May 11

Compile-Time Verification of Properties of Heap Intensive Programs. Mooly Sagiv, Thomas Reps, Reinhard Wilhelm.

Notes by: Eran Kravitz & Oren Zomer

http://www.cs.tau.ac.il/~TVLA

http://www.cs.tau.ac.il/~msagiv/toplas02.pdf

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.

Logical Structures (Labeled Graphs)

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

- Nullary relation symbols
- Unary relation symbols
- Binary relation symbols

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

The analysis only stores tables containing the following information:

- A set of individuals (nodes) U.
- Properties given by the relation symbols in P:
  - $P^0()$ →{0, 1}
  - $P^1(v)$ →{0, 1}
  - $P^2(u, v)$ →{0, 1}

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. $n(u_1, u_2)$=1 means that the next field of u_1 directly points to u_2).

Following is an example of the above, for a program with a list of up to 4 elements:

Name	Logical Structure	Graphical Representation
$S_0^\sharp$	unary preds.: indiv. | x | y | t | e — binary preds.: n	
$S_1^\sharp$	unary preds.: indiv. | x | y | t | e; u_1 | 1 | 0 | 0 | 0 — binary preds.: n | u_1; u_1 | 0	x → u_1
$S_2^\sharp$	unary preds.: indiv. | x | y | t | e; u_1 | 1 | 0 | 0 | 0; u_2 | 0 | 0 | 0 | 0 — binary preds.: n | u_1 | u_2; u_1 | 0 | 1; u_2 | 0 | 0	x → u_1 →^{n} u_2
$S_3^\sharp$	unary preds.: indiv. | x | y | t | e; u_1 | 1 | 0 | 0 | 0; u_2 | 0 | 0 | 0 | 0; u_3 | 0 | 0 | 0 | 0 — binary preds.: n | u_1 | u_2 | u_3; u_1 | 0 | 1 | 0; u_2 | 0 | 0 | 1; u_3 | 0 | 0 | 0	x → u_1 →^{n} u_2 →^{n} u_3
$S_4^\sharp$	unary preds.: indiv. | x | y | t | e; u_1 | 1 | 0 | 0 | 0; u_2 | 0 | 0 | 0 | 0; u_3 | 0 | 0 | 0 | 0; u_4 | 0 | 0 | 0 | 0 — binary preds.: n | u_1 | u_2 | u_3 | u_4; u_1 | 0 | 1 | 0 | 0; u_2 | 0 | 0 | 1 | 0; u_3 | 0 | 0 | 0 | 1; u_4 | 0 | 0 | 0 | 0	x → u_1 →^{n} u_2 →^{n} u_3 →^{n} u_4

Figure 1 - Concrete State Representation as Logical Structures

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)∧ n(v, w)) ∨ (x(v) ∧ y(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).

Table 1 - Concrete Interpretation Rules

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:

AND	True (1)	Unknown (½)	False (0)
True (1)	1	½	0
Unknown (½)	½	½	0
False (0)	0	0	0

Table 2 - Kleene's 3-Valued Logic - And Operation

OR	True (1)	Unknown (½)	False (0)
True (1)	1	1	1
Unknown (½)	1	½	½
False (0)	1	½	0

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

As mentioned above, a value of ½ 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 ½ functions as top, i.e. 0⊔1 = ½.

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 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:

$$p^S(u'_1, \dots, u'_k) = \sqcup \{p^B(u_1, \dots, u_k) \mid f(u_1) = u'_1, \dots, f(u_k) = u'_k\}$$

Remember that we are using a 3-valued logic, so the resulted value may be in {0, 1, ½} . 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.

Example: 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:

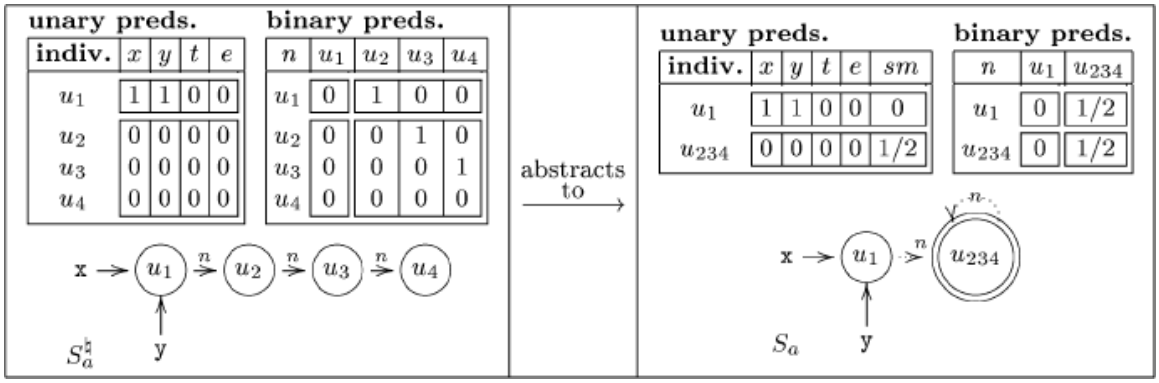


Figure 2 - Concrete to Abstract State Transformation

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 $n(u_1, u_1) = 0$.
- $n(u_1, u_{234}) = ½$, because: $n(u_1, u_{234}) = n(u_1, u_2) \sqcup n(u_1, u_3) \sqcup n(u_1, u_4) = 1 \sqcup 0 \sqcup 0 = ½$.
- $n(u_{234}, u_1) = n(u_2, u_1) \sqcup n(u_3, u_1) \sqcup n(u_4, u_1) = 0 \sqcup 0 \sqcup 0 = 0$.

- $n(u_{234}, u_{234}) = n(u_2, u_2) \sqcup n(u_2, u_3) \sqcup n(u_2, u_4) \sqcup$
  $n(u_3, u_2) \sqcup n(u_3, u_3) \sqcup n(u_3, u_4) \sqcup$
  $n(u_4, u_2) \sqcup n(u_4, u_3) \sqcup n(u_4, u_4) = 0 \sqcup 1 \sqcup 0 \sqcup 0 \sqcup 0 \sqcup 1 \sqcup 0 \sqcup 0 \sqcup 0 = ½.$

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 ½ (where ½ 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. We may, however, get warnings which would not be true (as we saw in the previous lecture with the example of the rotate method).

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

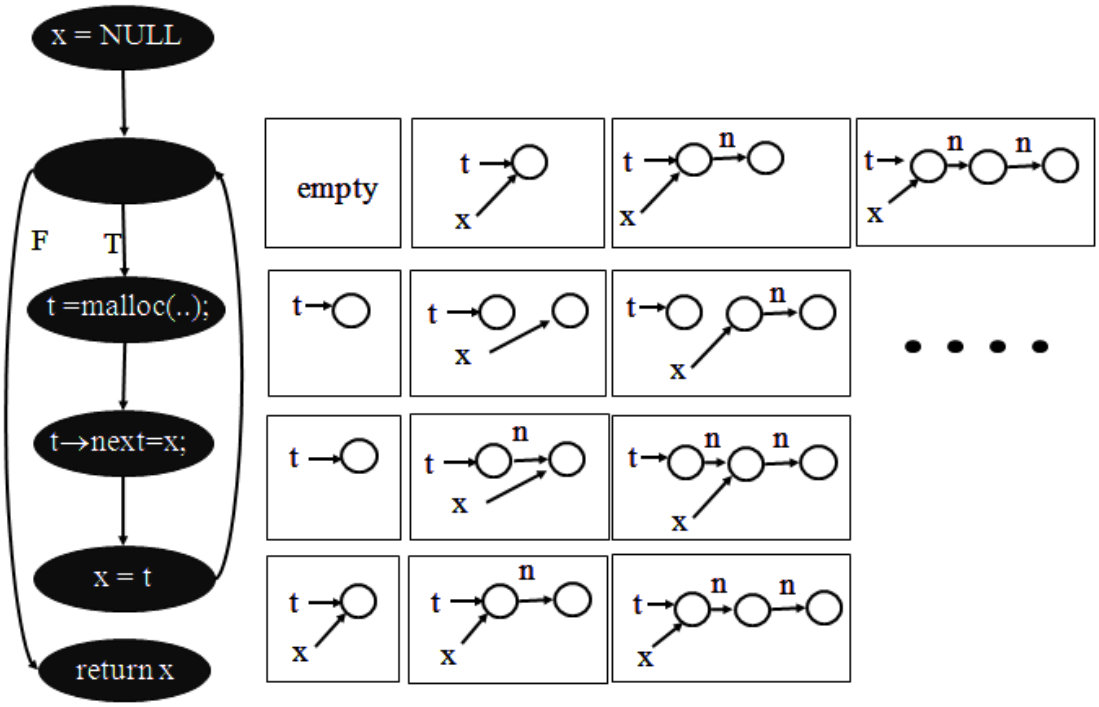


Figure 3 - List Creation Example, Concrete Representation

We would like to verify that:

1. The code given in the example does not leak memory.
2. 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:

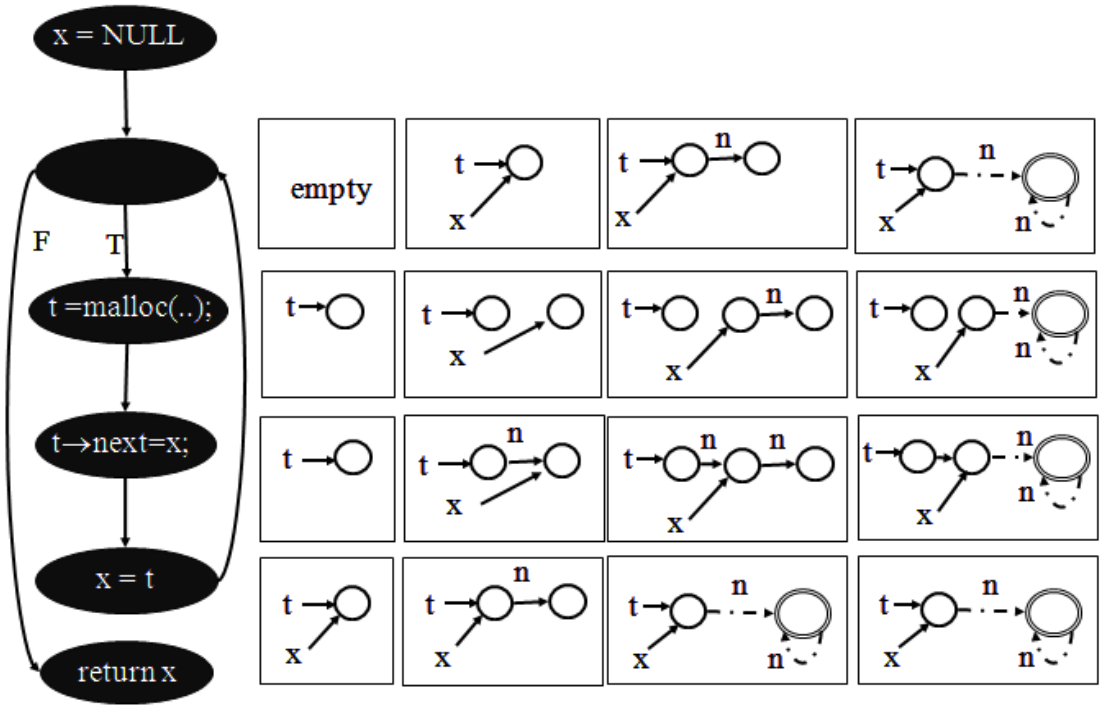


Figure 4 - List Creation Example, Abstract Representation

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.

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:

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:

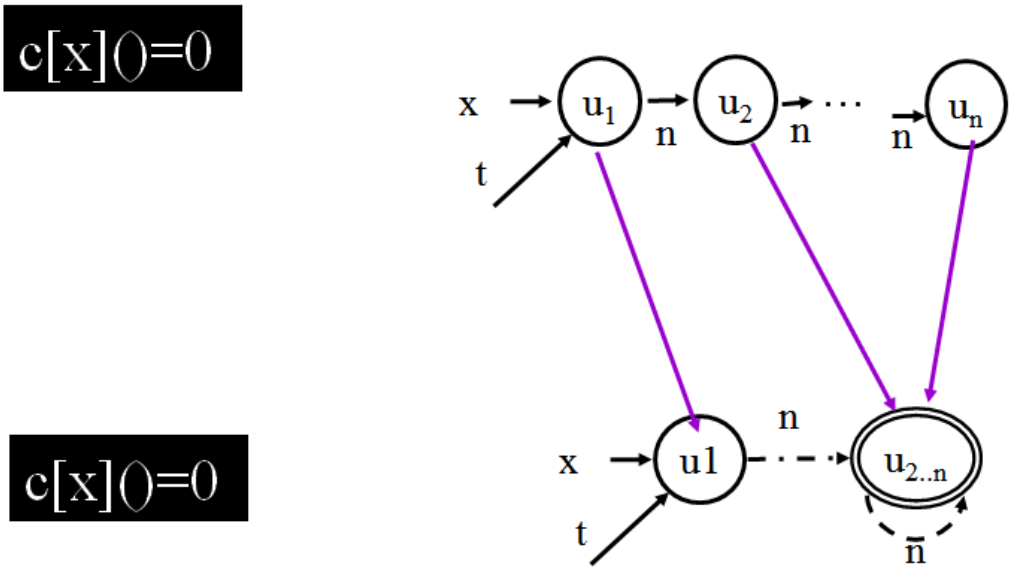


Figure 5 - Cyclicity Relation with no Cycles

And for a list that *does* contain a cycle:

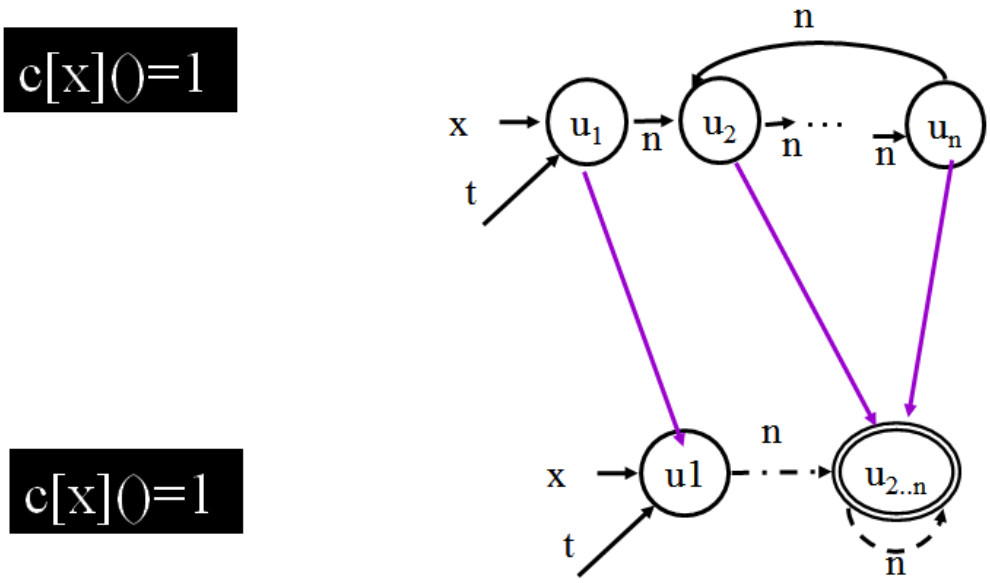


Figure 6 - Cyclicity Relation with Cycles

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.

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:

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

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):

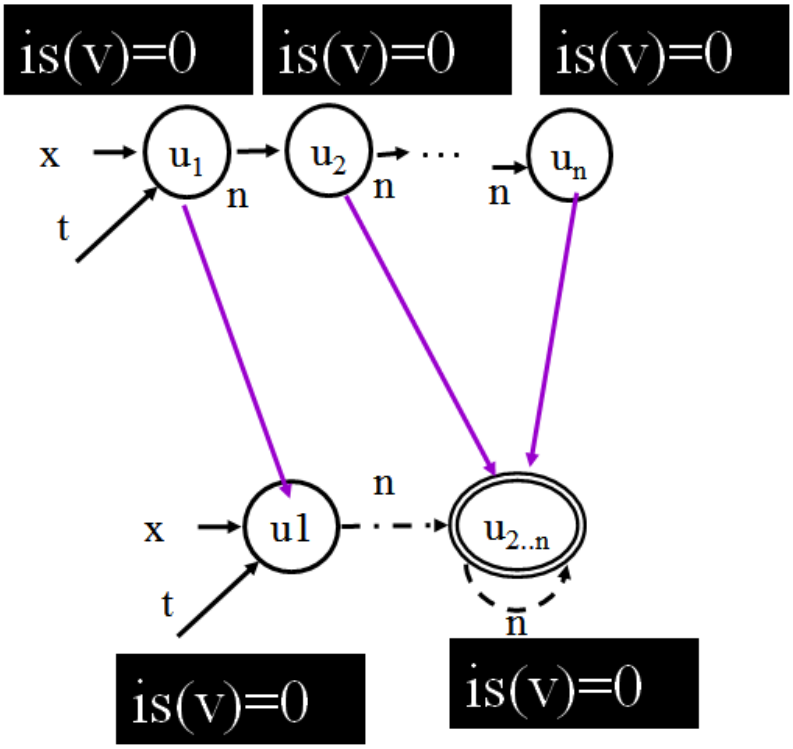


Figure 7 - Heap Sharing Relation with no Cycles

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:

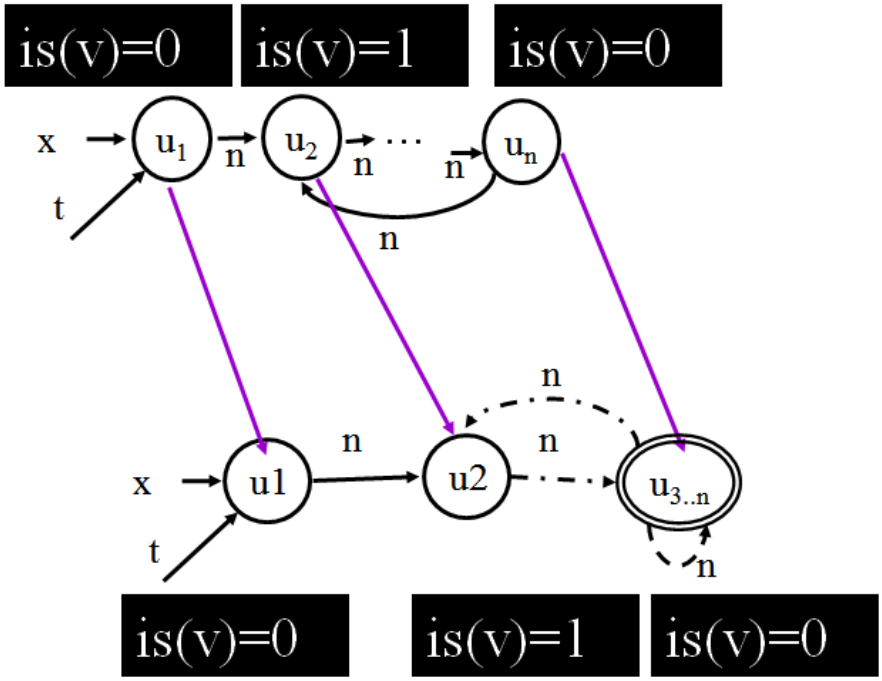


Figure 8 - Heap Sharing Relation with Cycles

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 $\{u_3, u_4, \ldots, u_n\}$ (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).

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:

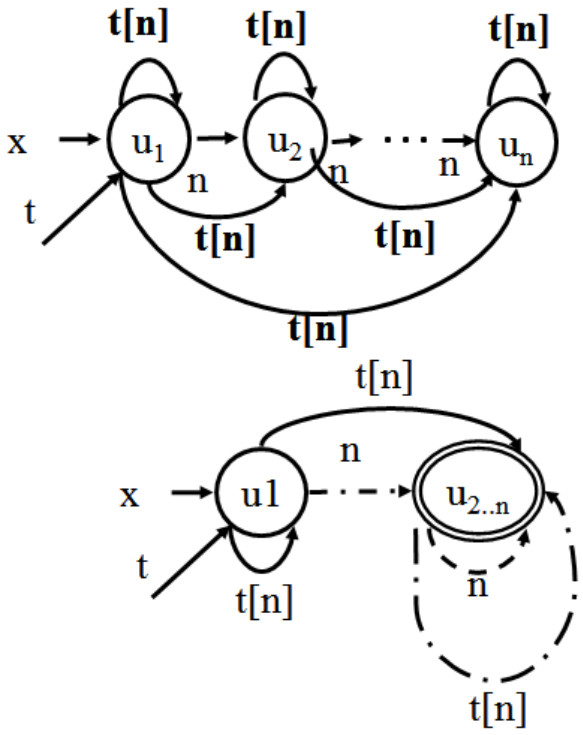


Figure 9 - Reachability Relation

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.
- $t[n](u_1, u_{2..n}) = t[n](u_1, u_2) \sqcup t[n](u_1, u_3) \sqcup \ldots \sqcup t[n](u_1, u_n) = 1 \sqcup 1 \sqcup \ldots \sqcup 1 = 1$.
- $t[n](u_{2..n}, u_1) = t[n](u_2, u_1) \sqcup t[n](u_3, u_1) \sqcup \ldots \sqcup t[n](u_n, u_1) = 0 \sqcup 0 \sqcup \ldots \sqcup 0 = 0$.
- $t[n](u_{2..n}, u_{2..n}) = t[n](u_2, u_2) \sqcup t[n](u_2, u_3) \sqcup \ldots t[n](u_2, u_n) \sqcup$
  $t[n](u_3, u_2) \sqcup t[n](u_3, u_3) \sqcup \ldots t[n](u_3, u_n) \sqcup$
  $\ldots$
  $t[n](u_n, u_2) \sqcup t[n](u_n, u_3) \sqcup \ldots t[n](u_n, u_n) = ½$ (note that for $t[n](u_i, u_j)$ where $i \leq j$ we get 1, while where $i > j$ we get 0).

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

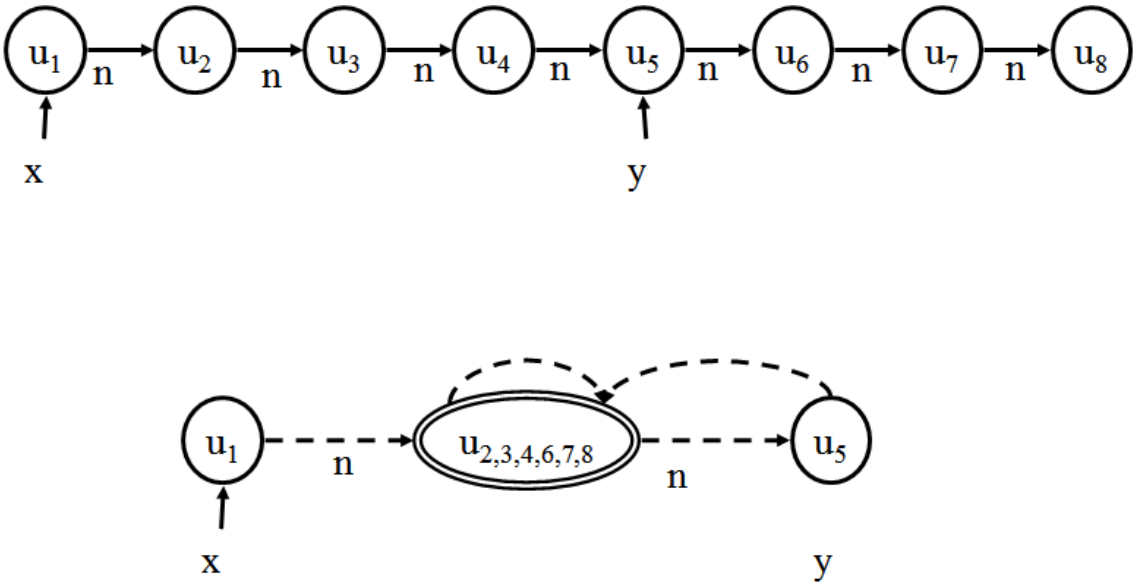


Figure 10 - List Segments without Reachability

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:

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

If we add this property to our analysis, we now get:

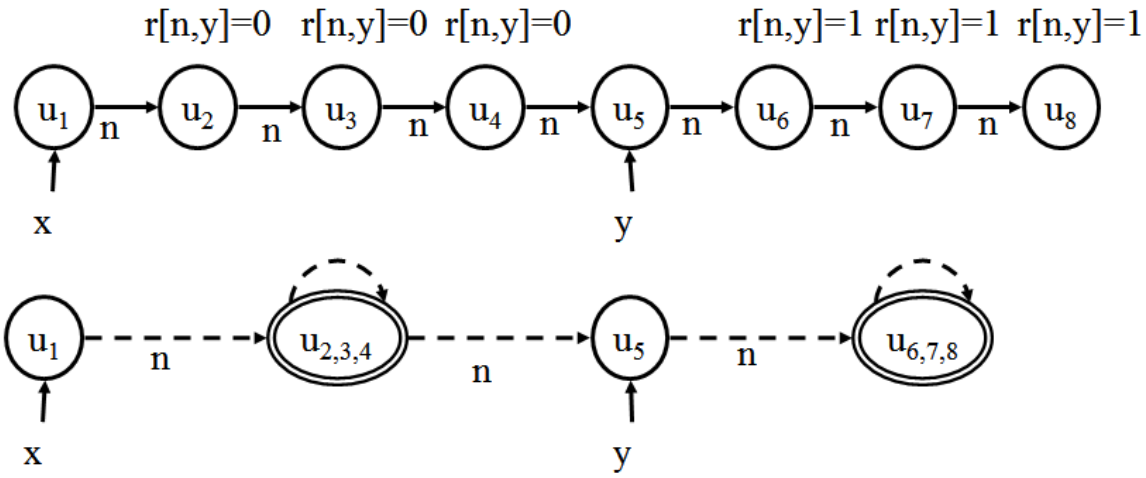


Figure 11 - List Segments with Reachability

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.

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	Change from Table 1
x = NULL	$x'(v) = 0$	No change – 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 piece of memory would always be unshared).
x = y	$x'(v) = y(v)$	No change.
x = y->next	$x'(v) = \exists w: y(w) \wedge n(w, v)$	No change.
x→next = NULL	$n'(v, w) = \neg x(v) \wedge n(v, w)$ $is'(v) = is(v) \wedge \exists v_1, v_2: n(v_1, v) \wedge \neg x(v_1) \wedge n(v_2, v) \wedge \neg x(v_2) \wedge \neg eq(v_1, v_2)$	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.

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.

Instrumentation and Embedding

We have seen several examples for instrumentations – 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 $P^S = P^B \cup \{\text{sm}\}$, 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:

$$p^S(u'_1, \dots, u'_k) = \sqcup \{p^B(u_1, \dots, u_k) \mid f(u_1) = u'_1, \dots, f(u_k) = u'_k\}$$

Or in other words:

$$p^S(u'_1, \dots, u'_k) = \begin{cases} 1 & \text{if } p^B(u_1, \dots, u_k) = 1 \text{ for every concrete individuals } u_1, \dots, u_k \text{ that are mapped to } u'_1, \dots, u'_k \text{ respectively} \\ 0 & \text{if } p^B(u_1, \dots, u_k) = 0 \text{ for every concrete individuals } u_1, \dots, u_k \text{ that are mapped to } u'_1, \dots, u'_k \text{ respectively} \\ 1/2 & \text{otherwise} \end{cases}$$

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, be we won't get contradictions):

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

(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:

$$p^S(u'_1, \dots, u'_k) = \sqcup \{p^B(u_1, \dots, u_k) \mid f(u_1) = u'_1, \dots, f(u_k) = u'_k\}$$

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:

$$\text{sm}(u') = \begin{cases} 0 & \text{if only a single individual } u \text{ is mapped to } u' \\ 1/2 & \text{if two or more individuals are mapped to } u' \end{cases}$$

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.

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 one-way-reachability property:

$$\mathrm{owr}[n](u_1,u_2) = [\![n^+(v,w) \wedge \neg n^+(w,v)]\!]_{[v\to u_1, w\to u_2]}$$

will be defined in the abstract structure as:

$$\mathrm{owr}^S(u_1',u_2') = \sqcup\{\mathrm{owr}[n_B](u_1,u_2) |\, f(u_1)=u_1', f(u_2)=u_2'\}$$

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^+, and evaluated in the 3-valued logic on an assignment Z in the following way:

  $$[\![(\mathrm{TC}\ v_1,v_2{:}\,\varphi)(v_3,v_4)]\!]_Z = \max_{\substack{n\geq 1, u_1,\ldots,u_{n+1}\in U,\\ Z(v_3)=u_1, Z(v_4)=u_{n+1}}} \min_{1\leq i\leq n} \left([\![\varphi]\!]_{Z[v_1\to u_i, v_2\to u_{i+1}]}\right)$$

  For example, a formula like $\varphi \equiv \neg n(v_1,v_2)$ can be used to create the TC formula:

  $$\varphi^+ \equiv (\mathrm{TC}\ v_1,v_2{:}\,\varphi)(v_3,v_4) \equiv (\neg n)^+(v_3,v_4)$$

  and $[\![\varphi^+]\!]_{[v_3\to u, v_4\to w]}$ would be evaluated as expected from a transitive operation:
  - 1 – if the there is a sequence of individuals $u = u_1, u_2, \ldots, u_{n+1} = w$ such that for every following individuals u_i, u_{i+1} the evaluation of $[\![\varphi]\!]_{Z[v_1\to u_i, v_2\to u_{i+1}]}$ is 1. (for this sequence, the min is 1, so the max min is 1)
  - 0 – if for any sequence of individuals $u = u_1, u_2, \ldots, u_{n+1} = w$ there is at least one couple of following individuals u_i, u_{i+1} that for them the evaluation of $[\![\varphi]\!]_{Z[v_1\to u_i, v_2\to u_{i+1}]}$ is 0. (for every sequence the min is 0, so the max min is 0).
  - ½ - otherwise

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

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

Or in other words:

$$\mathrm{owr}[n_B](u_1,u_2) \sqsubseteq \mathrm{owr}[n_S]\big(f(u_1),f(u_2)\big)$$
$$\mathrm{owr}[n_B](u_1,u_2) \sqsubseteq \mathrm{owr}[n_S](u_1',u_2') \quad \text{where } f(u_1)=u_1', f(u_2)=u_2'$$

Therefore, from the definition of the "least upper bound":

$$\mathrm{owr}^S(u_1',u_2') = \sqcup\{\mathrm{owr}[n_B](u_1,u_2) |\, f(u_1)=u_1', f(u_2)=u_2'\} \sqsubseteq \mathrm{owr}[n_S](u_1',u_2')$$

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:

1. $[\![v = w]\!]_{[v \to u_1, w \to u_2]} = 0$
2. $[\![v = w]\!]_{[v \to f(u_1), w \to f(u_2)]} = 1$

In order to fix this, such formulas are translated to the abstract world as:

$$(v = w) \wedge \neg \mathrm{sm}(v)$$

1. 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.
2. 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.
3. If a summary individual is compared to itself the result will be ½, which means that we don't know what result will be in the concrete structure.

Proof of Embedding Theorem

Let's look at a structure B which can be embedded into a structure S by a surjective $f: U^B \to U^S$ (denote as $B \sqsubseteq^f S$), and let φ be some formula with the free variables $v_1, \ldots, v_k$. 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.

We will prove by induction that for any assignment Z of the free variables $v_1, \ldots, v_k$ to some individuals $u_1, \ldots, u_k \in U^B$ (respectively) we get $[\![\varphi]\!]_Z \sqsubseteq [\![\varphi']\!]_{f \circ Z}$, where:

1. $f \circ Z$ is the assignment that maps the free variables $v_1, \ldots, v_k$ to $f(u_1), \ldots, f(u_k)$ (respectively)
2. φ' is the same formula as φ, except for the terms of the form $(v_i = v_j)$ which are replaced with $(v_i = v_j) \wedge \neg \mathrm{sm}(v_i)$

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 $[\![\varphi]\!]_Z \sqsubseteq [\![\varphi']\!]_{f \circ Z}$, we have to show that:

1. If $[\![\varphi']\!]_{f \circ Z} = 1$ then $[\![\varphi]\!]_Z = 1$
2. If $[\![\varphi']\!]_{f \circ Z} = 0$ then $[\![\varphi]\!]_Z = 0$

3. If $[\![\varphi']\!]_{f\circ Z} = ½$ the claim is true (nothing to prove in this case).

Basis

φ is an atomic formula. It is either a formula in the form $(v_1 = v_2)$ or a formula that evaluates some k-ary property. In the first case we get:

1. If $[\![\varphi']\!]_{f\circ Z} = 1$ then $[\![(v_1 = v_2) \wedge \neg \mathrm{sm}(v_1)]\!]_{f\circ Z} = 1$. According to the $f \circ Z$ assignment we get that $f(u_1) = f(u_2)$ and $\mathrm{sm}\big(f(u_1)\big) = 0$. From the definition of sm we get that only a single individual is mapped to $f(u_1)$, which means that $u_1 = u_2$. Therefore $[\![\varphi]\!]_Z = [\![(v_1 = v_2)]\!]_Z = 1$.
2. If $[\![\varphi']\!]_{f\circ Z} = 0$ then $[\![(v_1 = v_2) \wedge \neg \mathrm{sm}(v_1)]\!]_{f\circ Z} = 0$. According to the $f \circ Z$ assignment and the 3-valued "and", we get that either $f(u_1) \neq f(u_2)$ or $\mathrm{sm}\big(f(u_1)\big) = 1$. sm can only return 0 or ½, so the second case cannot happen. Therefore $f(u_1) \neq f(u_2)$ and because f is a function, we get $u_1 \neq u_2$. Finally, we get $[\![\varphi]\!]_Z = [\![(v_1 = v_2)]\!]_Z = 0$.

In the second case, φ is a formula that evaluates a k-ary property $p^B \in P^B$ that has a corresponding k-ary property $p^S \in P^S$, which is the evaluated property when we look at φ in S. Because $B \sqsubseteq^f S$, we get:

$$[\![\varphi]\!]_Z = p^B(u_1, \dots, u_k) \sqsubseteq p^S\big(f(u_1), \dots, f(u_k)\big) = [\![\varphi']\!]_{f\circ Z}$$

Induction Hypothesis

$$[\![\varphi]\!]_Z \sqsubseteq [\![\varphi']\!]_{f\circ Z}$$

Induction Step

φ is constructed of smaller formulas in one of the following ways:

1. $\varphi \equiv \varphi_1 \wedge \varphi_2$
   - $[\![\varphi']\!]_{f\circ Z} = [\![\varphi_1' \wedge \varphi_2']\!]_{f\circ Z} = 1$ means that both $[\![\varphi_1']\!]_{f\circ Z} = 1$ and $[\![\varphi_2']\!]_{f\circ Z} = 1$. By the induction hypothesis, $[\![\varphi_1]\!]_Z = 1$ and $[\![\varphi_2]\!]_Z = 1$, therefore $[\![\varphi]\!]_Z = [\![\varphi_1 \wedge \varphi_2]\!]_Z = 1$.
   - $[\![\varphi']\!]_{f\circ Z} = [\![\varphi_1' \wedge \varphi_2']\!]_{f\circ Z} = 0$ means that $[\![\varphi_1']\!]_{f\circ Z} = 0$ or $[\![\varphi_2']\!]_{f\circ Z} = 0$. WLOG, let's assume the first one happens. By the induction hypothesis $[\![\varphi_1]\!]_Z = 0$, therefore $[\![\varphi]\!]_Z = [\![\varphi_1 \wedge \varphi_2]\!]_Z = 0$.
2. $\varphi \equiv \neg\varphi_1$
   - $[\![\varphi']\!]_{f\circ Z} = [\![\neg\varphi_1']\!]_{f\circ Z} = 1$ means that $[\![\varphi_1']\!]_{f\circ Z} = 0$. By the induction hypothesis $[\![\varphi_1]\!]_Z = 0$ which means that $[\![\varphi]\!]_Z = [\![\neg\varphi_1]\!]_Z = 1$.
   - $[\![\varphi']\!]_{f\circ Z} = [\![\neg\varphi_1']\!]_{f\circ Z} = 0$ means that $[\![\varphi_1']\!]_{f\circ Z} = 1$. By the induction hypothesis $[\![\varphi_1]\!]_Z = 1$ which means that $[\![\varphi]\!]_Z = [\![\neg\varphi_1]\!]_Z = 0$.
3. $\varphi \equiv \exists v\colon \varphi_1$, where φ_1 contains the free variables: $v_1, \dots, v_k, v$
   - $[\![\varphi']\!]_{f\circ Z} = [\![\exists v\colon \varphi_1']\!]_{f\circ Z} = 1$ means that there is an individual $u' \in U^S$ such that $[\![\varphi_1']\!]_{(f\circ Z)[v\to u']} = 1$. Because f is surjective, there must be some $u \in U^B$ such that $f(u) = u'$, therefore $[\![\varphi_1']\!]_{f\circ(Z[v\to u])} = 1$. By using the induction

hypothesis we get $[\![\varphi_1]\!]_{Z[v\to u]} = 1$. Finally, we can see that this formula proofs that $[\![\varphi]\!]_Z = [\![\exists v\colon \varphi_1]\!]_Z = 1$.

- $[\![\varphi']\!]_{f\circ Z} = [\![\exists v\colon \varphi_1']\!]_{f\circ Z} = 0$ means that for any individual $u' \in U^S$ we get $[\![\varphi_1']\!]_{(f\circ Z)[v\to u']} = 0$. This is also true when $u' = f(u)$. Thus, for any $u \in U^B$ we get $[\![\varphi_1']\!]_{f\circ(Z[v\to u])} = 0$. By using the induction hypothesis we get that for any $u \in U^B$, $[\![\varphi_1]\!]_{Z[v\to u]} = 0$. Finally, we can see that this proofs that $[\![\varphi]\!]_Z = [\![\exists v\colon \varphi_1]\!]_Z = 0$.

4. $\varphi \equiv (TC\ v_1, v_2\colon \varphi_1)(v_3, v_4)$
   - $[\![\varphi']\!]_{f\circ Z} = [\![(TC\ v_1, v_2\colon \varphi_1')(v_3, v_4)]\!]_{f\circ Z} = 1$ means that there's a sequence of individuals $u_1', \dots, u_{n+1}' \in U^S$ such that the assignment $f \circ Z$ maps $v_3 \to u_1'$ and $v_4 \to u_{n+1}'$ and for every following individuals u_i', u_{i+1}' we have $[\![\varphi_1']\!]_{(f\circ Z)[v_1\to u_i', v_2\to u_{i+1}']} = 1$. Because f is surjective, we can find for that sequence a sequence $u_1, \dots, u_{n+1} \in U^B$ in which:
   $$u_1' = f(u_1), \dots, u_{n+1}' = f(u_{n+1})$$
   In the sequence that we choose, we can specifically choose $u_1 = Z(v_3)$ and $u_{n+1} = Z(v_4)$.
   For every following individuals u_i, u_{i+1} we get:
   $$[\![\varphi_1']\!]_{f\circ(Z[v_1\to u_i, v_2\to u_{i+1}])} = [\![\varphi_1']\!]_{(f\circ Z)[v_1\to f(u_i), v_2\to f(u_{i+1})]} = 1$$
   And from the induction hypothesis we get:
   $$[\![\varphi_1]\!]_{Z[v_1\to u_i, v_2\to u_{i+1}]} = 1$$
   We found a sequence which gives us:
   $$\min_{1\le i\le n}\left([\![\varphi_1]\!]_{Z[v_1\to u_i, v_2\to u_{i+1}]}\right) = 1$$
   Finally we get:
   $$[\![\varphi]\!]_Z = [\![(\mathrm{TC}\ v_1, v_2\colon \varphi_1)(v_3, v_4)]\!]_Z = \max_{\substack{n\ge 1, u_1,\dots,u_{n+1}\in U^B,\\ Z(v_3)=u_1, Z(v_4)=u_{n+1}}} \min_{1\le i\le n}(\dots) = 1$$
   - $[\![\varphi']\!]_{f\circ Z} = [\![(TC\ v_1, v_2\colon \varphi_1')(v_3, v_4)]\!]_{f\circ Z} = 0$ means that for every sequence of individuals $u_1', \dots, u_{n+1}' \in U^S$ such that the assignment $f \circ Z$ maps $v_3 \to u_1'$ and $v_4 \to u_{n+1}'$ there is at least one couple of following individuals u_i', u_{i+1}' which give: $[\![\varphi_1']\!]_{(f\circ Z)[v_1\to u_i', v_2\to u_{i+1}']} = 0$. This is particularly true for any $u_1, \dots, u_{n+1} \in U^B$ where $u_1 = Z(v_3)$ and $u_{n+1} = Z(v_4)$, and the derived sequence:
   $$u_1' = f(u_1), \dots, u_{n+1}' = f(u_{n+1})$$
   meaning that for some $1 \le \mathrm{i} \le \mathrm{n}$:
   $$[\![\varphi_1']\!]_{f\circ(Z[v_1\to u_i, v_2\to u_{i+1}])} = [\![\varphi_1']\!]_{(f\circ Z)[v_1\to f(u_i), v_2\to f(u_{i+1})]} = 0$$
   By the induction hypothesis, we get for that i:
   $$[\![\varphi_1]\!]_{Z[v_1\to u_i, v_2\to u_{i+1}]} = 0$$
   Therefore:
   $$\min_{1\le i\le n}\left([\![\varphi_1]\!]_{Z[v_1\to u_i, v_2\to u_{i+1}]}\right) = 0$$
   Finally we get:

$$[\![\varphi]\!]_Z = [\![(\mathrm{TC}\, v_1, v_2 : \varphi_1)(v_3, v_4)]\!]_Z = \max_{\substack{n\geq 1, u_1,\ldots,u_{n+1}\in U^B, \\ Z(v_3)=u_1, Z(v_4)=u_{n+1}}} \min_{1\leq i\leq n} (\ldots) = 0$$

The proof of the induction proves the Embedding Theorem.

☑

Transformers

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).

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 $B_{\mathrm{before}} \sqsubseteq^f S_{\mathrm{before}}$ (f maps the concrete individuals to their abstract individuals) then evaluations of such a formula φ will obey:

$$[\![\varphi]\!]_Z^{B_{\mathrm{before}}} \sqsubseteq [\![\varphi']\!]_{f\circ Z}^{S_{\mathrm{before}}}$$

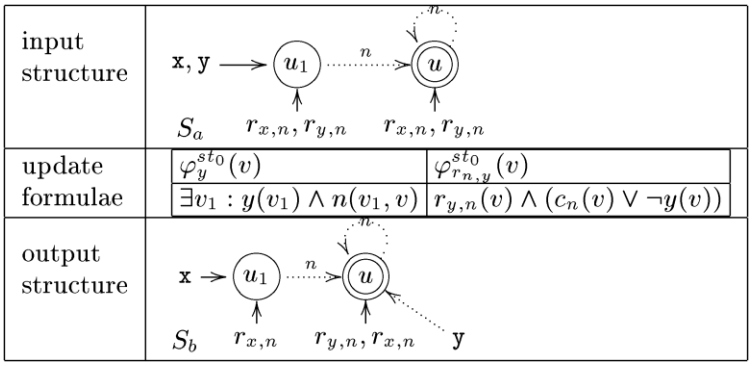
Therefore, every nullary, unary, binary or k-ary property $p(u_1, \dots, u_k)$ will be preserved:

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

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 ½.

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.

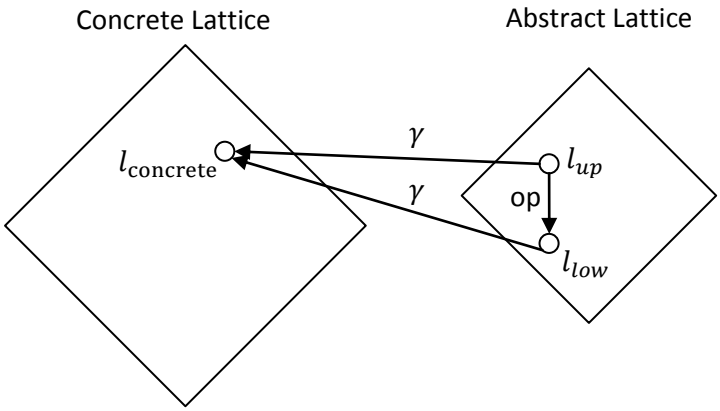


As we can see, we lost the information about the node that is pointed by y (we can still conclude that it is not null because there are reachable nodes).

Focusing

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 ½), 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



Example for a Semantic Reduction operation "op":

$\mathbf{op}(l_{up}) = l_{low} \sqsubseteq l_{up}$ and $\gamma(l_{up}) = \gamma(l_{low})$

embedded into the original structure. Therefore, the original abstract structure and the collection of new abstract structures represent the same set of concrete structures.

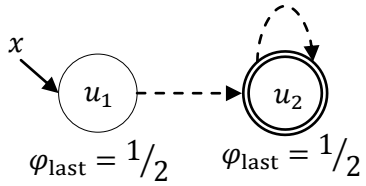
This technique is called **Focusing**. The idea of taking an element in our lattice (the original abstract structure) and converting it to a more precise element (the collection of new abstract structures), in a way that preserves their transformation to the concrete world, is called **Semantic Reduction**.

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.

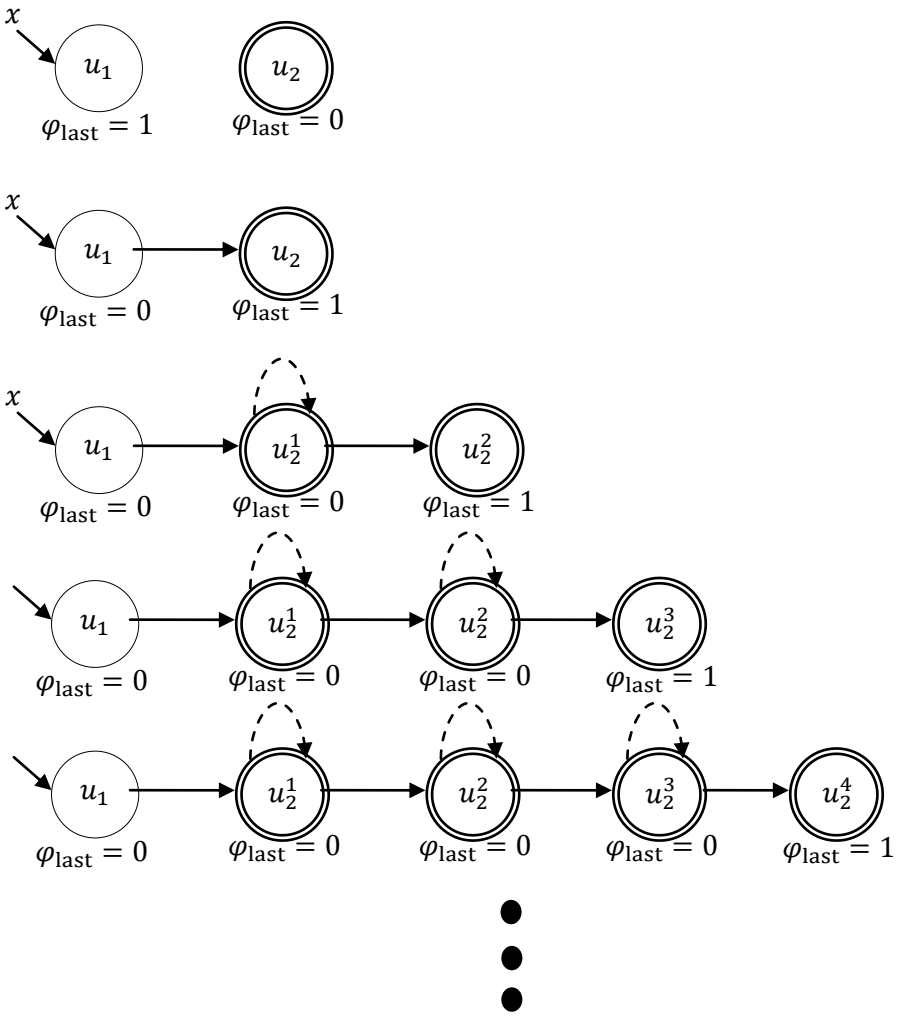
Not all formulas can be used in the focusing process. For example, let's look at the formula:

$$\varphi_{\text{last}}(v) \stackrel{\text{def}}{=} \forall v_1 \colon \neg n(v, v_1)$$

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_1 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 ½'s, and therefore it will have an indecisive $\varphi_{\text{last}} = {}^1/_2$. 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 Table XI. The idea behind this table is:

1. 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 $\exists v_1 : t(v_1) \wedge n(v_1, v)$.

2. 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 $x(v)$.

Table XI. The Target Formulae for *Focus*, for Statements and Conditions of a Program that Uses Type `List`

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`	$\{x(v), t(v)\}$

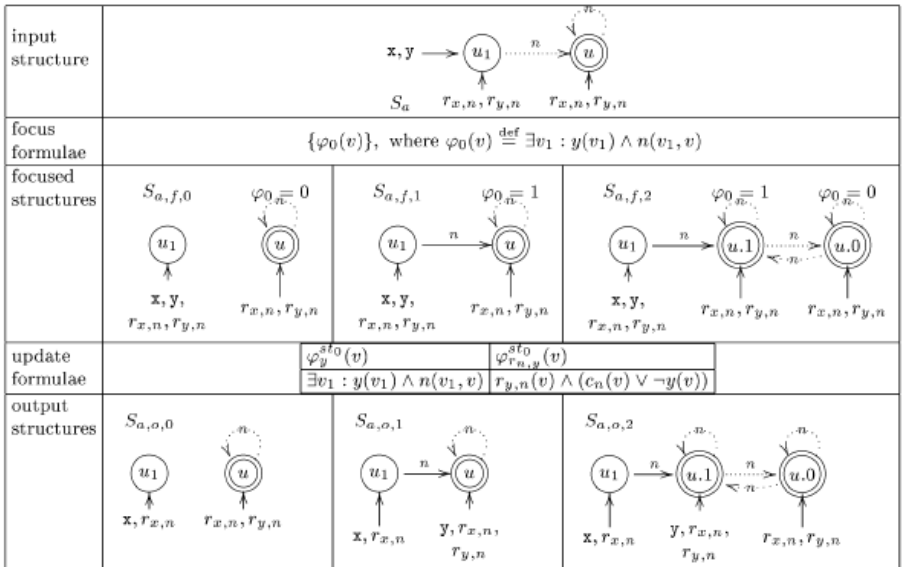
We can see that our focus formulas ask questions of: "Is there a route of length <specific length> from the stack pointer <specific stack pointer> to the node?" – For example, in the case of reading $t \to n \to n$, the focus formula $\varphi(v) \stackrel{\text{def}}{=} \exists v_1 \exists v_2 : t(v_1) \wedge n(v_1, v_2) \wedge n(v_2, v)$ asks the question: "Is there a route of length 3 from the pointer t 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:

1. 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.
2. 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.
3. 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.

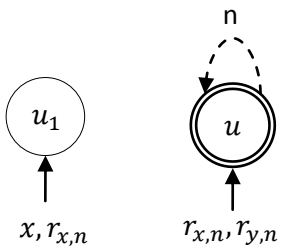
For example, in the statement $y = y \to n$, where x, y point to the first node of a linked list, we get (the r property is the reachability):



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.

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 constrains 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 $p(u_1, \dots u_k)$ equals to an assignment in some FO^{TC} formula: $[\![\varphi]\!]_{[v_1 \to u_1, \dots, v_k \to u_k]}$ (from definition or from a known constraint) then in any abstract structure, $p(u'_1, \dots u'_k)$ should be at least as precise as the evaluation of $[\![\varphi]\!]_{[v_1 \to u'_1, \dots, v_k \to u'_k]}$. Furthermore, if $p(u'_1, \dots u'_k)$ has a definite value and $[\![\varphi]\!]_{[v_1 \to u'_1, \dots, v_k \to u'_k]}$ has an incomparable definite value, then the abstract structure does not represent any concrete structure at all.

Some examples for known constraints:

1. $x(v_1) \wedge x(v_2) \to \mathrm{eq}(v_1, v_2)$
2. $x(v_1) \to \neg \mathrm{sm}(v_1)$
3. $n(v_1, v) \to \neg \mathrm{sm}(v)$
4. $n(v, v_1) \wedge n(v, v_2) \to eq(v_1, v_2)$
5. $n(v_1, v) \wedge n(v_2, v) \wedge \neg eq(v_1, v_2) \leftrightarrow is(v)$
6. $n(v_1, v) \wedge \neg is(v) \wedge \neg eq(v_1, v_2) \leftrightarrow \neg n(v_2, v)$
7. $r_{y,n}(v) \leftrightarrow y(v_1) \wedge n^*(v_1, v)$

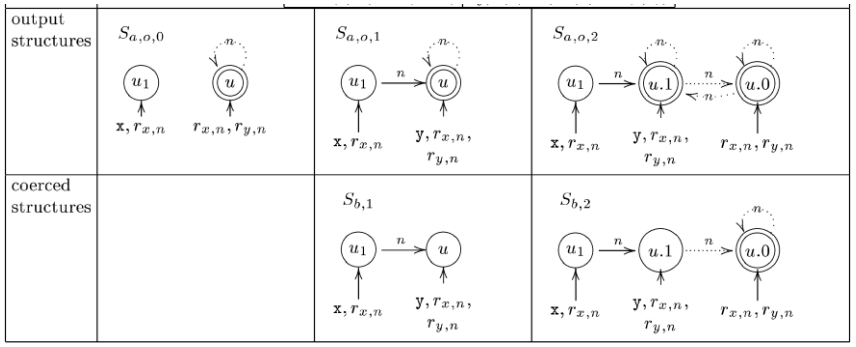
In our last example, assigning $v \to u$ in the 7th constraint with $v_1 \to u_1$ results in two incomparable definite values:

$$r_{y,n}(u) = 1$$

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

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 $v_1 \to u_1$, 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 6th constraint, with $v \to u, v_1 \to u_1, v_2 \to u$ in the central case ($S_{a,o,1}$), results in removing the indefinite self edge of u ($\neg n(u,u)$). The 6th constraint can also be used in the right case ($S_{a,o,2}$), with $v \to u.1, v_1 \to u_1$, to remove all the indefinite edges towards $u.1$: ($v_2 \to u.1$ for $\neg n(u.1, u.1)$ and $v_2 \to u.0$ for $\neg n(u.0, u.1)$.



In the following "list-insertion" example (Table X), we can see the big accuracy difference in the final states between the Kleene (Strawman) simplistic analysis, and the refined analysis that uses Focusing and Coercion.

Table X.
Selective applications of the abstract transformers using the strawman and refined approaches, for statements in `insert` that come after the search loop (for brevity, r_z is used in place of $r_{z,n}$ for all variables z, and node names are not shown).