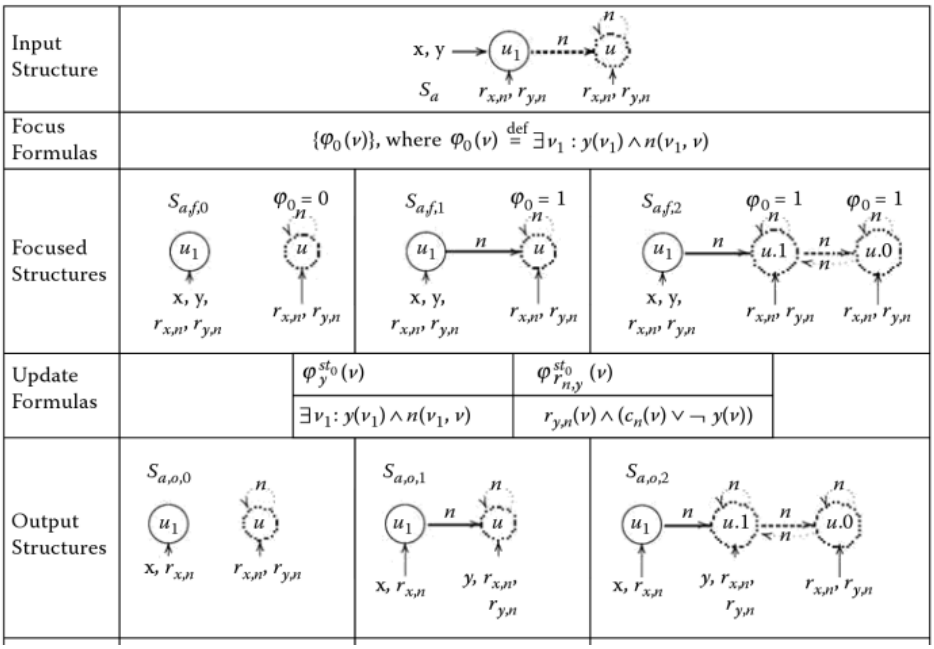
We can see that our focus formulas ask questions of: "Is there a route of length [specific length] from the stack pointer [specific stack pointer] to the node?" – For example, in the case of reading $t \to n \to n$, the focus formula $\phi(v) \equiv \exists v_1 \exists v_2 : t(v_1) \wedge n(v_1, v_2) \wedge n(v_2, v)$ asks the question: "Is there a route of length 3 from the pointer to the node v?".

If, for some node, there are indecisive routes (with indecisive edges) of the required length from the pointer to the node, and there are no decisive routes of the required length from the pointer to the node, the evaluation of this formula will be indecisive.

For such a node, we create the focused structures in 3 ways:

- For every indecisive route from the pointer to the node, create the structure in which the edges of this graph are resolved to 1 – this will make the formula resolve to 1.
- In every indecisive route from the pointer to the node, pick a single indecisive edge and make it resolve to 0 (if 2 routes share a common indecisive edge, it can be picked for both of them) – this will make the formula resolve to 0.
- If the node is a summary node, we should recall that the abstract individuals are distinguished by their unary properties. If we add a new unary property, we should split the summary node to 2 (duplicating its relations with the other nodes), and make one copy resolve to 1 and the other copy resolve to 0 like before.

For example, in the statement $y = y \to n$, where x, y point to the first node of a linked list, we get (the r property is the reachability):



Eventually, we should "drop" the new property. We do that by finding individuals that are equivalent to each other by the values of all "normal" unary properties, and merging them.

10.10.4 Coercion

We can see in the last example that we reached the output structure:

This structure represents all the concrete structures where y decisively points to NULL, and there are list-nodes, represented by u, that are reachable from y. Obviously, no concrete structures can create this situation, and a smart algorithm would drop this irrelevant structure.

The Coercion is another type of semantic reduction. It takes a collection of known constraints between some properties, and try to use them in order to make the indecisive properties more precise. The Coercion Principle (or Sharpening Principle) states that if in the concrete world a property $p(u_1, ..., u_k)$ equals to an assignment in some FO^{TC} formula: $[[\phi]]_{[v_1 \to u_1, ..., v_k \to u_k]}$ (from definition or from a known constraint) then in any abstract structure, $p(u'_1, ..., u'_k)$ should be at least as precise as the evaluation of $[[\phi]]_{[v_1 \to u'_1, ..., v_k \to u'_k]}$. Furthermore, if $p(u'_1, ..., u'_k)$ has a definite value and $[[\phi]]_{[v_1 \to u'_1, ..., v_k \to u'_k]}$ has an incomparable definite value, then the abstract structure does not represent any concrete structure at all.

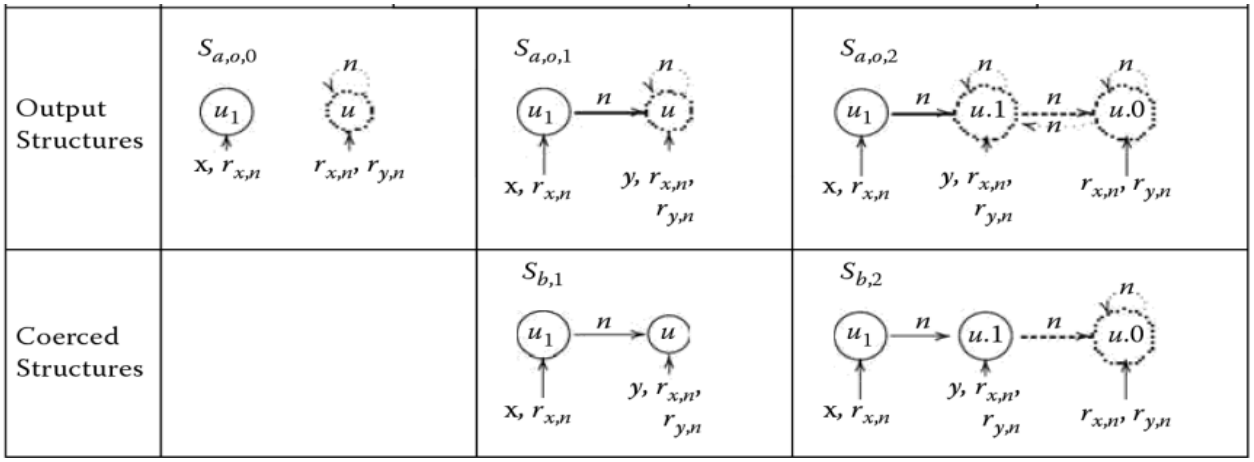
Some examples for known constraints:

- $x(v_1) \wedge x(v_2) \rightarrow eq(v_1, v_2)$
- $x(v_1) \rightarrow \neg sm(v_1)$
- $n(v_1, v) \rightarrow \neg sm(v)$
- $n(v, v_1) \wedge n(v, v_2) \rightarrow eq(v_1, v_2)$
- $n(v_1, v) \wedge n(v_2, v) \wedge \neg eq(v_1, v_2) \leftrightarrow is(v)$
- $n(v_1, v) \wedge \neg is(v) \wedge \neg eq(v_1, v_2) \leftrightarrow \neg n(v_2, v)$
- $r_{y,n}(v) \leftrightarrow y(v_1) \wedge n^*(v_1, v)$

In our last example, assigning $v \rightarrow u$ in the 7^{th} constraint with $v_1 \rightarrow u_1$ results in two incomparable definite values:

$r_{y,n}(u) = 1$
$[[y(v_1) \wedge n^*(v_1, v)]]_{[v_1 \rightarrow u_1, v \rightarrow u]} = 0$

Therefore, we can drop the whole structure.



We can also apply the coercion on the other two output structures in our last focusing example. Using the third constraint with $v_1 \rightarrow u_1$, we can see that u and $u.1$ are not summary nodes. Looking at the "is shared" property (not shown in the diagrams because all nodes have $is(v) = 0$) in the 6^{th} constraint, with $v \rightarrow u, v_1 \rightarrow u_1, v_2 \rightarrow u$ in the central case ($S_{a,0,1}$), results in removing the indefinite self edge of u ($\neg n(u, u)$). The 6^{th} constraint can also be used in the right case ($S_{a,0,2}$), with $v \rightarrow u.1, v_1 \rightarrow u_1$, to remove all the indefinite edges towards $u.1$: $v_2 \rightarrow u.1$ for $\neg(u.1, u.1)$ and $v_2 \rightarrow u.0$ for $\neg n(u.0, u.1)$.

In the following "list-insertion" example (Figure 10.13), we can see the big accuracy difference in the final states between the Kleene (One-Stage) simplistic analysis, and the Multi-Stage analysis that uses focusing and coercion.

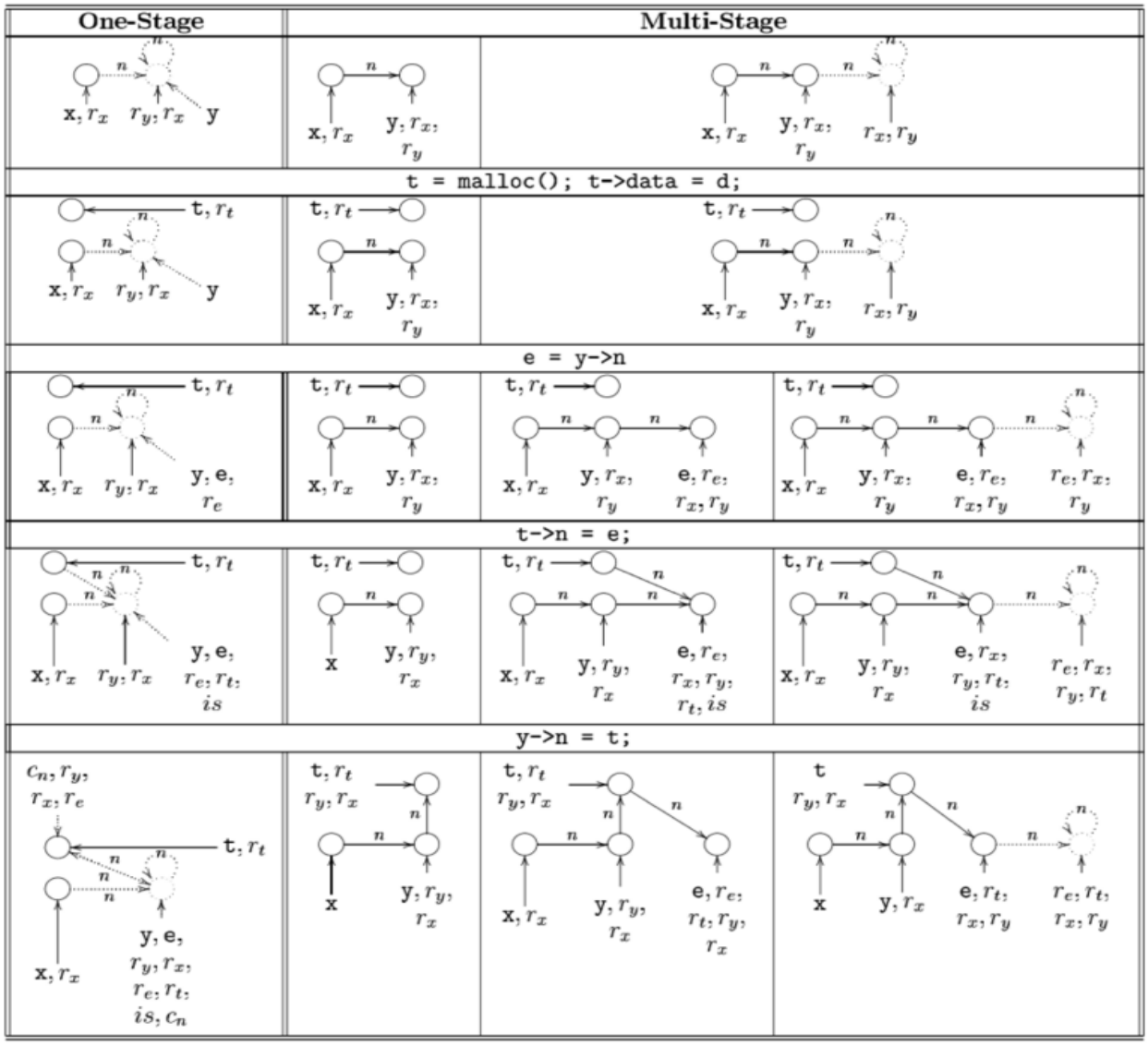


Figure 10.15: Selective applications of the abstract transformers using the one-stage and the multi-stage approaches, for the statements in insert that come after the search loop.

10.11 Summary Transformers

- Kleene evaluation yields sound solution
- Focus is statement specific implements partial concretization
- Coerce applies global constraints

10.12 TVLA

Three Valued Logic Analysis (TVLA) T. Lev-Ami and R. Manevich

- Input (FOTC)
  - Concrete interpretation rules
  - Definition of instrumentation relations
  - Definition of safety properties
  - First Order Transition System (TVP)
- Output
  - Warnings (text)
  - The 3-valued structure at every node (invariants)

10.13 Interprocedural Analysis

10.13.1 Procedure Handling

The simple case is dealing with pure function - i.e. function with no side-effects. In this case a procedure is simply input/output relation

```
main() {
    int
    w=0,x=0,y=0,z=0;
    w = inc(y);
    x = inc(z);
    assert:
    w+x is even
}
```

```
int inc(int p) {
    return 2 +
    p - 1;
}
```

p	ret
0	1
1	2
2	3
3	...

Figure 10.16: Simple procedure handling example.

As one can see this case is handled easily by methods shown previously.

10.13.2 Global Variables

In the above case, we ignored global variables. With the presence of global variables, function can have side-effects! This is a harder problem than before, how can we update global variable from within a procedure?

Can be fixed by adding global variables evaluation to the tables (g' in this example- representing the global variable g).

```
int g = 0
main() {
    int
    w=0,x=0;
    int
    y=0,z=0;
    w =
    inc(y);
    x =
    inc(z);
    assert:
    w+x+g
    is
    even
}
```

```
int inc(int p) {
    g = p;
    return 2
    + p - 1;
}
```

p	g	ret	g'
0	1	1	0
...	...	...	...

p	g	ret	g'
E	EO	O	E
O	EO	E	O

Figure 10.17: Procedure with side-effects handling example.

10.13.3 Heap and Pointers

For pointers we have aliasing and destructive updates, heap introduce problems by adding global variables and anonymous objects to the equation.

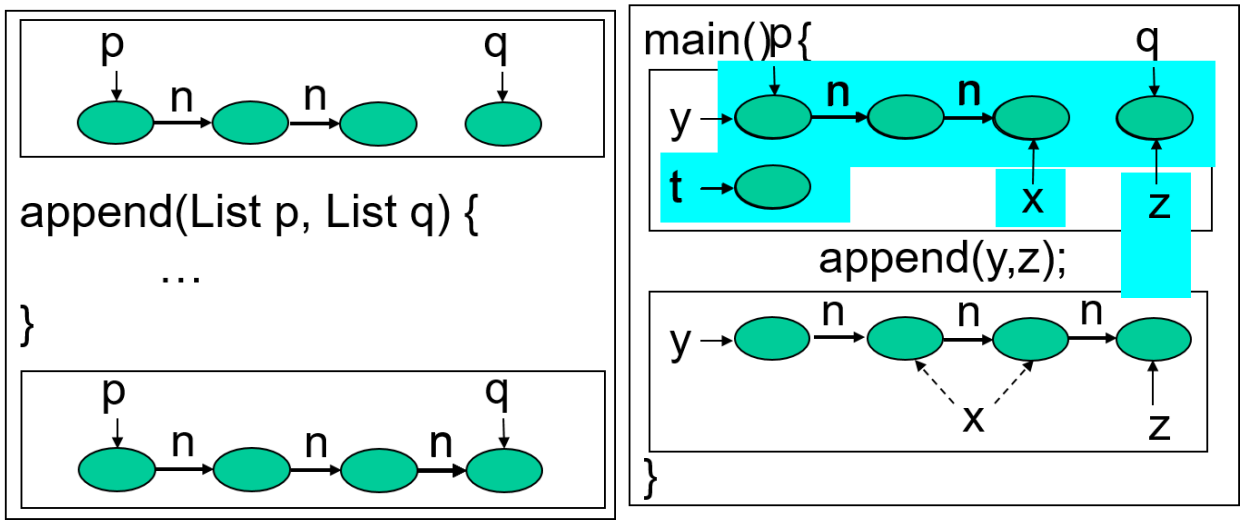


Figure 10.18: External sharing which might break the functional view; append(y,z).

Now we have situation where objects are "bypass" of the parameters.

10.13.4 Cut-Points

An interesting observation can be made by distinguishing 2 types of object - regular object, and object which we call Cut-Points (this is a run-time observation).

An object is a cut-point for an invocation if:

- It is reachable from an invocation parameter
- It is not pointed by an actual parameter
- It is reachable without going through an invocation parameter

A Cutpoint is simply an object which is reachable also from another source than the parameters.

Cutpoint examples:

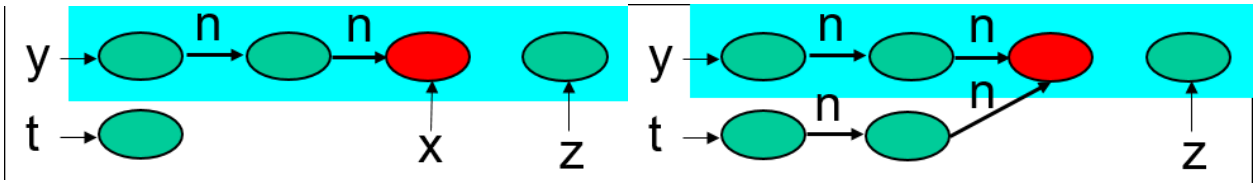


Figure 10.19: Cutpoints in Red; append(y,z).

Cutpoints has great significance in run-time, for example for a Garbage Collector, which could only consider the local variables and the Cutpoint.

Performance:

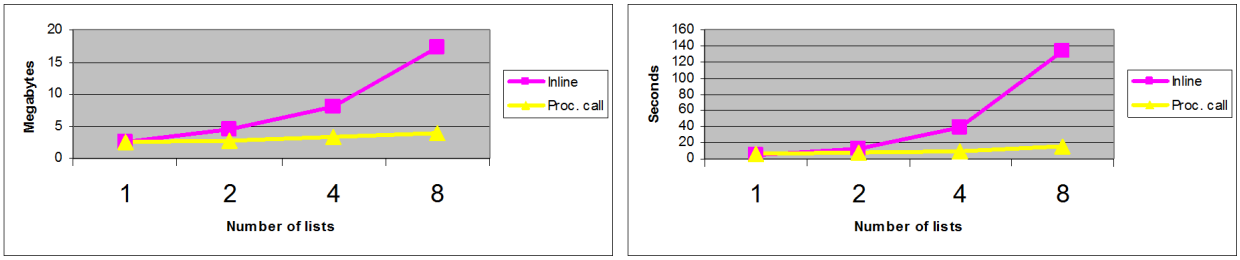


Figure 10.20: Inline vs. Procedural abstraction (Performance).

As expected, inlining running-time and space complexity is exponentialy increasing (Linear relation), while Procedural Abstraction showing little effect for the number of calls.