Interfacing TVLA and Sample

Raphael Fuchs

Bachelor Thesis Report

Chair of Programming Methodology
Department of Computer Science
ETH Zurich

August 11, 2011

Supervised by:
Dr. Pietro Ferrara
Prof. Dr. Peter Müller

Chair of Programming Methodology
inf | Informatik
Computer Science

Abstract

In this bachelor thesis we plug shape analysis into the generic static analyzer Sample, providing an interface to the tool TVLA. Our approach integrates the new domain with the existing semantic domains such that other information about the content of heap locations can be combined with TVLA.

Acknowledgements

I would like to thank my supervisor Dr. Pietro Ferrara for all his helpful support throughout this bachelor thesis, and Prof. Dr. Peter Müller for giving me the opportunity to gain interesting insights into the field of static analysis.

Furthermore, I would like to thank Uri Juhasz for all the helpful discussions we had regarding both practical and theoretical aspects of TVLA. I am also very grateful to Roman Manevich who provided us with a customized TVLA version according to our needs and who was very responsive in fixing any bugs we encountered.

Contents

1 Introduction **6**
1.1 Overview . . . 6
1.2 Shape Analysis: Motivation . . . 6
1.3 TVLA . . . 7
1.3.1 Representing Concrete Heaps . . . 7
1.3.2 Heap Abstraction . . . 8
1.3.3 Expressing Semantics and Programs . . . 8
1.3.4 Refining the Abstraction . . . 9
1.4 Sample . . . 10

2 Design **12**
2.1 Approach . . . 12
2.2 Structure of Heap Domains in Sample . . . 12
2.3 Heap State and Encoding . . . 13
2.3.1 Program Variable Predicates . . . 14
2.3.2 Field Predicates . . . 14
2.3.3 Summarization Predicate . . . 15
2.3.4 Name Predicates . . . 15
2.3.5 Instrumentation Predicates . . . 15
2.4 Graphical Notation for Three-Valued Heap Structures . . . 15
2.5 Translation of Simple Statements . . . 16
2.5.1 Program Variable Management . . . 17
2.5.2 Object Creation . . . 17
2.5.3 Variable Assignment . . . 18
2.5.4 Field Access . . . 18
2.5.5 Field Assignment . . . 19
2.5.6 Assumptions . . . 20
2.5.7 Heap Equality and Least Upper Bound . . . 21
2.6 Heap Identifiers . . . 22
2.6.1 Preserving Names . . . 22
2.6.2 A Naming Scheme for Heap Identifiers . . . 23
2.6.3 Updating the Identifier Space: Replacements . . . 25
2.7 Getting More Precise: Instrumentation . . . 26

3 Evaluation and Results **27**
3.1 Visualization . . . 27
3.2 Performance and Optimization . . . 27
3.3 Testing . . . 30
3.4 Representative Testcases . . . 32
3.4.1 Multiple Structures . . . 32
3.4.2 Abstraction: Creation of a List . . . 32
3.4.3 Traversing Lists . . . 33
3.4.4 Integer Fields and the Numerical Domain . . . 34
3.4.5 Replacements in Action . . . 35
3.4.6 Initialize and Sum Lists . . . 35

4 Conclusions 39
4.1 Known Issues 39
4.2 Future Work 39

A Detailed List of TVLA Actions 41

1 Introduction

1.1 Overview

The goal of this thesis is to enhance the static analyzer Sample with *shape analysis* capabilities by interfacing it with TVLA. In this introduction, we briefly present the motivation of our work as well as a brief description of TVLA and Sample.

1.2 Shape Analysis: Motivation

Shape analysis is concerned with statically and automatically inferring properties about the heap of a program, also called store. A state of the program heap can be thought of as a graph where the nodes represent blocks of allocated memory and the edges pointers between those blocks, together with a set of program variables (called the environment) which may also point to heap nodes.

One possible question is about the reachability of heap nodes. Given two nodes and starting from the first one, is it possible to reach the second one by following pointers? Other important properties include cyclicity, sharing, may-aliasing and disjointness [WSR00]. More generally, one may be interested in the "shape" of a heap, that is a characterization of its data structures. For a program which manipulates singly linked lists, this would mean that we could statically determine that its heap contains lists, and also whether they contain cycles or not. Other common data structures of interest include doubly linked lists, trees and DAGs.

The results of a shape analysis have many applications. They are helpful or even necessary for certain forms of program verification (e.g. freedom from null-pointer dereferences), optimization and automatic parallelization (disjoint data structures may be processed in parallel) etc.

Unfortunately, one can not give precise answers to all of the above questions for all possible programs. It is well-known that many interesting properties such as program termination are undecidable. Even decidable properties are often too expensive to be analyzed as the problem at hand suffers from state space explosion. Therefore we need to introduce approximation: The program semantics is approximated so that the answer is not wrong but less precise. Roughly, if we prove a property on the approximation, then this property is respected by all possible executions. Instead, if the property is not proven, this could be a false alarm because of a too rough approximation.

In the case of heap analysis, we often have programs that manipulate structures of unbounded size. For example, the length of lists is unbounded. However, we need a bounded representation in our analysis. This usually implies the need for a conservative approximation.

Heap abstraction has proven to be a particularly hard problem in static analysis. During the last decade, TVLA has been the most effective and popular approach to deal with this issue.

1.3 TVLA

Parametric shape analysis via 3-valued logic is a technique introduced by Sagiv et al. [SRW02] and to date remains the most promising approach to heap analysis. Lev-Ami implemented these ideas in a tool called TVLA (Three-valued Logic Analysis Engine) [LAS00], which we are going to use in this thesis.

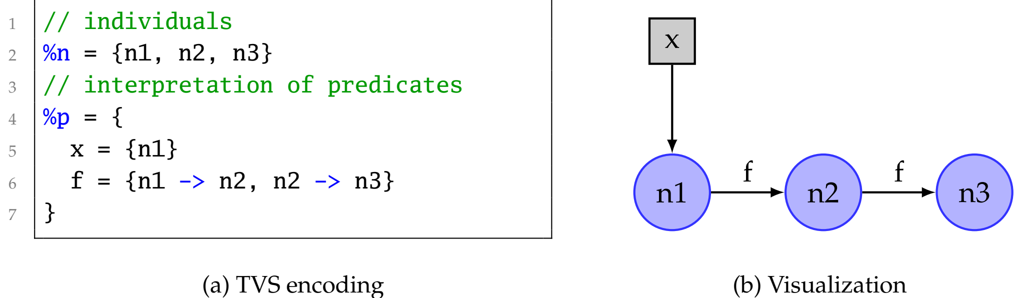
In the following, we outline some of the ideas behind this static analysis and how the theoretical concepts translate to TVLA. We try to keep the explanation general and focused on the logic for now.

1.3.1 Representing Concrete Heaps

Heaps in TVLA are represented using logical structures. A structure S consists of a tuple (U^S, I^S) where U^S is the universe with elements called *individuals* and I^S is an interpretation of a *vocabulary* of predicate symbols over U^S.

A structure describes a particular instance of the heap: We can interpret the individuals to be the set of nodes in the heap graph, i.e. the set of allocated memory blocks. The valuation of the predicates encodes all the information that makes up the heap. *Unary predicates* can for instance encode that a program variable points to a heap node: If variable x references node $n1$, we create a unary predicate x such that $x(n1) = 1$. *Binary predicates* on the other hand can express which objects a field references (defining the edges between nodes if the heap is thought to be a graph).

Example: Figure 1 shows our running example with a structure that consists of three heap nodes as it is encoded in the TVS format used by TVLA. We see that the unary predicate x holds for n1, i.e. $x(n1) = 1$, and n1 is related to n2 by the binary predicate f, i.e. $f(n1, n2) = 1$.



(a) TVS encoding (b) Visualization

Figure 1: A 2-valued heap structure

To extract information from such a heap, one may simply evaluate a formula of first-order logic (with transitive closure) over the structure.

1.3.2 Heap Abstraction

Instead of 2-valued logical structures, Kleene's 3-valued logic is used to describe abstract heaps with 3-valued structures: In addition to 'true' and 'false', there is a third 'unknown' (also written $1/2$) truth value.

When considered 'equivalent' in some sense, several heap nodes can be combined into a single one, called a summary node, representing one or more concrete nodes. A given 3-valued structure can (conservatively) represent infinitely many concrete ones.

The decision when to apply summarization is based on the notion of *canonical abstraction*: All nodes that agree on the values of a chosen set of unary *abstraction predicates* are considered to be equivalent and therefore mapped to the same individual in the output structure.

```
1 // individuals
2 %n = {n1, n2}
3 // interpretation of predicates
4 %p = { x = {n1}
5   f = {n1 -> n2: 1/2, n2 -> n2: 1/2}
6   sm = {n2: 1/2} // n2 is summarized
7 }
```

(a) TVS encoding

x
f
n1
n2
f

(b) Visualization

Figure 2: Abstracted structure (3-valued)

Example: Figure 2 displays the same structure as in the previous example, but this time after abstraction was applied (with x as abstraction predicate). Since $x(n2) = x(n3) = 0$ in the original structure, $n1$ and $n2$ were summarized in the 3-valued structure.

1.3.3 Expressing Semantics and Programs

In TVLA, the semantics of statements are defined by *actions*. Actions can take parameters and describe how to transform a given state into a new one using predicate logic. In particular, this is done using a set of *predicate-update formulae*. For each predicate in the vocabulary, a logical formula, the update predicate, specifies the new interpretation in terms of the pre-state.

As input, a TVP (Three-Valued Program) file declares the predicates used, the available actions, and specifies a control flow graph where the edges are instantiations of the actions.

Example: Let the abstract heap in Figure 2 be our pre-state and assume we want to set the f-field of node $n1$ to `null`. In a programming language this could be expressed as `x.f = null`. To encode this as TVP, we introduce a general action `setFieldNull(c,n)` which can be used to set a particular field of a node pointed by a program variable to `null`. We then instantiate this action with x (to denote $n1$) and our field predicate f

as parameter. Listing 1 shows the complete TVP and Figure 3 displays the resulting abstract heap after TVLA was invoked.

```
// predicate declaration
%p x(v_1) unique
%p f(v_1,v_2) function
%%
// action declaration
%action setFieldNull(c,n) {
   { n(v_1,v_2) = n(v_1,v_2) & !c(v_1) }
}
%%
// control flow graph
start setNextNull(x,f) end
```

Listing 1: TVP for example

```
// individuals
%n = {n1, n2}
// interpretation of predicates
%p = { x = {n1}
  f = {n2 -> n2: 1/2}
  sm = {n2: 1/2} // n2 is summarized
}
```

(a) TVS encoding

(b) Visualization

Figure 3: Result after execution

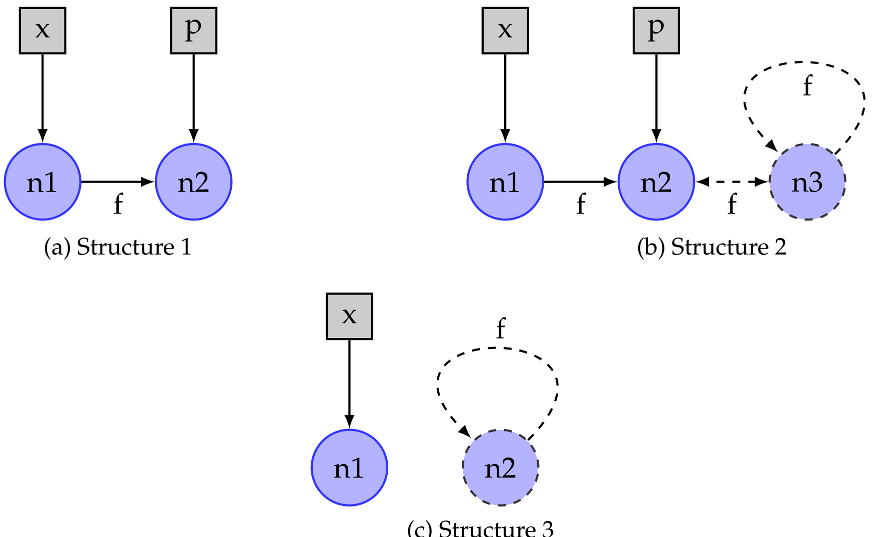
1.3.4 Refining the Abstraction

To improve the precision of the analysis, *instrumentation predicates* can be added. They express properties of interest in terms of basic *core predicates*. They often lead to finer distinctions among the concrete structures represented by the abstract heap and can be used to tune the shape analysis for certain data structures.

Summary nodes can also be split into separate nodes again in a process called *materialization*: Certain 'unknown' (1/2) values in the heap structures may be forced to take on definite ('true'/'false') truth values. *Focus formulas* are used to characterize the parts of the heap that need to assume definite values, i.e. that we are focusing on.

Finally, *integrity constraints* (also called compatibility constraints) ensure that the abstracted structures satisfy some consistency rules (e.g. global invariants). Sometimes, a 3-valued structure created does not represent any legal concrete structure and may therefore be dropped.

Example: Consider our running example again. Assume we start with the abstract state depicted in Figure 2 and want to access the node that the first node $n1$ references with



(a) Structure 1

(b) Structure 2

(c) Structure 3

Figure 4: Result of accessing field f

its field f, i.e. to execute `p = x.f`. The resulting heap structures are shown in Figure 4. In the first and second structure we see that a node was successfully materialized out of the summary node again and is now referenced by p. The information how many nodes there were exactly in the original structure is lost as part of the abstraction. However, this failed for the third structure c: The abstraction also did not preserve the fact whether the concrete nodes represented by $n2$ are actually reachable from $n1$. Therefore, the case where this is not the case was also considered. We can improve the analysis and eliminate this case by adding an instrumentation predicate for reachability.

1.4 Sample

Sample (Static Analysis of Multiple LanguagEs) is a generic static analyzer. It is based on the theory of abstract interpretation [CC77, CC79] and was developed at the Chair of Programming Methodology at ETH over the last 2 years.

Different languages can be translated to an intermediate representation (`Simple`) which is then analyzed by Sample. Currently, there are back-ends for Scala and JVM byte-code.

It supports and already contains a wide range of different analyses. It has non-relational numerical domains for intervals, as well as relational domains like octagons and polyhedra. Recently, a new string analysis described in [CFC11] was implemented in Sample. Former work also includes a domain for the static type analysis of pattern matching [Fer10].

A few simple heap domains exist already in Sample. One of them approximates all heap references with exactly one abstract location. Other variants use program points as the abstract locations to keep the heap bounded. This thesis seeks to improve the

heap abstraction with the power of TVLA.

2 Design

This section outlines the design of our heap domain. We start by explaining the general ideas of our approach and then move on to give the detailed translation of Sample heap domain concepts to TVLA concepts.

2.1 Approach

The ideas behind TVLA are very general and could be implemented in Sample. We chose to use the binary distribution of TVLA 3.0 [1] as an external tool and interface it to Sample. This decision has the following reasons: On one hand, it would be a tremendous effort to replicate all the needed functionality and exceed the scope of this thesis. On the other hand, TVLA is already heavily optimized and proved to be reliable.

The structure of Sample implies that we cannot simply encode the whole Simple program as a three-valued program (TVP) and let TVLA perform the analysis of that whole program. Instead, we have to invoke TVLA for every statement which modifies the structure of the heap. This is because Sample not only tracks the effect of a statement on the heap domain, but also on a separate semantic domain.

The procedure therefore is as follows for every operation which modifies the heap:

1. Write a TVP file that declares the actions to execute.
2. Encode the valuations of all predicates of the current heap as a TVS file
3. Run TVLA on the given TVP and TVS
4. Parse the resulting TVS output.
5. Update our heap representation and produce a new state of the heap domain.

2.2 Structure of Heap Domains in Sample

One of the requirements was to integrate our new heap analysis into the existing infrastructure provided by Sample. In particular, it must implement the trait HeapDomain, the relevant parts of which are shown in Listing 2.

```
1  trait HeapDomain[T <: HeapDomain[T, I], I <: HeapIdentifier[I]] extends Analysis
       with LatticeWithReplacement[T] {
2    /**
3     Creates an object on the heap
4     */
5    def createObject(typ : Type, pp : ProgramPoint) : (HeapIdSetDomain[I], T,
         Replacement);
6    /**
7     Returns an identifier which represents the field of a given object
8     */
```

[1] TVLA 3.0alpha, http://www.cs.tau.ac.il/~tvla/

```
9   def getFieldIdentifier(objectIdentifier : Assignable, name : String, typ :
        Type, pp : ProgramPoint) : (HeapIdSetDomain[I], T, Replacement);
10
11  /**
12   Called when an assignment was completed.
13   */
14   def endOfAssignment() : (T, Replacement);
15  /**
16   Assigns the given expression to the given variable
17   */
18  def assign[S <: SemanticDomain[S]](variable : Assignable, expr : Expression,
        state : S) : (T, Replacement);
19  /**
20   Assigns the given expression to the field of identifier
21   */
22  def assignField(obj : Assignable, field : String, expr : Expression) : (T,
        Replacement);
23  /**
24   Assumes that a given expression holds
25   */
26  def assume(expr : Expression) : (T, Replacement);
27  /**
28   Create a variable
29   */
30  def createVariable(variable : Assignable, typ : Type) : (T, Replacement);
31  /**
32   Create a variable which is a parameter of analyzed method
33   */
34  def createVariableForParameter(variable : Assignable, typ : Type, path :
        List[String]) : (T, Map[Identifier, List[String]], Replacement);
35  /**
36   Removes a variable
37   */
38  def removeVariable(variable : Assignable) : (T, Replacement);
39  /**
40   Computes the least upper bound of two heaps
41  */
42  def lubWithReplacement(left : T, right : T) : (T, Replacement)
43  /**
44   Computes the greatest lower bound of two heaps
45  */
46  def glbWithReplacement(left : T, right : T) : (T, Replacement)
47 }
```

Listing 2: Interface provided by a HeapDomain

2.3 Heap State and Encoding

The heap domain not only provides operations to access and modify the heap, but it has to keep its own internal state which it uses to conservatively represent the program

heap at a point during program execution. In other words, each instance of HeapDomain describes a set of possible concrete heaps. It is natural to represent this information in the same way TVLA does, that is by a finite set of three-valued structures.

As we have seen in Section 1.3.1, such structures consist of a vocabulary set, the predicate symbols. All the structures in a heap state are defined over the same vocabulary. We now give an overview over all the essential predicates we use.

2.3.1 Program Variable Predicates

For every program variable x, we keep a unary predicate $P_x(v)$. A program variable may either point to a node, or it may be null. In other words, $P_x(v) = 0$ holds for all individuals v except at most one. We can express this in TVLA with the property unique.

```
%s PVar {x,y,z,..}
foreach (x in PVar) {
  %p x(v_1) unique
}
```

Listing 3: Declaration of program variable predicates

2.3.2 Field Predicates

Heap nodes are connected to each other when the field of an object references another one. For every possible field f, we introduce a binary predicate $P_f(v_1, v_2)$. For example, if the field n of node a references node b, it holds that $P_n(a, b) = 1$.

Since a field can reference at most one other object, P_f is a special case of a general binary predicate: To express this in TVLA, we can specify the property function which adds the following integrity constraints:

$$\exists v : P_f(v, v_1) \wedge P_f(v, v_2) \Rightarrow v_1 = v_2$$
$$\exists v_1 : P_f(v, v_1) \wedge v_1 \neq v_2 \Rightarrow \neg P_f(v, v_2)$$

```
%s Fields {n, i, ...} // all field names
foreach (f in Fields) {
  %p f(v_1,v_2) function
}
```

Listing 4: Declaration of field predicates

2.3.3 Summarization Predicate

Summarized nodes play a crucial role in TLVA. One abstract summarized node represents one or more nodes in the concrete. TVLA's built-in unary predicate sm serves to denote whether an individual is summarized. It doesn't need to be declared in TVP.

2.3.4 Name Predicates

In order to refine the precision of the analysis, we have to assign a unique name to every node by letting a unary predicate with the chosen name point to the node. We call those predicates name predicates.

```
1 %s Names { n1, n2, ...}
2 foreach (z in Names) {
3   %p z(v_1) nonabs
4 }
```

Listing 5: Declaration of name predicates

2.3.5 Instrumentation Predicates

To make the analysis more precise, we add several instrumentation predicates which try to capture certain shape invariants.

2.4 Graphical Notation for Three-Valued Heap Structures

We adopt the widely established notation to visualize three-valued structures throughout this document, favouring it over the TVS input and output format of TVLA. Figure 5 incorporates all relevant elements of our notation which can be explained as as follows:

- Rectangular boxes like x denote program variables.
- Ovals represent a heap nodes (individuals of the structure). In our example, a, b and c are the heap nodes.
- Dashed ovals represent summarized nodes, like node c in the example.
- An arrow from a program variable to a heap node means that the program variable points to the node. In Figure 5, variable x points to node a.
- Similarly, for a unary predicate, a label is drawn with arrows starting from the label pointing to the nodes for which it holds. In the example, $n1$, $n2$ and $n3$ are such unary predicates.
- Labelled arrows between heap nodes denote that the given field of the first node references the second one. E.g. the field i of node a references node b.

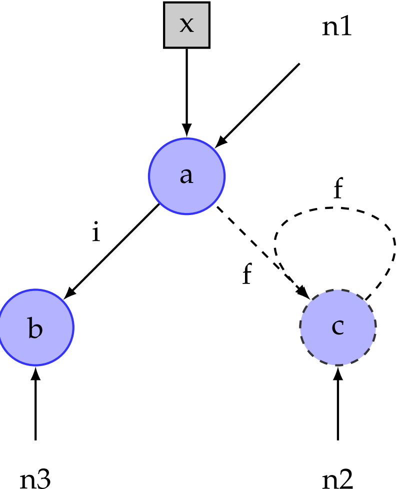


Figure 5: Graphical notation example

- Solid arrows symbolize `true`, dashed arrows `unknown` (1/2) values, and the absence of arrows `false`. Consider e.g. the dashed arrows between a and c: It implies that $f(a, c) = 1/2$.

2.5 Translation of Simple Statements

While we describe the programs we intend to analyze using TVP in TVLA, we have Simple in Sample. This language was specifically designed to support object-oriented features. This results in a major difference in the level of abstraction between the two languages; TVP files like the examples shipped with the TVLA binary distribution typically look like direct translations of C programs. On the other hand, Simple programs still exhibit an object-oriented structure and are much more concise, at the cost of complexity.

For example, Simple does not clearly distinguish between expressions and statements. Furthermore, it has a complex type system whereas types are usually not considered in TVLA.

As a consequence of these differences, a suitable translation from Simple to TVLA becomes necessary: We need to express the higher-level Simple statements in terms of more primitive TVLA operations.

When Sample executes a program in the abstract, a single Simple statements results in several method calls on the heap domain. These method calls in turn invoke TVLA actions. Every operation of our heap domain is assumed to modify the heap and return the new resulting state.

Some operations also produce a result (e.g. a new node) which we would like to refer to later. This is achieved using `HeapIdentifiers`. However, since different structures

contained in the heap may have different results and we sometimes need to take their union, those operations return a set of such identifiers.

2.5.1 Program Variable Management

In contrast to pure TVP where all predicates (and therefore program variables) are declared globally and before execution, Simple has scoped variables. This implies that variables may appear and disappear throughout execution. For an example, consider this Scala code:

```
1  if (test) {
2    val a = new a
3    val b = new b
4    ...
5  }
```

Listing 6: Scoped variables

When `createVariable` or `removeVariable` are called, we simply declare a new program variable predicate or remove it from all structures. This is done without running TVLA. Note that we ignored variable shadowing for simplicity, but this could be done by encoding the scope in the variable name.

2.5.2 Object Creation

Dynamic object creation is essential for most real-world object-oriented programs. It is also the reason why we need to abstract concrete heaps since heaps can grow unboundedly.

As mentioned, Simple does not really distinguish between statements and expressions: An expression '`new A`' may for instance occur inside an `if` statement, and the resulting value is assigned only after the `else` branch was evaluated. However, if we create a new heap node in TVLA without letting anything point to it, we may not be able to access it later on again in another statement, like an assignment which is usually done after an object creation.

To overcome this problem, we decided to automatically create a *temporary program variable* and let it point to the newly created heap node. After an assignment, the heap operation `endOfAssignment` is performed and we drop all the created temporary variables.

Our TVLA action in Listing 7 makes use of the built-in predicate `isNew` which designates the new node. Note that it is important to provide update formulas for any instrumentation predicates we may add later since they cannot be updated automatically by TVLA for new nodes.

```
1  %action createObject(temporary) {
2    %new
```

```
3    {
4      temporary(v) = isNew(v)
5      // ... manual updates of instrumentation predicates ...
6    }
7  }
```

Listing 7: Action for object creation

2.5.3 Variable Assignment

In most assignments, program variables are assigned the result of another expression. This other expression may be another variable, the resulting heap identifier of another heap operation or `null`. The most common cases are:

```
1  x = y              // source is a variable
2  x = new SomeClass  // source is heap id of a new node
3  x = y.n            // source is heap id of a field
4  x = null           // constant null
```

Listing 8: Variable assignments

If the right-hand side is not `null`, we always have a unary predicate pointing to the source of the assignment. In the case of a variable, it is the program variable predicate and in case of a heap identifier it is the temporary which was created. We may therefore simply copy the valuation of the unary predicate.

We treat a `null`-assignment as a special case, with an additional TVP action in Listing 9.

```
1  // assign source to target unary predicate (program variable)
2  %action copyVariable(target,source) {
3    %f { source(v) }
4    {
5      target(v) = source(v)
6    }
7  }
8  // special case for null
9  %action setVariableNull(target) {
10   {
11     target(v) = 0
12   }
13 }
```

Listing 9: Action for variable assignment

2.5.4 Field Access

In Sample we expect expressions of the form "`target.field`" to return an identifier which points to the content of the referenced field. This identifier may possibly also represent

`null`, for example in the case of uninitialized fields.

Afterwards, the identifier is usually used as the source of an assignment (e.g. `y = x.n`) or the target of another field access (e.g. `x.n.n`). In any case, we create a new temporary variable pointing to the result of the performed field access, just like we do for created objects.

Example: When the Simple statement `y = x.n.n` is executed, it is translated to the following sequence of operations (slightly simplified pseudo code):

	Simple Notation	*HeapDomain method*	*TVLA Action used*
1.	`temp1 = x.n`	`getFieldIdentifier`	`extractField`
2.	`temp2 = temp1.n`	`getFieldIdentifier`	`extractField`
3.	`y = temp2`	`assign`	`copyVariable`
4.	`temp1 = temp 2 = null`	`endOfAssignment`	`setVariableNull`

Accessing the object referenced by a field is one of the situations in which the object we are trying to access may have been summarized with other nodes. However, we would like our result be one specific, concrete node and not a summarized one. This is where materialization comes into play: If we add the focus formula

$$\exists(v_1, v_2) : P_{target}(v_1) \wedge P_f(v_1, v_2)$$

we get definite values for the temporary which points to the accessed node. It either points to a single non-summarized node or is `null`.

```
%action extractField(destinationVar,targetVar,fieldName) {
   %f { E(v_1,v_2) target(v_1) & field(v_1,v_2) }
   {
      destinationVar(v) = E(v_1) targetVar(v_1) & fieldName(v_1, v)
   }
}
```

Listing 10: Action for field access

2.5.5 Field Assignment

The treatment of assignments to fields of an object is similar to normal assignment. The cases that may occur look basically the same:

```
x.n = y             // source is a variable
x.n = new SomeClass // source is heap id of a new node
x.n = y.n           // source is heap id of a field
x.n = null          // constant null
```

Listing 11: Field assignments

However, the translation to TVLA is different, as it involves a field predicate and we need to access the target whose field is assigned. Also note that in `x.n = y.n`, the left-hand side and right-hand side are treated in a different way by Sample: It evaluates the

field access `y.n` but `x.n` is not interpreted as a field access; the later simply denotes the target and the field which is to be assigned. We again have a separate action to set fields to `null` as can be seen in Listing 12.

```
%action setField(target,field,source) {
  %f { target(v), source(v) }
  {
    field(v_1, v_2) = field(v_1, v_2) | target(v_1) & source(v_2)
  }
}
// spcial case for settings fields to null
%action setFieldNull(target,field) {
  %f { target(v) }
  {
    field(v_1, v_2) = field(v_1, v_2) & !target(v_1)
  }
}
```

Listing 12: Action for field assignment

2.5.6 Assumptions

When the control flow of a program branches, it is often useful if an analysis can assume a condition that lead to the particular branch taken. This is exactly the purpose of assumptions in Sample.

For a heap domain, the relevant branching conditions are reference comparison expressions. We limit ourselves to the following most common cases:

```
x == y
x != y
x == null
x != null
```

Listing 13: Implemented assumptions for the heap domain

As we have decided in Section 2.3, our heap state contains a set of heap structures. When executing `assume`, we can evaluate the condition in all the structures and safely drop those in which it evaluates to false. We need to make sure it evaluates to a definite value, so we add focus predicates for the predicates of the program variables in the expression.

In TVP, we can specify a precondition formula which filters out structures which do not satisfy it. Listing 14 shows the declared actions.

```
%action assumeVariableEqual(var1, var2) {
  %f { var1(v), var2(v) }
  %p A(v) var1(v) <-> var2(v)
  {}
}
```

```
6  %action assumeVariableNotEqual(var1, var2) {
7    %f { var1(v), var2(v) }
8    %p !A(v) var1(v) <-> var2(v)
9    {}
10 }
11 %action assumeVariableNull(var) {
12   %f { var(v) }
13   %p !(E(v) var(v))
14   {}
15 }
16 %action assumeVariableNotNull(var) {
17   %f { var(v) }
18   %p E(v) var(v)
19   {}
20 }
```

Listing 14: Actions for assume

2.5.7 Heap Equality and Least Upper Bound

During an analysis, Sample performs an iteration over the control flow graph of the program, executing the abstract transformers on the states until a fixed point is reached. For this to work, we need a suitable definition of equality and a least upper bound operation on heap states. We took the decision that our abstract heaps are equal when they contain the same structures in the following sense: Two structures are considered equal if they have the same universe (i.e., the names of heap nodes match) and they agree on the values of all predicates over the universe. We do not determine if they are isomorphic ignoring the names.

A node in the control flow graph can have several predecessors. This happens for example after `if-else`-statements when the two branches merge again, or at the entry of a loop due to a back-edge. In that case, a *join* operation is used to combine the post-states of the predecessors into a single pre-state of the current node. It is important that such an operation is safe in the sense that it returns an upper bound of the information contained in the inputs.

Sample invokes the operation `lub(s1: T, s2: T): T` on an abstract domain `T` whenever it needs to join two states. Since in our heap domain every heap state consists of a set of three-valued structures, we could simply compute their union. However, during repeated application of this operation, the set may grow unbounded and the analysis would not terminate because no fixed point is reached. The reason for this is that the union of two structure sets may not be minimal: it can contain structures which are isomorphic or are contained in each other.

We found that TVLA automatically performs a join operation on all the input structures at the start of program, not only on joins of the control flow graph. Therefore we pass the union of the input structures to TVLA and let it “minimize” this set (i.e., we get as output a set of structures which are not isomorphic but still contain all the concrete structures described by the input).

```
%action lub() {
  {}
}
```

Listing 15: Action for the least upper bound

2.6 Heap Identifiers

Identifiers are used in Sample to name entities (variables or heap nodes) which are relevant to the analysis of a program. While different domains are independent to a large extent, their states share a common set of identifiers.

The canonical example for identifiers is an integer variable identifier which simply is the variable name listed in the program. The interval domain associates all these numerical identifiers with an integer interval to approximate their runtime values. At the same time, another semantic domain may handle the identifiers differently and track for instance their dynamic type.

For our heap domain, we let the heap nodes, that are the individuals of three-valued structures, denote our identifiers. It seems reasonable that in one structure, identifiers are unique (otherwise the universe would not be a set) while on the other hand, different structures may contain the same heap identifiers.

The set of heap nodes changes throughout the analysis, and so does the set of identifiers; it is not fixed. Identifiers are created when new objects are created. However, one purpose of the heap domain is to abstract the concrete heap by summarizing nodes. When that happens, identifiers may disappear. A new identifier also appears when materializing a node from a summary node.

2.6.1 Preserving Names

Tracking the names of nodes in TVLA is challenging. The input names are translated to an internal format, and when producing output TVLA numbers the nodes in an unknown and unpredictable way.

When structures given as input are modified (that is, they are summarized or materialized), TVLA does not tell us what happens to these structures in the output.

To solve these issues, we add predicates to track names, from now on referred to as *name predicates*. Each time we run TVLA, a unary predicate is added for every node name, pointing to the named node. Afterwards, the values of the name predicates tell us what happened to the original names. Table 1 lists the most common cases and how the results are interpreted.

Usually unary predicates are used to distinguish between different structures when a join of heap states is performed. Because we are just naming nodes, we do not want the

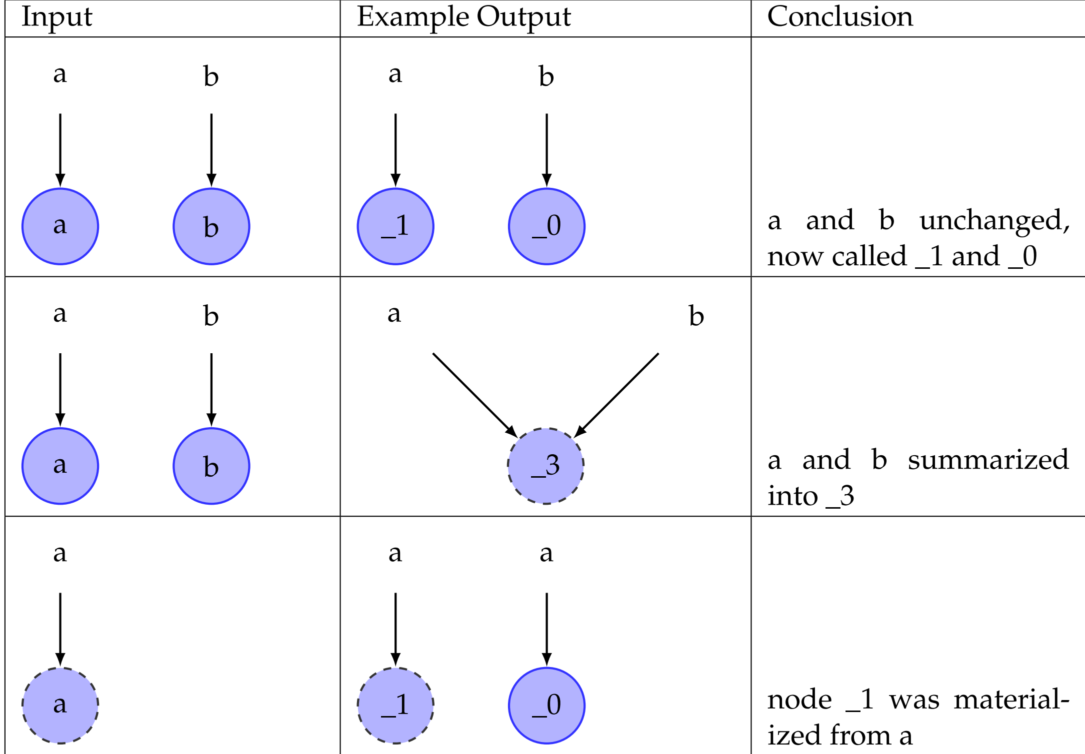


Table 1: Illustration of name predicates with examples

name predicates to have an influence on the abstraction. We were able to achieve this behaviour using the property "non-abs" to declare non-abstraction predicates. Such non-abstraction predicates allow nodes to be merged even though different non-abstraction predicates hold for them [LAS00]. Experiments showed that "non-abs" only has an influence in TVLA version 3.0a with an option to use a *partial join* [LAIS06, MSRF04].

2.6.2 A Naming Scheme for Heap Identifiers

So far we omitted what the names of heap nodes, our heap identifiers, look like. For the reasons given above, we cannot use the names assigned to them by TVLA. A first simple idea is to consecutively number all the created heap nodes. The numbers are assigned based on the pre-state, not counted globally, since we always need to obtain the same result when an operation is performed on a pre-state.

However, in general we would lose a lot of precision regarding the semantic domain, as illustrated in the example in Listing 16. Given the same pre-state, the newly created objects get assigned the same heap identifier in both branches of the `if`-statement, even though they might be used and modified in a different way. When the states after the two branches are combined into the post-state of the `if`-statement, Sample takes the least upper bound. The values associated with the said heap identifier in the other domains are also combined into one, entailing a loss of precision which would not have happened if we had assigned two different names.

```
// pre-state
if (someCondition) {
  x = new Object
  // ..more heap updates..
} else {
  y = new Object
  // ..more heap updates..
}
// post-state
```

Listing 16: Imprecision of consecutive names

As an improvement, we introduce a naming scheme that considers the context in which a heap identifier first appears: The name of a new node is based on the program point pp where it is created. At a given program point pp, several nodes may be created due to loops, so we add a counter c, forming a tuple (pp, c). The counter is then incremented at each iteration. Let PPC denote the set of all these tuples:

$$PPC := ProgramPoint \times \mathbb{N}$$

We also have to consider the case where two or more nodes are summarized into one: In this case the names are combined to form a single one in an operation we call *merge*. We do this by taking the union of the name tuples (pp, c) described above. For this reason, every name consists of a set $E \in \mathcal{P}(PPC)$ of such tuples, which simply has a single tuple for newly created nodes.

Sometimes new nodes appear for other reasons than performing a `new`- statement, e.g. when a nodes was materialized out of a summary node. As illustrated in the third case of Table 1, such a node 'inherits' the naming predicate of the summary node. It is therefore necessary to add another characteristic uid to the set of the program point / loop counter pairs in order to make all names unique within a structure. A name thus is a tuple (E, uid). By default, we let $uid = 0$, and only increase it when necessary, basically numbering nodes like in the naïve naming scheme described above, as a last resort to guarantee the uniqueness of names. Formally, the set of our heap identifiers is defined as

$$HeapID := \mathcal{P}(PPC) \times \mathbb{N}$$

We now move on to formally define the *merge* operation. While the basic idea is to take the union of all the $(pp, c) \in PPC$ of the nodes to be merged, we to have at most one element of a given pp in such a set. Therefore we adopted this solution: If both nodes contain a tuple with the same program point, say (pp, i) and (pp, j), the resulting node will only contain $(pp, min(i, j))$ instead of both:

$$merge : HeapID \times HeapID \rightarrow HeapID$$

$$merge(n_1, n_2) = merge\left((E_1, uid_1), (E_2, uid_2)\right) := (E', min(uid_1, uid_2))$$

where

$$E' = \{(pp, c) \mid \exists (pp, c') \in E_1 \cup E_2 \wedge c = \begin{cases} min(c_1, c_2) & \text{if } (pp, c_1) \in E_1 \wedge (pp, c_2) \in E_2 \\ c_1 & \text{if } (pp, c_1) \in E_1 \wedge \nexists c_2 : (pp, c_2) \in E_2 \\ c_2 & \text{if } (pp, c_2) \in E_2 \wedge \nexists c_1 : (pp, c_1) \in E_1 \end{cases} \}$$

The reason for this specialized operation is to keep bounded the number of identifiers we produce. The number of program points pp contained in our identifiers is clearly bounded, as every program has finitely many of them. Since TVLA also uses a bounded abstraction for the heap, new nodes repeatedly created at a given program point will start to be summarized with existing ones. Here lies the reason for the rather convoluted *merge* operation: By taking the minimum of the counters c, we ensure they are not increased further.

Note that other naming schemes would be possible and sound. However, we found it to be most suitable for the precision of our analysis.

2.6.3 Updating the Identifier Space: Replacements

We described how to assign and update identifiers in a heap structure of our heap domain. However, the space of identifiers is shared with our semantic domain, and so it needs to be made aware of our modifications. For example, we need to make explicit that a new identifier appeared because the two existing ones were merged, so that the information associated with them in other domains also can be combined.

To capture all updates to identifiers within a structure after we executed a heap operation, we introduced a data structure rep, for "replacement". All our heap operations return a replacement to Sample for further processing. A replacement is a partial function

$$rep: \quad \mathcal{P}(HeapID) \rightarrow \mathcal{P}(HeapID)$$

It basically describes what the heap identifiers of the input structure under consideration were replaced with. Consider the following examples:

1. Some heap identifier n was not summarized with others or materialized. This corresponds to the case in the first row in Table 1 and is expressed with a replacement such that

$$rep(\{n\}) = \{n\}$$

2. Two heap identifiers n_1 and n_2 were merged due to summarization into an identifier n_3. This for instance happens in the second row of Table 1. Then

$$rep(\{n_1, n_2\}) = \{n_3\}$$

   The replacement can be used by another domain: E.g. assume there is a state σ_N of the interval domain satisfying $\sigma_N(n_1) = [1..2]$ and $\sigma_N(n_2) = [2..3]$. Given the replacement, it can be updated to σ'_N such that $\sigma'_N(n_3) = [1..3]$.

3. An identifier n_5 was duplicated from n_4, e.g. due to materialization of the summary node n_4. This case corresponds to the third row of Table 1. Then

$$rep(\{n_4\}) = \{n_4, n_5\}$$

Again, consider the same domain as above, this time with $\sigma_N(n_4) = [1..1]$. After the replacement was applied, we have $\sigma'_N(n_4) = \sigma'_N(n_5) = [1..1]$.

2.7 Getting More Precise: Instrumentation

With the predicates and actions described above, our analysis performs a conservative approximation of the concrete heap semantics. However, summarization immediately results in a significant information loss which makes the analysis very imprecise.

By adding instrumentation predicates and some constraints, we can compensate for a part of this information loss. However, such instrumentation predicates are always designed to tune the analysis to a certain data structure one is interested in. It remains impossible to obtain precise results for all cases. Moreover, adding more instrumentation predicates usually makes an analysis slower. We took the decision only to support lists as part of this thesis, while additional predicates could be added in further work.

The predicates we chose are more or less standard when analyzing singly-linked lists and can also be found in the examples shipped with TVLA. Briefly, the predicates, their purpose and definition in terms of core predicates are the following:

- *Transitive Reflexive Reachability:* For every field `f`, we add a predicate which describes whether a node `v2` can be reach from `v1` by following zero or more `f`-fields:

  ```
  t[f](v_1, v_2) = f*(v_1, v_2) transitive reflexive
  ```

- *Shared-ness:* For every field `f`, we define a predicate that captures whether there is more than one object (node) whose field `f` references a given node `v`:

  ```
  is[f](v) = E(v_1, v_2) (v_1 != v_2 & f(v_1, v) & f(v_2, v))
  ```

- *Reachability from variables:* For every field `f` and program variable `x`, we define a predicate that captures whether a node `v` can be reached from `x` along zero or more `f`-fields:

  ```
  r[f,z](v) = E(v_1) (z(v_1) & t[f](v_1, v))
  ```

From the reachability properties combined with the fact that nodes are not shared can be used to infer that a heap structure is indeed acyclic. Nodes with a single field can be shown to form an acyclic singly-linked list.

3 Evaluation and Results

3.1 Visualization

Heaps are complex structures and are therefore best rendered graphically for human inspection. To visualize the results of an analyzed program, we considered two use cases.

Interactive Visualization

In the first use case, a user interactively explores the results in a GUI. This is usually done after manually analyzing a single method at hand. For that purpose, we extended the existing GUI of Sample. It is able to display the program's control flow graph (CFG), as well as the state at every program point.

In our analysis, a heap state is displayed as a set of graphs including all the program the program variables. Each graph visualizes one three-valued structure. The numerical information associated with an identifier can be viewed by clicking on it. All the heap graphs are drawn using the `jGraph` library and all heap nodes can be moved by dragging them.

Figure 6 shows our visualization for the code snippet in Listing 17. The control flow graph of the program appears in Figure 6a. After clicking on the end node in this graph, the three-valued structures of the result are displayed like in Figure 6b. In our example, there are two distinct structures, divided visually by a horizontal line. Finally, Figure 6c shows how the numerical state associated with a heap node can be viewed: The analysis determined that the value of the selected node is in the interval $[1..1]$.

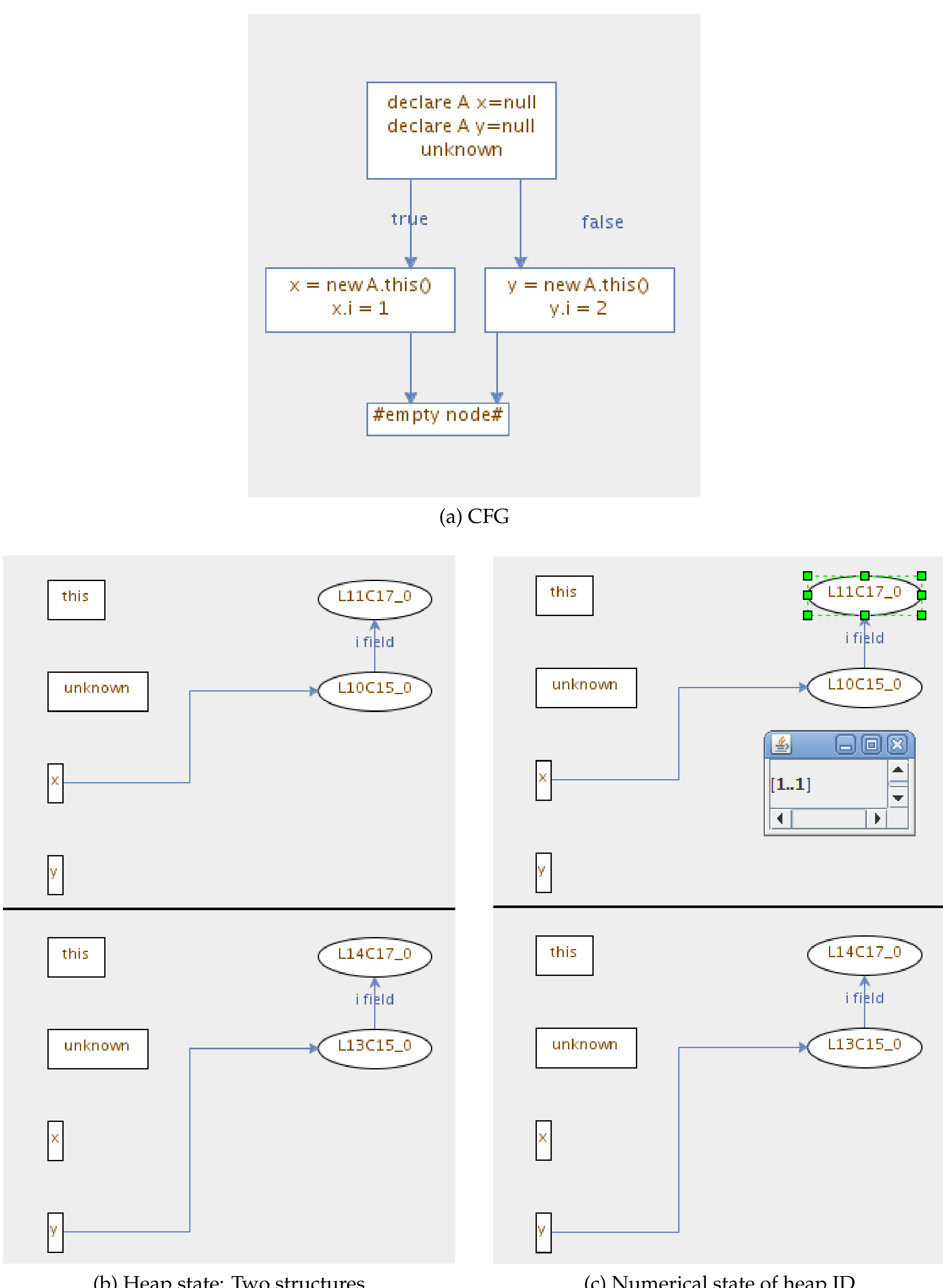
Non-interactive Visualization

Sometimes the results of an analysis should be saved for later use in a non-interactive fashion, such as when analyzing dozens of methods. In that case, we generate images with graphs of all heap structures in the end state of the program.

Automatically layouting large graphs well is a difficult endeavour, so we chose to use `Graphviz DOT` for that task. Figure 7b shows the rendered graph of the heap state after executing Figure 7a. Apart from minor differences in appearance, the rendered graph corresponds to the graphical notation for heaps described in Section 2.4. That is, the program variable `x` references the first heap node that was created, namely `L5C13_0`, while the field `n` of the later references node `L6C11_0`.

3.2 Performance and Optimization

As outlined in Section 2.1, we decided to use TVLA as an external tool. After some experimentation it soon became evident that our analysis suffered from severe perfor-



(a) CFG

(b) Heap state: Two structures

(c) Numerical state of heap ID

Figure 6: Sample GUI visualization

```
class A {
  var i = 0;
  var n: A = null
}

object SomeObject {
  def someMethod(unknown: Boolean) = {
   var x,y : A = null
   if (unknown) {
     x = new A
     x.i = 1
   } else {
     y = new A
     y.i = 2
   }
  }
}
```

Listing 17: Scala code of GUI visualization example

```
class A { var n: A = null }

object SomeObject {
  def someMethod = {
    val x = new A
    x.n = new A
  }
}
```

(a) Program code

(b) Heap end state

Figure 7: Graphviz DOT visualization

mance problems. These were to a large part due to the start-up of a new JVM for every call to TVLA. We then tried to invoke the `main` method of the TVLA java library directly in order to save the process creation and JVM initialization time. However, this did not work since some static data internal to TVLA was not reset. The default binary distribution of TVLA is designed to be used as a standalone application to analyse whole programs, which is in sharp contrast to our usage.

Upon request to the developers, we were able to obtain a customized version of TVLA which allowed us to reset the internal state after one execution and then call the `main` method again. This lead to a reduction of the execution times of about 80% in general.

Furthermore, we found the iteration performed by Sample to execute a program in the abstract domain to be inefficient. While this may not be critical for other analyses, it has a huge impact on our heap analysis. The problem is that Sample frequently executes the same statement in the CFG several times on the same pre-state. While the result obviously does not change, we perform a full run of TVLA. To improve the performance, we cached the results of previous heap operations on our heap domain: Whenever our domain is requested to perform an operation, we check if such an operation was previously applied to the same heap state. If that is the case, we can return the cached result. We observed a 60% reduction of the execution times on average. The memory usage for the caching seems to be negligible compared to the memory footprint of TVLA.

3.3 Testing

To assess the implementation of our domain, we developed two test-suites which consist of Scala methods. We verified by manual inspection for each test case that the results are indeed safe approximations of the real execution behaviour.

We ran Sample with two domains, our new `TVSHeap` heap domain and a non-relational numerical domain which uses interval abstraction. All analyzed examples are written in Scala, therefore we used the Scala compiler back-end to translate the source code to `Simple`.

The analysis was run on a machine with a Intel Core 2 Duo at 2.53GHz and 4GB of RAM. We used the Java HotSpot 64-Bit Server VM included in Java SE Runtime Environment 1.6.0_26-b03 with the option `-XX:NewRatio=2`, which helped us to improve garbage collection performance.

Table 2 contains all the experimental results. Column **#tr** shows the number of times TVLA was called during the analysis, while column **#tr-unique** shows the actual calls performed when caching of the results was enabled. The running times are displayed in columns **JVM-nc** (JVM invocation method without caching), **JVM** (JVM with caching) and **MR** (`main+reset` invocation method with caching).

Note that the `main+reset` invocation method produced some invalid results for the test cases `initializeFixedList`, `createNumericalList` and `initializeAbstractedListFields`. This is most likely due to a bug in the customized binary TVLA version that we obtained from the TVLA developers.

Testcase	# tr	# tr-unique	JVM-nc [s]	JVM [s]	MR [s]
createObject	12	4	4.9	2.0	0.7
createAndOverWrite	26	9	9.9	3.7	0.7
assignNextField	26	9	10.4	3.9	0.6
accessNullField	20	7	8.0	2.7	0.3
assignFieldSelf	18	7	6.9	2.5	0.3
overwriteField	24	10	8.8	3.8	0.4
createObjectIfCondition	24	7	9.0	2.7	0.2
conditionalAssignment	30	11	11.4	4.4	0.4
conditionalAssignmentVariant	49	14	18.1	5.9	0.6
assumeEqual	35	10	12.6	3.9	0.2
assumeUnequal	32	13	11.8	5.1	0.3
assumeUnequal2	36	10	12.7	3.8	0.2
createSharedObject	50	19	19.8	8.1	0.8
linkObjects	84	32	34.8	13.8	1.4
linkAndTraverseObjects	104	38	44.2	16.1	1.9
createObjectWhile	100	39	38.0	15.8	1.1
appendByFieldAccessTwo	42	15	16.1	5.9	0.6
buildList	52	20	20.1	7.4	0.5
appendByFieldAccessThree	60	22	23.5	8.9	0.9
appendByFieldAccessFour	112	72	116.0	73.6	4.7
createPrependList	129	54	52.2	22.1	1.4
createAppendList	178	103	76.7	47.3	3.3
swapLoop	126	34	49.6	12.5	0.9
accessNextSummarized	60	23	22.3	9.6	0.6
traverseFixedShortList	247	93	104.0	48.7	3.3
traverseSummarizedList	331	164	149.2	89.1	8.6
assignNumericField	22	8	9.9	5.0	1.3
assignAndAccessNumericField	28	10	11.9	4.4	1.0
createThreeElementList	78	30	36.2	13.8	2.0
createSummarizedIntList	82	32	37.8	15.1	2.0
assignTwoFields	32	12	14.8	5.9	0.9
assignFieldsAndSummarize	112	44	58.5	23.6	4.1
assignAndAddFields	170	55	102.2	32.4	8.3
createOneOrTwoNodes	112	22	49.8	9.9	1.8
swapHeapObjectsOnce	85	24	37.2	11.2	1.8
initializeFixedList	370	109	183.8	57.9	4.5
createNumericalList	302	148	151.7	79.9	14.4
initializeAbstractedListFields	476	220	253.2	144.8	31.8
sumListElementsZero	893	609	683.3	524.7	86.7
			2521.4	**1351.9**	**195.4**

Table 2: Non-numerical and numerical testsuite execution times

```
def conditionalAssignmentVariant(unknown: Boolean) = {
  val x = new A
  val y = new A

  val z = if (unknown) x else y
}
```

Listing 18: `if`-branches and heap structures

3.4 Representative Testcases

In this section, we present the analysis results of a few notable test cases, representative of what we were able achieve with the new domain. Note that the Scala code of our test cases looks rather imperative than functional because Sample's support for interprocedural analysis is incomplete. For that reason, we did not use any method calls.

Sometimes heap node names in the scheme we proposed tend to become too long, so we decided to abbreviate them in the resulting graphs for the sake of clarity. We provide the names and contents of the numerical domain separately.

3.4.1 Multiple Structures

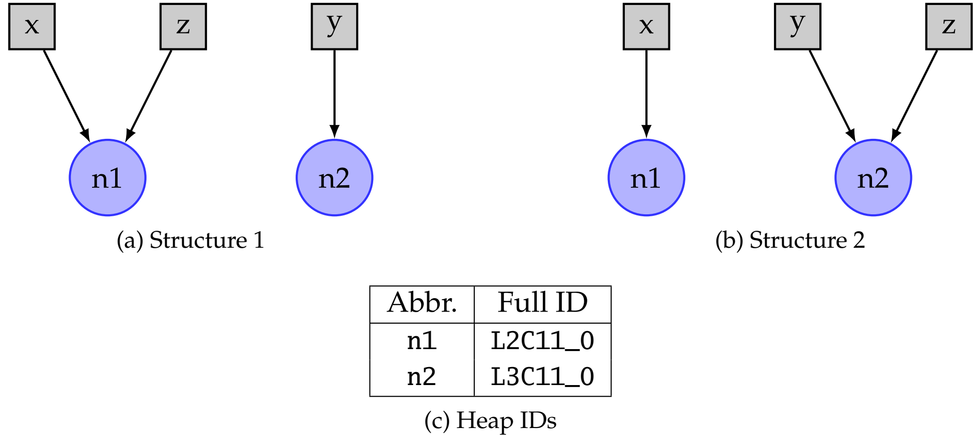
Our first example in Listing 18 illustrates how our heap domain increases the number of three-valued structures to represent the heap when necessary. This usually happens when previously abstracted (summarized) parts of the heap are concretized again, and when paths of the control flow join. Here we show the later case.

After both branches of the `if`-statement were evaluated, we take the least upper bound of the two states, i.e. we take the union of both structures and move on to perform the assignment of the expression result. Canonical abstraction determines that the structures cannot be combined into one, since they do not agree on the value of the unary predicate for `z`. In one structure, it points to the object referenced by `x`, while in the other one it points to the one referenced by `y`. The result is displayed in Figure 8.

It is also worth mentioning that whenever we need a condition that cannot be determined statically, we use a boolean variable unknown passed as a parameter to the method. We noticed that the Scala compiler sometimes is smart enough to eliminate branches if conditions are too simple (e.g. depending only on constants).

3.4.2 Abstraction: Creation of a List

In the next example, things get more interesting: The program in Listing 19 creates an unbounded number of heap objects. More precisely, it creates a singly-linked list of one or more elements whose head is referenced by the variable x. It uses a loop to allocate new objects and prepend them to the existing tail.



(a) Structure 1

(b) Structure 2

Abbr.	Full ID
n1	L2C11_0
n2	L3C11_0

(c) Heap IDs

Figure 8: Multiple Heap Structures: Result

```
1  def createPrependList(unknown: Boolean) = {
2    var x = new A
3    var t:A = null
4
5    while (unknown) {
6     t = new A
7     t.n = x
8        x = t
9     t = null
10   }
11 }
```

Listing 19: List creation

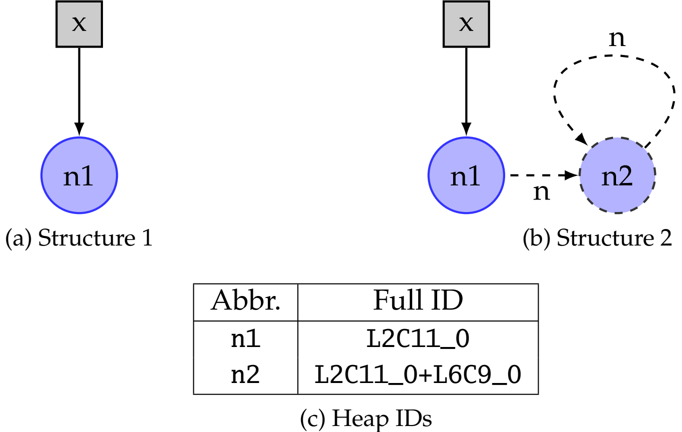
Again, for technical reasons, we used the variable unknown to model a condition that cannot be decided. One could also think of it as a random variable which decided whether the (concrete) program keeps executing the loop or stops.

It could be that the `while`-loop body is never executed: In that case, we get a list of length one. However, if we keep executing the loop, abstraction takes place to keep the heap bounded: All elements beyond the head are summarized into a single one, since they look the same to TVLA (regarding the predicate values). Therefore, a summary node is created which stands for one or more elements.

Figure 9 displays the abstract heap state after the method was executed. It contains a structure for each of the described cases.

3.4.3 Traversing Lists

The last testcase showed how abstraction is performed. However, what is much more difficult is how to obtain precise information when accessing parts of the heap which



(a) Structure 1

(b) Structure 2

Abbr.	Full ID
n1	L2C11_0
n2	L2C11_0+L6C9_0

(c) Heap IDs

Figure 9: List creation result

```
def traverseSummarizedList(x: AcyclicList) = {
  // variable which afterwards references the last element
  var end : AcyclicList = null

  var cur = x
  while (cur != null) {
    end = cur
    cur = cur.n
  }
}
```

Listing 20: List traversal testcase

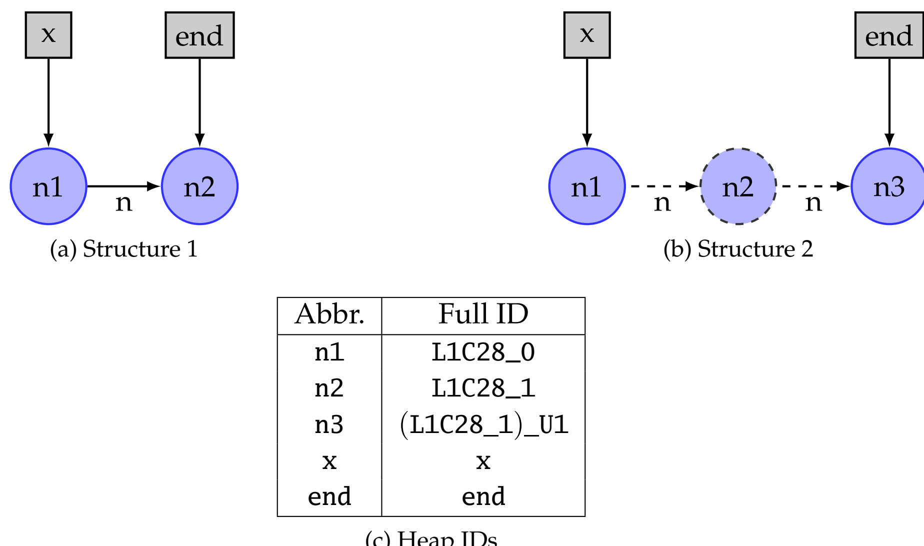
were summarized.

The input state at the start of the method in Listing 20 consists of a singly-linked list of length two or more. With instrumentation predicates added (for reachability and sharing), we were able to successfully traverse the list. We let a reference move along an arbitrary long (since abstracted) list, pointing to the last element in the end (Figure 10).

This testcase also shows that our analysis actually makes use of branching conditions: When the loop body is entered, we let the heap domain assume the condition is true, if it is not entered we assume the negation. This can be seen from the fact that cur is `null` in the end state.

3.4.4 Integer Fields and the Numerical Domain

In the absence of summarization, it is desirable not to lose any precision. The testcase in Listing 21 demonstrates that we can assign and access fields without any precision loss since all the objects are referenced by program variables: As can be seen in the result



(a) Structure 1

(b) Structure 2

Abbr.	Full ID
n1	L1C28_0
n2	L1C28_1
n3	(L1C28_1)_U1
x	x
end	end

(c) Heap IDs

Figure 10: List traversal result

in Figure 11, the sum of all integer fields is exactly 21. We also see how the interval domain tracks the numerical information associated with the integer-valued fields `i` and `j`.

3.4.5 Replacements in Action

We introduced the concept of replacements to update the numerical domain when the identifiers change. Here we consider the (admittedly artificial) case where two objects are created with integer fields of different value. The references to these objects are then swapped depending on a condition. At the end of the `if`-statement in Listing 22, the least upper bound of the original and the swapped heap states is taken. While in the concrete, the structures of these heap states are not equivalent, they look structurally the same to TVLA: In both cases, there are two variables each pointing to a node with an integer field. Therefore, they are joined into a single one.

Our mechanism for names is able to detect this and creates a replacement to merge the numerical values of the fields, i.e. both fields now have the numerical value [1..2], since the distinction between the unswapped and swapped cases was lost. The end state is depicted in Figure 12.

3.4.6 Initialize and Sum Lists

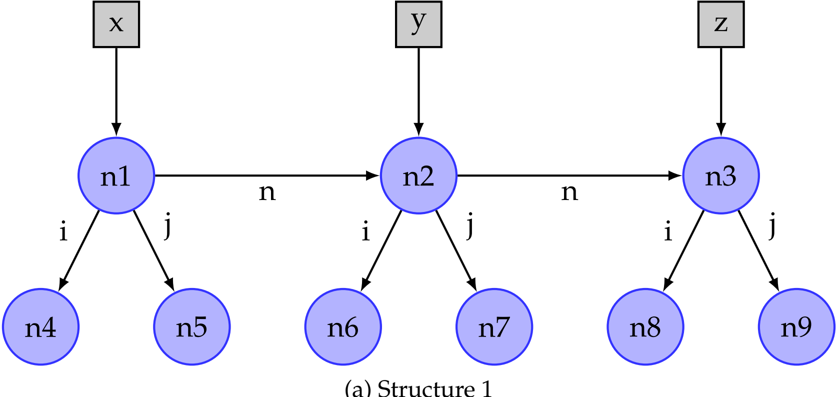
In Listing 23 we show the most challenging testcase we considered: A singly-linked list is initialized with all `i` fields set to 0. We then traverse the list and sum up all the elements we have seen and are able to deduce that the sum is exactly 0.

```
class TwoIntNode {
  var n: TwoIntNode = null // next
  var i: Int = 0
  var j: Int = 0
}

def assignAndAddFields = {
  val x = new TwoIntNode
  x.i = 1; x.j = 2
  var y = new TwoIntNode
  y.i = 3; y.j = 4
  var z = new TwoIntNode
  z.i = 5; z.j = 6
  x.n = y;
  y.n = z

  val sum = x.i + x.j + x.n.i + x.n.j + x.n.n.i + x.n.n.j
}
```

Listing 21: Field assignment testcase

```
def swapHeapObjectsOnce(unknown:Boolean) = {
  var x = new IntNode
  x.i = 1
  var y = new IntNode
  y.i = 2
  var t: IntNode = null

  if(unknown) {
    t = x
    x = y
    y = t
    t = null
  }
}
```

Listing 22: Swap fields testcase

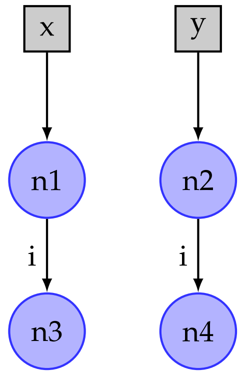


(a) Structure 1

Abbr.	Full ID	Semantic Domain
n1	L8C11_0	⊤
n2	L10C11_0	⊤
n3	L12C11_0	⊤
n4	L9C9_0	[1..1]
n5	L9C18_0	[2..2]
n6	L11C9_0	[3..3]
n7	L11C18_0	[4..4]
n8	L13C9_0	[5..5]
n9	L13C18_0	[6..6]
x	x	⊤
y	y	⊤
z	z	⊤
sum	sum	[21..21]

(b) Heap IDs and semantic state

Figure 11: Numerical fields: Result



(a) Structure 1

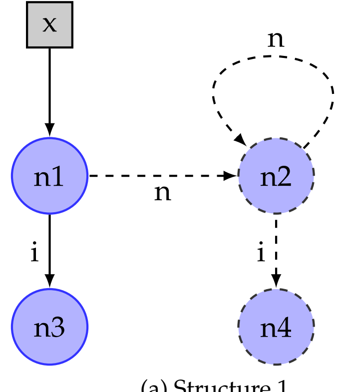
Abbr.	Full ID	Semantic Domain
n1	L2C11_0+L4C11_0	⊤
n2	(L2C11_0+L4C11_0)_U1	⊤
n3	L3C9_0+L5C0_0	[1..2]
n4	(L3C9_0+L5C9_0)_U1	[1..2]
x	x	⊤
y	y	⊤

(b) Heap IDs and semantic state

Figure 12: Swapping numeric fields: Result

```
def sumListElementsZero = {
  x = /* code to build and initialize a list with fields set to 0 */

  // traverse the list and sum up elements
  var cur = x
  var sum = x.i
  while (cur != null) {
    sum += cur.i
    cur = cur.n
  }
}
```

Listing 23: Sum list elements testcase



(a) Structure 1

Abbr.	Full ID	Semantic Domain
n1	L194C17_0	$\top$
n2	L196C17_0+L198C17_0	$\top$
n3	L195C15_0	[0..0]
n4	L197C15_0+L199C15_0	[0..0]
x	x	$\top$
sum	sum	[0..0]

(b) Heap IDs and semantic state

Figure 13: Summing up list elements: Result

4 Conclusions

We successfully implemented a new heap analysis in Sample which is much more powerful than the existing ones. Moreover, since it relies on TVLA, it is parametric in the predicates used, allowing different choices to be made if necessary.

Our analysis yields precise results for programs that manipulate singly-linked lists and integers, which we demonstrated in the given examples written in Scala. However, when other data structures are involved, the abstraction often becomes imprecise very quickly. We are under the impression that TVLA does not scale well to general programs. All the examples shipped with its binary distribution contain very specific predicates tailored towards a given type of program to analyze.

4.1 Known Issues

Currently there is an open issue concerned with typing of the heap nodes: We do not encode any type information in TVLA; i.e. all our heap nodes are untyped. It is therefore possible that heap nodes of different type are summarized, leading to imprecision. Instrumentation predicates could be used to consider types, but doing so is not within the scope of this thesis, since it is a highly non-trivial task given the complex type system of Scala and Simple.

4.2 Future Work

As shown above, we used the interval domain as the semantic domain to evaluate our analysis. However, in principle our design allows any semantic domain of Sample to be plugged in. It would be interesting to use such domains, e.g. the existing one for string analysis, and see whether it produces precise results. It seems likely that our heap analysis improves the precision of the whole analysis.

Furthermore, different instrumentation predicates should be added to analyse other common data structures apart from singly-linked lists. Adding more instrumentation to the existing ones while the performance is not degraded severely will be a major challenge.

References

[CC77] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM, 1977.

[CC79] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 269–282. ACM, 1979.

[CFC11] G. Costantini, P. Ferrara, and A. Cortesi. Static analysis of string values. In *International Conference on Formal Engineering Methods (ICFEM)*, volume 6991 of *Lecture Notes in Computer Science*. Springer, October 2011. To appear.

[Fer10] P. Ferrara. Static type analysis of pattern matching by abstract interpretation. In *Formal Techniques for Distributed Systems (FMOODS/FORTE)*, volume 6117 of *Lecture Notes in Computer Science*, pages 186–200. Springer-Verlag, 2010.

[LAIS06] T. Lev-Ami, N. Immerman, and M. Sagiv. Abstraction for shape analysis with fast and precise transformers. In *Computer Aided Verification*, pages 547–561. Springer, 2006.

[LAS00] T. Lev-Ami and M. Sagiv. Tvla: A framework for kleene logic based static analyses. *Master's thesis, Tel Aviv University*, 2000.

[MSRF04] R. Manevich, M. Sagiv, G. Ramalingam, and J. Field. Partially disjunctive heap abstraction. *Static Analysis*, pages 159–412, 2004.

[SRW02] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 24(3):217–298, 2002.

[WSR00] R. Wilhelm, M. Sagiv, and T. Reps. Shape analysis. In *Compiler Construction*, pages 1–17. Springer, 2000.

A Detailed List of TVLA Actions

```
/**
 * All the Actions as declared in the TVP file passed to TVLA
 *
 * NOTE: Above we omitted the instrumentation predicate updates
 *       and some details for simplicity. Here we provide the
 *       complete listing. The actions are loosely based on the actions
 *       for the SLL example of the downloadable TVLA archive.
 */

// Action for "target = null"
%action setVariableNull(target) {
  { target(v) = 0 }
}

// Action for "target = source" (variable assignment)
%action copyVariable(target,source) {
  %f { source(v) }

  { target(v) = source(v) }
}

// Action for "target.field = null."
%action setFieldNull(target,field) {
  %f { target(v) }

  {
    field(v_1, v_2) = field(v_1, v_2) & !target(v_1)
  }
}

// Action for "target.field = source."
%action setField(target,field,source) {
  %f { target(v), source(v) }

  {
    field(v_1, v_2) = field(v_1, v_2) | target(v_1) & source(v_2)
  }
}

// Action for "destination = target.field"
%action extractField(destination,target,field) {
  %f { E(v_1,v_2) target(v_1) & field(v_1,v_2) & t[field](v_2,v) }

  {
    destination(v) = E(v_1) target(v_1) & field(v_1, v)
  }
}

// Action for "target = new Object"
%action createObject(target) {
```

```
51    %new
52    {
53      target(v) = isNew(v)
54      foreach (f in Fields) {
55        t[f](v_1, v_2) = (isNew(v_1) ? v_1 == v_2 : t[f](v_1, v_2))
56        is[f](v) = is[f](v)
57        r[f, target](v) = isNew(v)
58        foreach(z in PVar-{target}) {
59          r[f,z](v) = r[f,z](v)
60        }
61      }
62    }
63  }
64
65  // Action for assumptions "if(v == null) ..."
66  %action assumeVariableNull(var) {
67    %f { var(v) }
68    %p !(E(v) var(v))
69
70    { }
71  }
72
73  // Action for assumptions "if(v != null) ..."
74  %action assumeVariableNotNull(var) {
75    %f { var(v) }
76    %p E(v) var(v)
77
78    { }
79  }
80
81  // Action for assumptions "if(var1 == var2) ..."
82  %action assumeVariableEqual(var1,var2) {
83    %f { var1(v), var2(v) }
84    %p A(v) var1(v) <-> var2(v)
85
86    { }
87  }
88
89  // Action for assumptions "if(var1 != var2) ..."
90  %action assumeVariableNotEqual(var1,var2) {
91    %f { var1(v), var2(v) }
92    %p !A(v) var1(v) <-> var2(v)
93
94    { }
95  }
96
97  // Action for least upper bound of two heap states
98  %action lub() {
99
100 { }
101 }
```

Listing 24: All used TVP actions