Modification Shapes

Laurent Mauborgne

January 30, 2012

1 Concrete Semantics: Heaps and Procedures

1.1 Most Concrete Semantics and First Uniform Abstraction

In the most concrete semantics, a program execution is a sequence of states, and those states are arrays of numbers. In general, this low-level view of states is too complex, so we abstract it into a stack containing values of program variables and a heap. The heap itself is abstracted as a set of disjoint memory locations, and accesses outside of those locations are abstracted by a total loss of information (including error state). In that setting, the value of a variable can be either a number, a valid location, or the special location **null**which does not correspond to any valid location. A location itself is an array of values. These different values can be refered to either by field names or by numbers. To simplify, we will suppose here that references inside a location don't overlap (see Min's work on union types for a more general case). It is interesting to have a uniform representation of the states of a program by considering that the memory location containing the global variables is a location and the global variables the fields of that location, and the stack is a set of locations where the local names are fields of that locations, and with two additional fields, one containing the location of the next element of the stack and the other one the previous element. The basic assumption is that one field name correspond to exactly one value inside a location, so scoping mecanism such as function calls or block will correspond to creating a new stack location.

TODO: Does that need to be more formal? So that the abstraction if more precisely defined?

1.2 Formal Definitions and Notations

We define Loc a set of memory locations, $Fields$ a set of names and BV a set of basic values such as integers plus floats plus booleans...

A *memory state* is a partial function that maps each couple of memory location and field name to either a memory location, a basic value, or the special value **null**. **null** corresponds to a fictive memory location that cannot be in the domain of a memory state and stands for an unallocated memory location. We write $MS = Loc \times Fields \to Loc \cup BV \cup \{\mathbf{null}\}$ for the set of all possible memory states. We suppose that *Loc* contains at least a special location *Glob* containing the global variables and a special field *Stack* containing the location of the top of the call stack.

A *path* will be a possibly empty sequence of names.

1.3 Program Executions

A program will be composed of procedures. Each procedure is given by a name, a set of arguments, local variables and a sequence of instructions. Each instruction is given a unique label.

A run starting in a given memory state M for a procedure consists in the sequence of memory states starting with M and following the flow of instructions of that procedure.

The concrete semantics of a program with a given starting procedure is the set of all runs from any valid memory state for that procedure. The notion of "valid" memory state might depend on a type attached to each memory location.

2 Shape Graphs

In order to represent sets of memory states, we will summarize different locations. One way to achieve that is to group such locations according to access path, using a graph labeled by field names.

2.1 Notations on graphs

We will manipulate labeled graphs, which are in general defined by a triple $\langle \mathcal{N}, \mathcal{L}, \mathcal{E} \rangle$ such that $\mathcal{N}$ is the set of nodes of the graph, $\mathcal{L}$ is the set of edge labels and $\mathcal{E}$, a subset of $\mathcal{N} \times \mathcal{N} \times \mathcal{L}$, is the set of edges of the graph.

If p is a sequence of elements of $\mathcal{L}$ and n a node in a graph $\langle \mathcal{N}, \mathcal{L}, \mathcal{E} \rangle$ then $n \star p$ is the set of nodes reachable from n following the path p and is formally defined by $n \star \epsilon = \{n\}$ and $n \star pf = \{\, m \mid \exists n' \in n \star p, \langle n', m, f \rangle \in \mathcal{E}\}$.

2.2 The Classic Case

Classically, a *shape graph* represents a set of memory states. A shape graph is simply a graph labeled by field names. The set of memory states represented by a shape graph is defined by means of the *embedding* relation wich ensures that no path of the memory state is forgotten in the shape graph. This is achieved by requiring the existence of a function mapping memory states to nodes.

To represent complex shapes and more precisely abstract sets of memory states, we need more information. Such informations are added to the shape graphs in the form of properties attached to sets of nodes, so that now a shape graph $\mathcal{G}$ is a set of nodes $\mathcal{G}_N$, a set of edges (subset of $\mathcal{G}_N \times \mathcal{G}_N \times Fields$) and a set of properties $\mathcal{G}_\pi$ (element of $\wp(\mathcal{G}_N) \to \wp(MS)$). Now we can define formaly the embedding relation:

Definition 1 *A memory state m is embedded in a shape graph $\mathcal{G}$ (and we write $m \tilde{\in} \mathcal{G}$) if there is a mapping ϕ from Loc to nodes of $\mathcal{G}$ such that*

- *for all couple (l, f) in the domain of m, if $m(l, f)$ is a location then $\langle(\phi(l), \phi(m(l, f), f\rangle$ is an edge of the abstract graph,*
- *and for all $N \subset \mathcal{G}_N$ in the domain of $\mathcal{G}_\pi$, m restricted to $\phi^{-1}(N)$ is in $\mathcal{G}_\pi(N)$.*

In general it is inefficient to associate properties to each subset of nodes in the graph. It is important to keep things local as much as possible. So, we will nearly always attach properties to single nodes, sometimes to edges in the shape graphs and to the entire graph for field values, but as little as possible for the other subsets of graph nodes.

2.3 Reachability

Examples of properties include abstractions of the number of elements (usually 0, 1 or many), whether the elements are in a cycle, or if the elements are reachable from a given node.

Given n and m two nodes and F a set of edge labels, we will note $\mathcal{R}_F n(m)$ to describe that a location m either is n or is the image of (o, f) with $f \in F$ and $\mathcal{R}_F n(o)$, this means that n is reachable from m by following edges with labels in F. Formally, we write that this property is true for a given memory state M in the following way (least fixed point interpretation):

$$M \models \mathcal{R}_F n(m) \Leftrightarrow n = m \vee \exists f \in F, \text{ such that } M \models \mathcal{R}_F M(n \star f)(m)$$

n and m may be omitted when there is no possible confusion.

Another useful notion is reachability *avoiding* a set of nodes L. That can be expressed as

$$M \models \mathcal{R}_F n_{\setminus L}(m) \Leftrightarrow n = m \vee \exists f \in F, \text{ such that } \big(M(n, f) \notin L \wedge M \models \mathcal{R}_f M(n, f)_{\setminus L}(m)\big)$$

TODO: See how this can be lifted to sets of labels

2.4 Value Domains

The properties attached to each node and each edge in the shape graphs are intended to provide shape information which cannot be described by a graph alone. In addition, most analyses will need to track the value of the fields in the memory location. This can be done by reusing existing abstract domains, considering that each field of each node in the shape graph correspond to a variable. That allows for a finite number of variables for the value abstract domains. Then we have to take care that if a node is of possible cardinality greater than 1, then all assignments to a field of that node is a may-assign.

On the concrete side, the abstract value associated with a shape graph restricts the memory states that can be embedded into that shape graphs to those states such that the field values are in the concretization of the abstract value.

2.5 Modification Information

Shape graphs are not enough to represent modifications as they just represent sets of memory states. I see two ways of augmenting shape graphs to include that information: we can use one or 2 shape graphs to represent couples of shape graphs, or we can instrument the concrete semantics with modification tags.

In order to analyse different contexts for all procedures, the second solution requires that we associate with each node and for each context a modification information in the concrete. This information can then be abstracted to various degrees in the abstract shape graphs.

The first solution seems preferable, as it lets the issue of context approximation to another possibly separate abstraction. It might be the case that it is less precise, though. I will describe the property of modification in that context to start with:

The property $\mathcal{M}_f n$ is true if (n, f) is defined in one memory state but not in the other or if the value associated with (n, f) is different from the one associated to (n', f). Formally, if M and M' are memory states,

$$(M, M') \models \mathcal{M}_f n \Leftrightarrow M(n, f) \neq M'(n, f)$$

Note: notice that this definition is symmetrical and maybe not what we would have for the usual meaning of modified. In particular, a variable that is modified but which is reset to its initial value at the end of the procedure will not be seen as modified by that property. If you want it to be deemed modified, you need to represent not just pairs of states but sequences. That is not unfeasible, but do we need it?

In order to introduce the modification in the shape graph we extend the concretization to pairs of memory states. That means that instead of having local invariants associated to each locations in the program, we will have relational invariants associated with couples of memory locations. We don't need to associate explicitly such an invariant with all couples. If the two elements of the couple are the same, then we are back to classical shape graphs.

The concretization of a shape graph $\mathcal{G}$ with modification informations is the set of couple of memory states (M, M') such that M' is embedded in the shape graph without modification properties and M is any memory state such that together with M' respects the modification properties. Note that modification properties can typically be expressed in terms of the $\mathcal{M}$ property or be any property constraining a couple of memory states. In general also, the memory state M will be constrained by the invariant at its memory location. We could have more precise results if we added some way of referring to the shape graph at that location.

2.6 Properties and Sub-Structures

One restriction of classic shape graphs is that properties are local to abstract nodes or abstract edges. The addition of precise modification informations might require to split such nodes, with the possible consequence of losing information about other properties. In our setting, It is possible to add properties to sets of abstract nodes, but we will have to be very careful with them, as locality is one of the main sources of efficiency in shape graphs.

Such addition would also be useful in representations of overlapping structures in order not to lose information when modifying one of the structures.

TODO: link with region logics

3 Transfer Functions

To design precise transfer functions we need to define focus and normalizing functions. A focus will split abstract nodes or edges to yield more precise results, while still representing the same set of (couple of) memory states.

For each function the formal idea will be explained. When it may improve readability, we will show its effects on a particular case : case 3 where the property associated to nodes and edges is about their cardinality and may take three values :

- cardinality 1
- cardinality either 1 or 0
- undetermined cardinality.

3.1 Focus

This operation is also called rearrangement or materialization in related works. Focusing is relative to a path (sequence of field names) from a stack variable. The goal of focusing is to provide a set of shape graphs representing the same set of memory states as the original shape graph, but such that in each shape graph the node referred to by the path is distinguished from the rest of the heap. This is meaningful only if we have properties for the number of locations that can be embedded in a node.

The process of focusing will consist in following the path. That is to focus on path $p.f$ we first focus on p then $p.f$ on the resulting shape graphs. Focus on the empty path is identity, as the empty path refers to the stack which is always one node. To focus on $p.f$ assuming p is focused, for each node n in the set of nodes $stack \star p.f$, if the cardinality property of n can be different from 1, then we can split it. Splitting consists in creating at least two nodes n_0, n_1, …such that the cardinality of n_0 contains the value 1 but no greater value and the abstract sum of the cardinalities of the n_k contains the cardinality of n. Then, in the resulting graph, for each node in $stack \star p.f$, we create a graph such that this node is the only one in $stack \star p.f$, and the union of all the graphs is a superset of the initial graph.

In the 3 case, the problem that may be raised are linked to the two possibly-non-one-cardinality cases. First, in the case of a zero-or-one cardinality node in

$stack \star p.f$, two nodes are created n_0 of cardinality one and n_1 of cardinality zero. In that case, the edge labelled f is removed. The represented node may still though exist if there were other existing edges going to or from it. Second, in the case where there is no information about the cardinality of the node, two graphs may be deduced for each node in $stack \star p.f$, corresponding to n_0 of cardinality one and n_1 without any property over cardinality. Both graphs should be straightforward once the previous case has been explained.

3.2 Assignment

Let us consider an instruction $p.f = r$ where p is a possibly empty path. To abstract such an assignment, if we don't care about cardinality, we can just add edges labeled f from all nodes in $p.f$ to all nodes in r, but that would not be very precise. We can do much better by first focusing on p. Then we know that in each shape graph resulting from that focus we can first remove the edges labeled by f and starting from p and then add edges labeled f of cardinality at most 1 from p to each node in r. We can be even more precise by focusing on r, so that we can add the constraint that the cardinality on the new edge is exactly 1. Another consequence of that second focus is that is keeps track of aliasing informations. TODO: See if that is useful to do.

3.3 Binary Operations: Graphs Unification

A standard way of performing binary operations is to start with graphs unifications, such that both graphs differ only on the properties on edges and nodes, and then we apply the binary operation to the properties. The operation is recursive with memoization on pairs, so that we don't loop. Starting from the heap, for each pair of nodes, we do the following: if at least one edge labeled f is present from one node and not the other, then we can add edges of cardinality 0 to new nodes of cardinality 0 to the second node. If we have multiple edges with the same label then we have a choice, with a cost/precision trade-off: either we examine all possible combinations, meaning node creations in at least one graph, or we merge the nodes by performing unions.

Using graph unification, it is easy to compute union, intersection or if one graphs subsumes the other.

3.4 widening

Widening is easier to define on two graphs $\mathcal{G}_1$ and $\mathcal{G}_2$ such that $\mathcal{G}_1 \leq \mathcal{G}_2$. Then $\mathcal{G}_2$ is an extension of $\mathcal{G}_1$. That means that there is an edge that is in $\mathcal{G}_2$ but not in $\mathcal{G}_1$ (otherwise they only differ on properties and we can apply widening on properties only).

4 Examples

4.1 The Composite Design Pattern

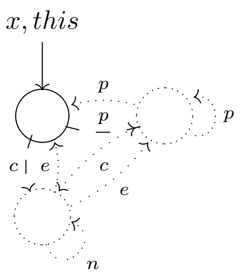
```
class Composite {
private Composite p;
private List c;
private int total=1;

void add(Composite x) {
c = new List(x,c);
x.p= this;
addToTotal(x.total);
}

private void addToTotal(int t) {
1
Composite x=this;
2
while(x != null){
3
x.total += t;
4
x = x.p;
5
}
6
}
}

class List {
Composite e;
List n;
}
```

The goal is to show that `c1.add(c2)` only modifies the region of the heap that is reachable through the `parent` node from node `c1`. One application would be to show that if `c1`, `c2` and `c3` point to separate heap regions then adding `c2` and `c3` to `c1` does not modify `c2`.

To illustrate the approach, we analyze the `addToTotal` procedure. We will draw shape graphs with the convention that dotted nodes and edges are associated with no cardinality property, plain ones with exactly one cardinality property, and dashed ones with cardinality zero or one. One may refer to case 3. For this example we will also suppose that we have no cycle through the p field in order to simplify the drawings. To this purpose, we also have chosen not to represent the `total` field.

Then, just before the loop, we have:

2$_1$

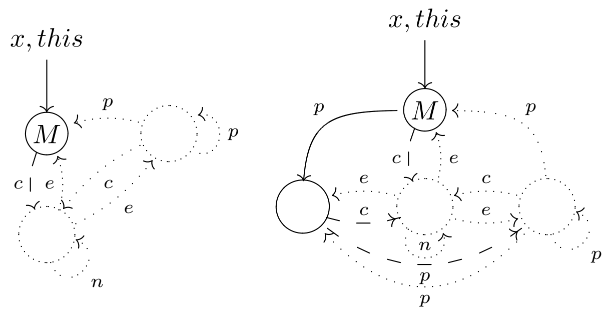


We have chosen to separate the two nodes representing `Composite` in order to represent the information on cardinality we have, that is why there are three nodes rather than only two.

Then we proceed to analyzing the loop. First the test `x != null` does not change the set of memory shapes we consider here, as `this` cannot be null. Then we modify `x.total`, so we add the property that `x.total` might have been modified. This means that `x` may have been modified, which is represented by M on the corresponding abstract node. This means that if a node n does not have an M label, then $\forall (m,l) \in Loc \times Fields$ such that $N(m,l) = n$, we have $\neg \mathcal{M}_f m$.

Finally for that first iteration, we perform `x = x.p`. If we don't focus on $x.p$, then notice that we loose the equality between x and $this.p$, as we could have $\mathtt{x} = \mathtt{this.c.e}$, this would result in a very imprecise analysis. So we first focus on $x.p$, which gives two shape graphs, depending on whether the cardinality of that edge is 0 or 1:

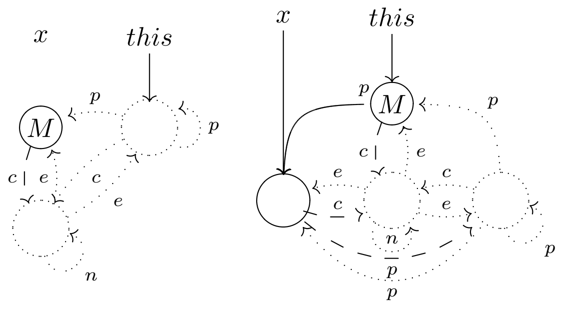
4$_1$



The first graph is easy. For the second one, we have at least one edge of cardinality greater than 0 going into the summary node, from which we can infer that it represents at least one node. From a node of cardinality one, edges cannot have cardinality greater than one, which is reflected on the graph.

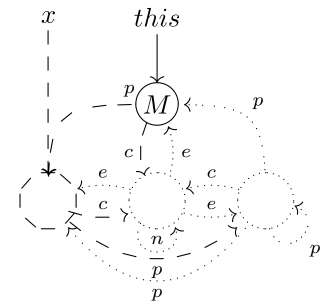
Then, on each graph, the assignment is easy enough, and we have at the end of the first iteration:

$\mathbf{5}_1$



Then we have a choice, either to take the union or keep a set of shape graphs. If we take the union, we first expand the first graph by virtually creating nodes and edges of cardinality 0. Then we get:

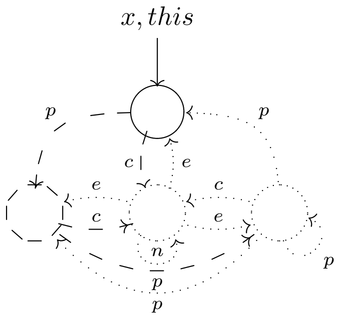
$\mathbf{5}_1$



Note that we lose the relationship between the cardinality of x and $self.p$. That would lead to imprecisions in the final result, but that information can be kept in many ways. One way is to have an equality domain dealing with location values of the fields. In that way, we keep that the value of x was the same as the value of $self.p$.

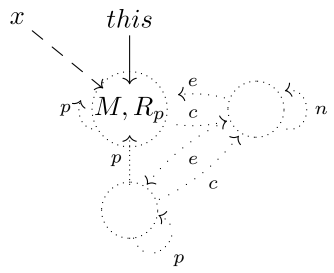
Then we have to perform a union or a widening between the incoming invariant at the head of the loop and the invariant after one loop iteration. Here, an early widening will be more precise. The widening will proceed as follows: first we match the two graphs, so that the incoming invariant becomes represented by

$\mathbf{2}_1$



Then we check for differences between the two graphs $\mathbf{2}_1$ and $\mathbf{5}_1$. We see that the only change is in the value of x, so we chose to merge those nodes in the invariant after iteration. We get

$\mathbf{2}_2$



Notice that on that new node, we have performed a bunch of computations:

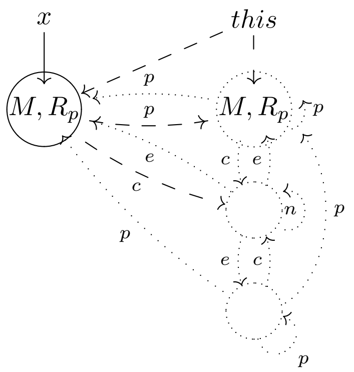
- cardinality is the abstract sum of of $[0, 1]$ and 1, which is top,
- outcoming nodes have been summed in the same way,
- modification is the union of the modification on both nodes,
- and most interestingly, we inferred a reachability predicate.

A reachability predicate can be inferred if we merge two nodes linked through an edge. Note that we must ensure that this edge is of cardinality 1 when the cardinality of the node is not 0, as the reachability information we track is a must be reachable. Then, we can also infer that no outgoing edge labeled by p is possible because we summarize all nodes reachable through p. We have omitted the parameters that can be easily infered from the context and this is represented through R_p.

After widening, we compute another iterate. First the guard, which will only change a dashed edge form x into a plain edge, expressing that this value cannot be null after the guard. Then the assignment which imposes focusing on x and $x.p$.

To focus on x, we first construct two different graphs depending on the cardinality of the M, R_p labelled node: either 1 or undetermined. When unifying the two graphs, we obtain:

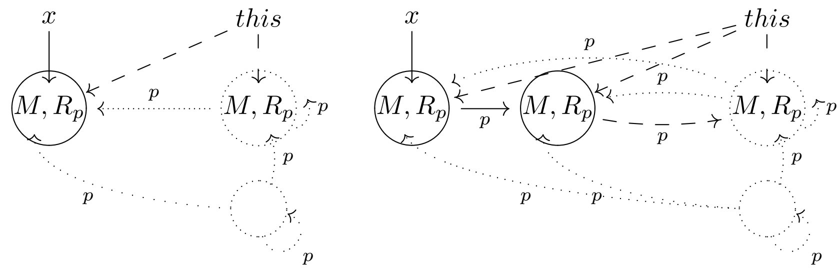
$\mathbf{4}_2$



Note that this shape graph loses the information that *this* is non null, but that can be kept as another node property. Also note that there can always be a c edge between a *Composite* node and the *List* node and an e edge in the other direction. So, to keep the drawings manageable we will subsequently assume they are there and not draw the *List* node.

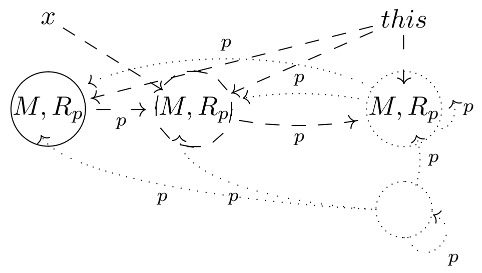
Focusing on $x.p$ gives two graphs, depending on whether that pointer is null:

$\mathbf{4}_2$



After performing the assignment in each case and computing the union, we get:

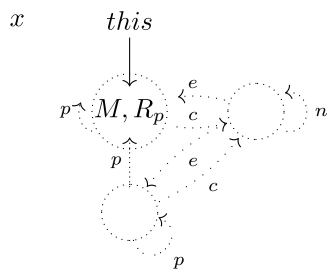
$\mathbf{5}_2$



Finally, we check that this shape graph represents a subset of the set obtained at the previous iteration represented at $\mathbf{5}_1$, meaning we reached a post fixpoint for the `while (x!= null)` loop.

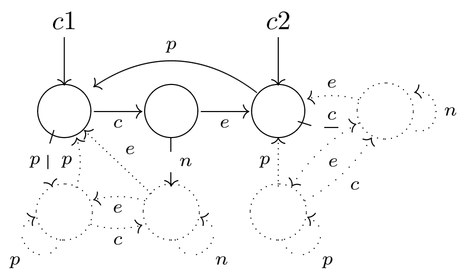
So, we have computed the following invariant at the end of the procedure, once we have exited the loop, meaning `x = null`:

$\mathbf{6}_1$

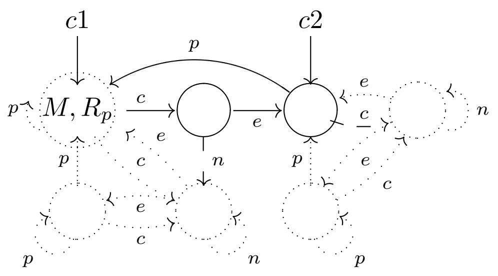


The information encoded in that invariant is that x is null and all the nodes where the total field have been modified are reachable through p pointers from *this*.

From that information about `addToTotal`, we can conclude for the original problem: if we compute `c1.add(c2)` with `c1` and `c2` in disjoint regions, then the resulting graph before calling `addToTotal` will be



Then the effect of `c1.addToTotal` will give, as expected,



Note that we could obtain a similar result with TVLA, although the output will be a set of shape graphs in that case, and probably result in much more complex analyses.

4.2 The Subject-Observer Design Pattern

```
class Node {
List o; // the nodes observing this
List s; // the subjects or nodes observed by this
int val;

// the subject interface
//
void register(Node x) {
o = new List(x, o);
x.notify();
}

int get(){
return val;
}
//the observer interface
//
void addSubject(Node x) {
s = new List(x, s);
x.register(this);
}

//We call this to notify that one of the subjects has changed
void notify() {
1
int temp = 1;
for(List l = s; l!=null; l = l.n)
temp = f(l.e.get(),temp); //f computes internal value
if(val != temp) {
val = temp;
2
List l = o;
3
while(l!=null){
4
l.e.notify();
5
l = l.n;
6
}
7
}
}
}

class List {
Node e;
List n;
}
```

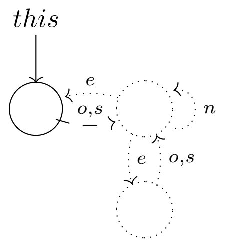
The subject/observer pattern can be seen as an extension of the composite pattern: to encode the composite into the subject/observer, we just restrict to one observer per node. In general, the `notify()` method will perform an arbitrary computation and update the internal state if necessary. The example we give here corresponds exactly to the behavior of the compound pattern if restricted to one observer per node and the fuction f is the sum.

The key procedure here is now `notify()`. For the analysis, the difference is not just more than one parent, it is the fact that the procedure mixes loops and recursive calls.

For the interprocedural analysis of this example, we will abstract the stack by its top element and its tail. We will use normal variable names for local variables at the top of the stack and primed variables for all those in the tail. The modification property is also a local property, so we will also need a primed version of that property. With the aim of clarity, we are not going to draw the `val` field.

To begin with, we suppose that the (implicit) parameter of `notify` is well formed and non null, and the stack tail is empty, so that we have:

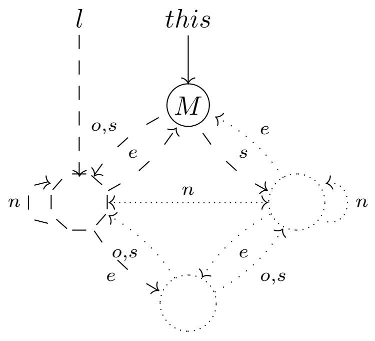
1$_1$



Notice that we did not make a lot of assumptions: a `Node` could observe itself, the subject list can be a sublist of the observers...

Then, not knowing what f does, except that it cannot modify the heap as it does not have any variable pointing to the heap, we must assume that the test `val!=temp` might be true and mark `this.val` possibly modified. Next step, we assign `this.o` to `l`. That means that we have to focus on *this.o*, then perform the assignment to l on each graph and take the union. This is similar to the previous example, so we just give the result:

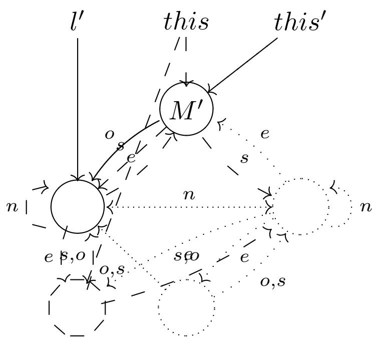
3$_1$



Then if we take the true branch of the test `l!=null`, and make the function call, we first have to focus on *l.e*.

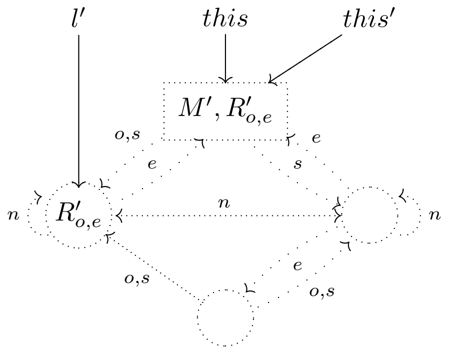
We then assign the primed values to the non primed value union their current value and assign `l.e` to `this`. The result is:

$\mathbf{4'}_1$



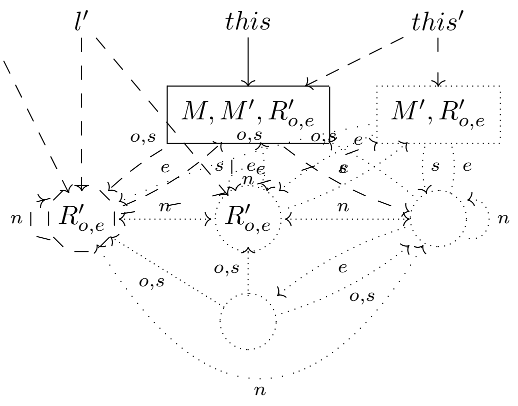
So we have a new value at the entry of the procedure, which we have to widen according to the previous value. As usual with widenings, we don't touch values that were empty in the previous iteration (all the primed variables here). So we just take care of `this` wich pointed to one node and now points to two which we merge and to which we add everything reachable through a repetition of the path between them, that is `oe`. In fact we will even be more abstract and include all path from `this` labelled with some `o` and `e` (but only those two labels). In the same way, we add to the node in the path. Note that the reachability predicate infered is reachability from $this'$. The result of the widening, giving the new state of the heap at the entry of the procedure, is:

$\mathbf{1}_2$



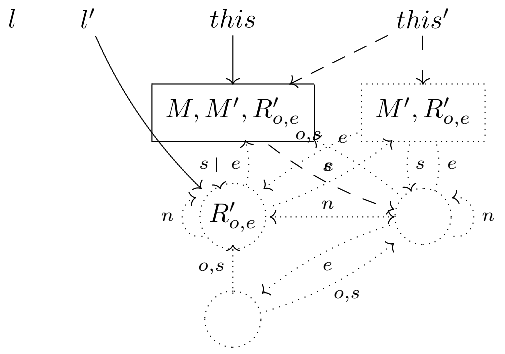
After performing the assignment to `l` (focusing, assigning and then merging), we have:

$\mathbf{3}_2$



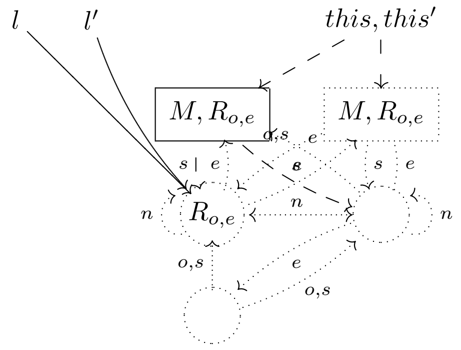
This time, if we take `l` non null and call `notify` we obtain a smaller shape graph at the entry of the procedure, meaning that we attained a local fixpoint. So we explore the false branch of the test `l!=null` and we obtain, as the output of the procedure,

$\mathbf{7}_1$



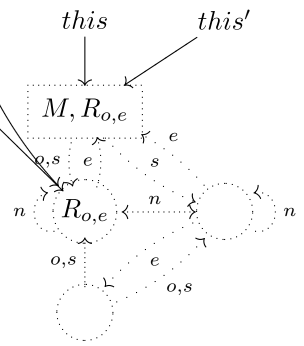
This shape can now be used to compute the return heap of the call `l.e.notify()`. To do that, we just have to forget the unprimed variables and copy the primed variable into unprimed, as the current value can be any value in the tail of the stack before the return from the function call. Note that the modification predicate must be the union of the primed and unprimed ones. Note that we have everywhere we have a predicate, the primed version holds also here. We omit those predicates in the next few pictures. That gives:

$\mathbf{5}_1$



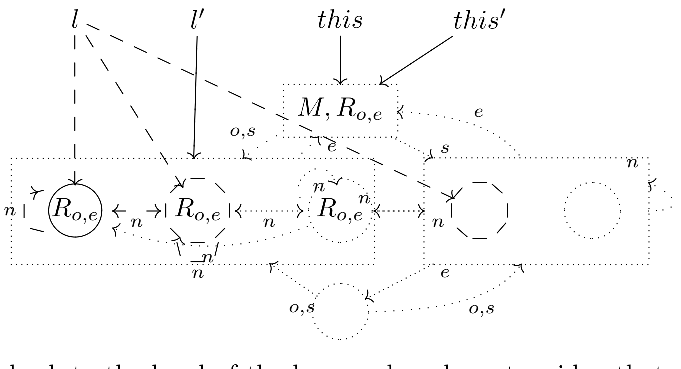
This can be simplified, as there is no definite edge to a definite node, due to the erasure of *this*. An equivalent, simpler graph is:

$\mathbf{5}_1$



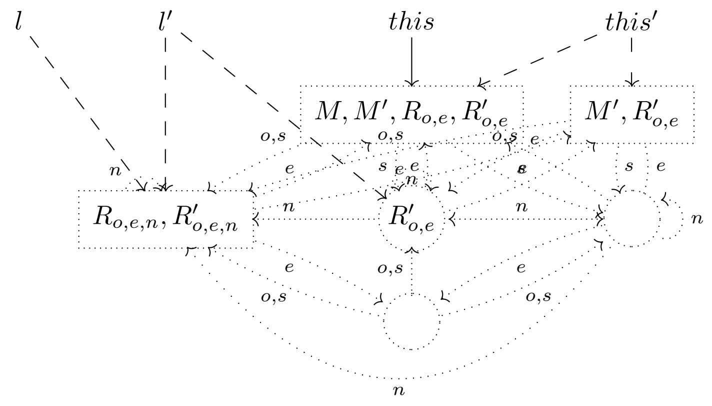
Then we have to assign `l.n` to `l`, which yields (in the following graph we regroup everything which was focused from l in an attempt to make it more clear):

$\mathbf{6}_1$



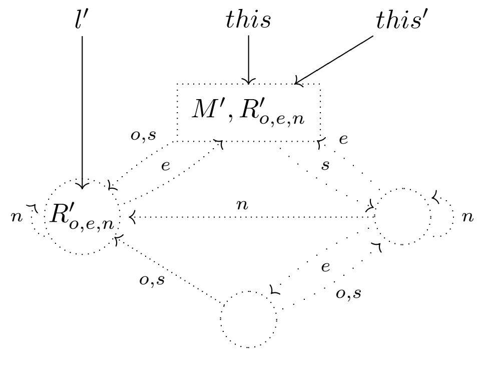
Then we go back to the head of the loop and we have to widen that shape graph with the graph $\mathbf{3}_2$. The result will merge the properties in the nodes that can be pointed to by *this* and *this'* and merge the 3 nodes that can be pointed to by l, adding everything that is reachable through `n` fields.

$\mathbf{3}_3$



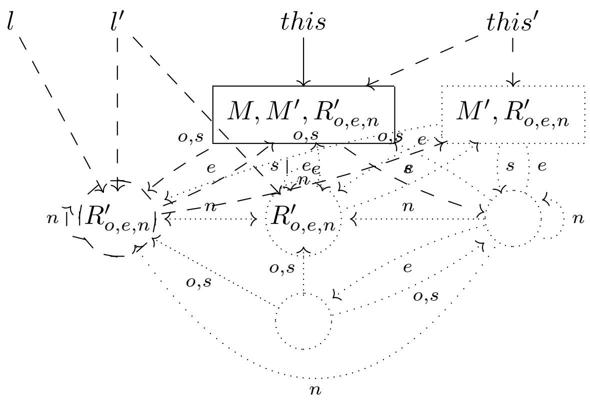
This new shape graph can now be transformed for a call to `l.e.notify()`. We focus on `l.e`, then merge *this* and *this'*, assign `l.e` to `this` and merge l and l', all properties become primed. Then we have to widen with the graph $\mathbf{1}_2$. This time the widening should merge the two nodes that can be pointed to by *this* and those pointed to by l'. Concerning l', we have a finite domain of reachability so we can take the union of $R'_{o,e}$ and $R'_{o,e,n}$, which is $R'_{o,e,n}$. For *this*, we merge nodes with property M' and $R'_{o,e}$ to nodes without any property but reachable only from an e field from a node with property $R'_{o,e,n}$, so we can infer M' and $R'_{o,e,n}$. The result of the widening at the entry of teh procedure is thus:

$\mathbf{1}_3$



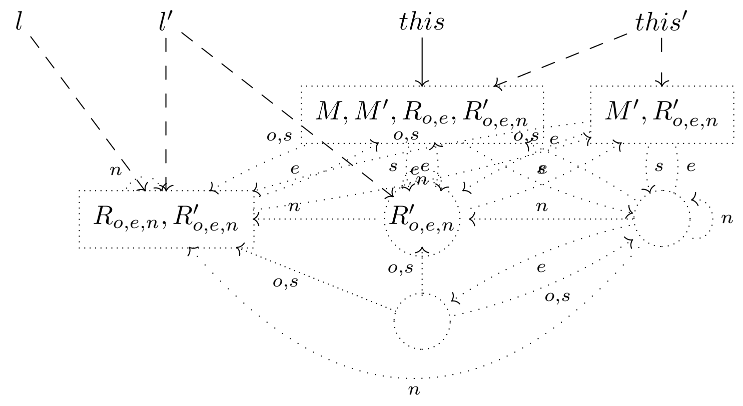
From that graph we assign `this.o` to `l`, yielding

$\mathbf{3}_4$



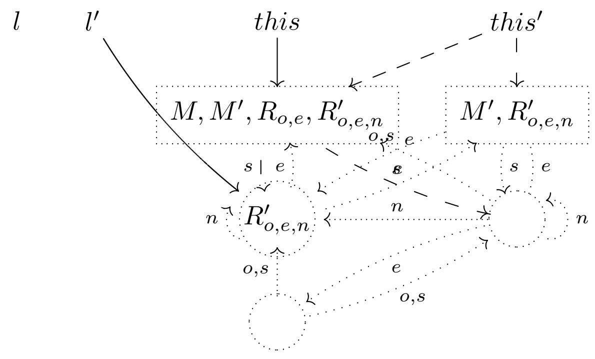
We can also take the union of that computation with the previous iteration ($\mathbf{3}_3$), so that we have

$\mathbf{3}_5$



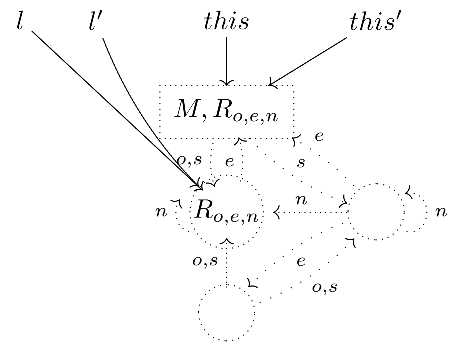
So, taking l to be null, we can have a more general output:

$\mathbf{7}_2$



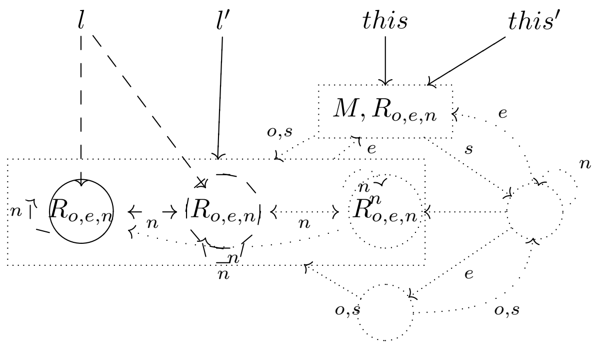
On a return from `notify()`, we thus get (keeping primed properties implied here)

5$_2$



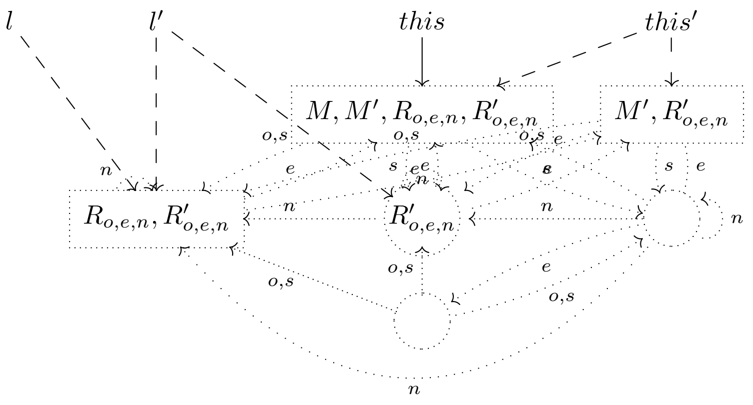
Then we assign `l.n` to `l`

6$_2$



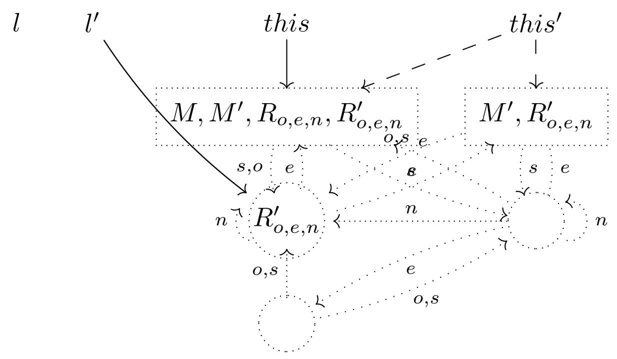
We can widen with **3**$_5$, and this time no node needs to be merged, just the property of the nodes that can be modified to be enlarged, so that we get

3$_6$



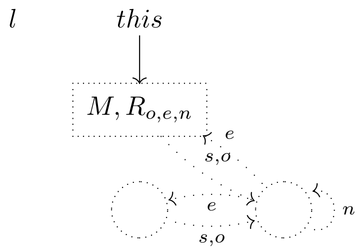
That will not change the input to the procedure, but the output will become

7$_3$



That will not change the return from `l.e.notify()` so we have reached a global fixpoint.

If we specialize the output to the case where the call stack to notify is of size one (no recursive call), we have the invariant at the end of the call is



From that, it is easy to prove that if we add a subject in a disjoint region, no node of the subject's region will be modified.