Titel:

Shape Analysis for Algorithm Animation

Strengths and Weaknesses

Dissertation
zur Erlangung des Grades des
Doktors der Ingenieurwissenschaften
der Naturwissenschaftlich-Technischen Fakultäten
der Universität des Saarlandes

von
Sascha A. Parduhn

Saarbrücken
28.11.2011

Tag des Kolloquiums:	21.12.2011
Dekan:	Prof. Dr. H. Hermanns
Berichterstatter:	Prof. Dr. R. Seidel
	Prof. Dr. Dr. h.c. mult. R. Wilhelm
Vorsitz:	Prof. Dr. H. Hermanns
Akad. Mitarbeiter:	Dr. M. Neuhäuser

Acknowledgment

First and foremost my thanks go to the professors Raimund Seidel and Reinhard Wilhelm. Their help and insights were invaluable for the progress of my work. But even more valuable than their considerate professional skills was the trust and patience they showed me by giving me a second chance. It was this trust that made this work possible in the first place. For that I will be forever grateful.

I also want to thank Mooly Sagiv, Roman Manevich, Tal Lev-Ami and the rest of the gang that implemented TVLA and were always available when problems arose.

Writing a phd-thesis is a long and sometimes frustrating process and I want to thank my fellow co-workers for their help in getting me through the bad times as well as sharing the good times. My special thanks go to Dierk Johannes for his patience while listening to my ideas, as well as for his mathematical expertise. I also want to thank Christian Hoffmann for the long hours of discussion, both professional and spiritual. I may not have found redemption yet, but you set me on the right way.

Finally I want to dedicate one paragraph to the people that keep the day to day infrastructure working. Since I also had to care for the server and net infrastructure of the chair of theoretical computer science, I learned quite a bit about all the tedious little problems that crop up every day. It is a thankless job to keep things working, because if you do a good job, no one will notice and if things go bad, you get blamed. So this is to you, the Rechnerbetriebsgruppe, for always helping me out, fast and unbureaucratic.

Abstract

We present a non-traditional approach to algorithm visualization that is based on shape analysis, a static program analysis, that so far has been mostly used to verify program properties. Our approach differs from traditional visualizations: Instead of simulating a concrete program execution, we use the invariants produced by shape analysis to look at all possible program executions at the same time. This allows us to show meta level properties like correctness very easily, but at the price of a higher level of abstraction than most traditional visualizations. For example when sorting we do not model individual data values, but instead maintain a relation that tells us if a data value is smaller than another.

Shape analysis is not a new technique, but its application to algorithm visualization is: The analysis output can quite naturally be interpreted as a set of graphs, however, suitable presentation of the shape analysis output requires preparation of the data. This includes methods to reduce the size of the analysis output and methods to make it easier to understand. The inherent abstraction also raises new problems. We discuss these problems and how they influence the layout of our visualization. We also give guidelines for creating analyses specifically with visualization in mind and provide a detailed description of an example analysis, that was created according to those guidelines. Lastly we implemented a visualization tool that was used as a testbed for many of the methods mentioned in the thesis and we will present a short overview of its features.

Zusammenfassung

Wir präsentieren eine unübliche Herangehensweise an Algorithmenvisualisierung, die auf Shape Analyse basiert, eine statische Programmanalyse, die bisher hauptsächlich zum verifizieren von Programmeigenschaften verwendet wurde. Unsere Herangehensweise unterscheidet sich von traditionellen Visualisierungen: Statt eine konkrete Programmausführung zu simulieren, verwenden wir Invarianten, die von Shape Analyse produziert wurden, um alle möglichen Programmausführungen gleichzeitig zu betrachten. Dies erlaubt es uns Meta-Eigenschaften wie Korrektheit sehr einfach zu zeigen, aber zum Preis einer höheren Abstraktionsebene als in vielen herkömmlichen Visualisierungen. Zum Beispiel beim Sortieren modellieren wir keine individuellen Datenwerte, sondern benutzen eine Relation, die uns sagt ob ein Datenwert kleiner ist als ein anderer.

Shape Analyse ist keine neues Verfahren, aber ihre Anwendung zur Algorithmenvisualisierung ist es: Die Ausgabe der Analyse kann relativ natürlich als Menge von Graphen interpretiert werden, aber, für eine sinnvolle Darstellung der Ausgabe müssen die Daten erst aufbereitet werden. Dies beinhaltet Methoden die Größe der Ausgabe zu reduzieren und Methoden die Verständlichkeit zu erhöhen. Die inhärente Abstraktion wirft auch wieder neue Probleme auf. Wir erörtern diese Probleme und die Art und Weise wie sie unsere Visualisierung beeinflussen. Wir geben außerdem Richtlinien, wie man Analysen ganz spezifisch zum Zwecke der Visualisierung erstellt und geben eine Beschreibung einer Beispielanalyse. Wir haben auch ein Visualisierungswerkzeug programmiert, mit dem viele der Methoden in dieser Arbeit getestet wurden und das wir kurz erläutern.

Eidesstattliche Versicherung:

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Die aus anderen Quellen oder indirekt übernommenen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet.

Die Arbeit wurde bisher weder im In- noch im Ausland in gleicher oder ähnlicher Form in einem Verfahren zur Erlangung eines akademischen Grades vorgelegt.

Saarbrücken, den   Unterschrift

Contents

1 Introduction

1.1 The Importance of Visualization

“Seeing is believing” is an idiom that was first recorded in this form in the seventeenth century. It means that in order to be convinced of something, humans often need physical, concrete (i.e. visual) evidence. The idiom has since been used as title for books[1], films[2], novels, songs[3] and even is the name of an organization[4] that aims to avoid blindness. The deeper meaning of the idiom, however, can be traced much further back. There are multiple mentions of this in the bible, one of the most famous ones is probably the story of the doubting Thomas, where the apostle Thomas refuses to believe in Jesus resurrection, till he was allowed to touch the wounds. Afterwards he proclaimed his faith in Jesus, to which the latter replied: “Thomas, because thou hast seen me, thou hast believed: blessed are they that have not seen, and yet have believed.”, King James Bible, John 20:29.

The above shows, that human sight obviously has a special, very important place in the human mind. This impression is further confirmed if we look at all the essentially useless things, that persons do, only to improve their visual appearance: Any kind of cosmetics, fancy clothes, even if they are not very practical, hair styles, body training, just to look “good”, tattoos, jewelry and many more. Those were only some of the examples that apply directly to the human person. In reality, almost all consumer products also always have the requirement to look “good” or they may be perceived to be qualitatively worse than a better looking product.

Why is human vision so special? There are a number of interesting facts: Sight is the sense with the highest information bandwidth by far! No other sense can match the sheer volume of possible input that comes from our vision. Just imagine reading a book, it is possible to read the book much faster if you do not vocalize the words than if you read them aloud. Now imagine watching a movie, how fast can you translate the layout, colors, emotions into other senses?

Human vision is fast and parallel (see Healey and Enns [11]). There seems to be neurons in the eyes and brain, that are responsible for different kinds of information: color, shape, orientation, texture, movement. All of these factors can be recognized independently from each other and very fast. All these factors can, in fact, be processed and recognized faster than the eye is able to focus! Tests (see Sagi and Julész [28], Treisman [33, 34]) have shown, that even when showing a picture only for less than 250 milliseconds (which is the time required to focus the eye on a certain region) it is still possible to see if there is an abnormal color in an otherwise homogeneous picture, or if there is a circle in a picture full of squares. It should be noted though, that there are limits to the number of different colors, shapes, textures etc. that can be recognized at

[1] Searching for “seeing is believing” at www.amazon.com gives 1854 hits. The books range from scientific or semi-scientific, detective stories, christian youth books to even erotica. Anything from short stories to novels seems available.
[2] Searching for movies at www.amazon.com gives 19 results.
[3] Again www.amazon.com gives 84 hits.
[4] The homepage of the organisation is at www.seeingisbelieving.org.uk

once and these limits are usually very low. Due to the speed of the recognition, this is often called pre-attentive processing, because there is no time to focus consciously on the tasks.

Human vision is good at pattern recognition! Our visual memory is very good at recognizing known features, for example human faces. It might thus seem strange that we sometimes have problems recognizing differences in animals or sometimes even humans in other cultures with slightly different facial features. This is due to the fact that we can indeed focus so much an what we expect, that we are literally blind to even very big changes, that are obvious if someone points us to it. This phenomenon is called *change blindness*. It is easily demonstrated by having a short movie clip, that shows a picture, then a short black screen and then the same picture again, just with one big difference and then loop the movie. The black screen in between these two pictures kind of resets the eye and when we see the next picture it seems to look exactly the same, even if the difference is very obvious once pointed out! More about this phenomena can be found in Mack and Rock [23], Rensink [25, 26].

Human vision helps cognition and extends our memory! To understand how visualization can help the cognition of a problem, just consider a mathematics lesson. Imagine the exercise is to prove a statement by transforming one formula to another. There might be a scant few that can do it without writing down the formula and the various transformation steps, but many will have to take the proof line by line. They might have to look at the formula to get an idea of how to transform it in the next step. While this may also already count as extending the memory, a better example might be the simple school multiplication of two non-trivial numbers: Depending on the size of the numbers, it may not be possible to multiply them without writing the process down. A more trivial example is looking up a cooking recipe or writing down a shopping list.

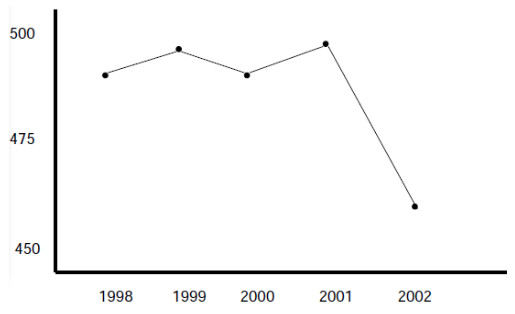
With so much emphasis in our brain for vision, it is not surprising, that visualization always has been of great importance in human society. And because of the last point, it is also widely used in teaching and science in general. For a more comprehensive work on the impact of human vision and visualization, see Healey and Enns [11].

1.2 Algorithm Animation and Information Visualization

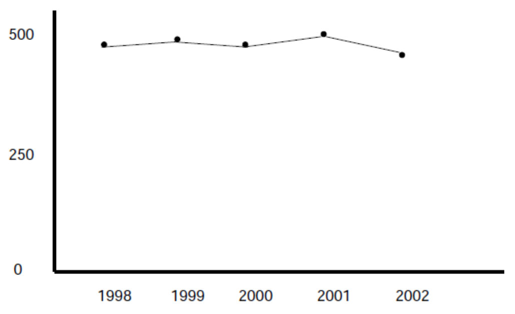
In scientific visualization the purpose is not to produce pretty pictures, but to help understanding. We want to help the understanding of algorithms in undergraduate courses like Algorithms and Data Structures, which is a very important basic course for any computer science student. It is therefore obvious, that our main focus lies on algorithm visualization, but, due to the nature of our work, there is also a bit of information visualization involved.

Information visualization is concerned with efficiently portraying possibly huge amount of data in such a way that important information can be easily extracted. As a simple example, let us assume we are interested in comparing some statistics of federal states in the US. We want to see the percentage of collage degrees in the population and the per capita income in each state.

In the census bureau website you can freely get this data, but you only get shown one



(a) This might be a development on the stock exchange.



(b) The same information as in (a). Seen with the right scale.

Figure 1: Here we see the same information displayed in two different ways. In (a) it might look like a stock market crash, while in (b) we can see the whole scale and thus can deduce correctly, that it is just a slight decline. The above diagrams are purely fictional and only serve to illustrate how changing the baseline may affect the perception of how serious the stock price drop is.

single state at a time. This already makes it hard to compare two states, even more so if you compare all of them. The next step might be to put all the data in a table. If the table is unsortable, then it is again very hard to even answer simple questions like what is the state with the highest per capita income or the state with the lowest percentage of college degrees.

In a sortable table these questions can be answered quickly by sorting according to the key in question. But if we want to know if there is a correlation between college degrees and per capita income, then it becomes very hard again. Even more so, if you ask questions like are there outliers in the general correlation.

It is much more efficient to visualize the table as a graph. That way a general correlation can easily be seen, as well as any outliers that may exist, i.e., states where there is a low number of people with a college degree, but still a very high per capita income.

The simple example shows, that visualization is done to help cognition and to answer concrete questions. Information visualization could therefore be described as taking items without a direct physical correspondence (raw data) and mapping them to a 2-D or 3-D physical space with the explicit aim, to give information a visual representation that is useful for analysis and decision-making.

One important thing to keep in mind for visualization, be it information visualization or algorithm visualization, is to avoid telling "*graphical lies*", a term coined by Edward Tufte. Consider Figure 1(a) as an example of such a lie. Due to the changed scaling (y-axis starts at 450 instead of 0), the drop looks a lot more severe then it actually is, if the whole scale is considered, as shown in Figure 1(b).

Algorithm animation is a wide field and due to its sheer size and the many kinds of methods for visualization, it is not easy to give a comprehensive overview. In Kerren and Stasko [16] the authors give an overview of the state of the field and while it is by no means comprehensive it shows how many sides there are to algorithm animation. I will shortly touch upon the most important points.

They start with the “classic” visualizations, the most well known is probably the “sorting out sorting” by Baecker [2] where nine sorting algorithms are shown in parallel, to demonstrate their runtimes on certain input.

They then distinguish visualizations by the specification method used, i.e. how the visualization was obtained in the first place:

- One common example is that the creator of the visualization constructs some mapping between the program states and the visualization states. An extreme example of this would be a static visualization that always starts with the same input and thus always shows the same steps of the algorithm. In that case it would mean nothing more than playing back a recording. Usually though, it is a bit more interactive or at least randomized to generate different program executions. A classic example of such a system is `PAVANE` by Roman and Cox [27].

- Another method to obtain a visualization is an idea that stems from debugging: The programmer or visualizer can put marks at certain program points and if these program points are reached, then a visualization will be generated based on program state and certain parameters the programmer can specify. In other words, the creator of the visualization can decide when an *interesting event* happens and what should by visualized when it happens. This approach was pioneered in `BALSA` by Brown [4] and its successor `ZEUS`, see Brown [5]. The method is aptly called *event driven.*

- The next method mentioned is visual programming. This might seem odd at first, since strictly speaking, visual programming is usually not considered algorithm animation. But the point of visual programming is to make programs easier to specify by using visual representations for program commands. So in a way, it is visualization backwards, using the visual to create the program instead of the other way around. Still there is a clear visual abstraction for program statements, and thus it can also be considered visualization.

- The last method mentioned is *automatic animation.* While undoubtedly the easiest method for the creator of the visualization, total automatic creation of animations is extremely difficult, see Brown [6]. Thus systems have been created that offer different amounts of automation. There is always the trade off between how much work must be put into the specification and how versatile the system can be. It is intuitively clear that it is much easier to automatically generate a visualization if the application is narrowed down to a certain algorithm with certain data structures, than if there are many possible data structures and algorithms. In other words, the more powerful the tool, the harder it becomes to specify.

Next the authors of Kerren and Stasko [16] distinguish algorithm animations by the visualization techniques used. Due to the near limitless possibilities here, it is again not possible to be comprehensive, but there are some basic distinctions that can be made:

- 2D versus 3D is always a somewhat difficult and heated topic when talking about visualizations. There are strong believers for either side, yet the truth, as so often, is somewhere in the middle. The obvious advantage of 3D is the extra dimension and thus the additional "space" to show more information than what is possible in 2D space. This advantage comes with several disadvantages though, the most obvious being the occlusion of objects, i.e. an object is blocked from view by one or more other objects that are "in front" of it, relative to the camera position. Another disadvantage is the possibility to "get lost" in 3D space, i.e. losing the mental map of where in the 3D space the camera is at the moment. While the world we live in certainly is a 3D space, the way our orientation works is by reference points and the sense of where up and down is, i.e. gravity. That means that while it may be relatively easy to run around inside a virtual 3D model of a house without getting lost, the same is not true if we have to navigate through a black space where some sheets of papers are floating in the great big nothing. This can be compared to the disorientation of divers, when they lose all reference points, thus not even knowing which way the surface is.

- Although again not strictly visualization, there is the possibility to use audio to reinforce and sometimes replace visual cues. It can also be used for pattern recognition, the pattern forming a kind a melody, that makes it easy to notice interruptions in the pattern and that in turn allows to easily find some kinds of programming errors. The most common use that most computer users are familiar with, is the possibility to use sound cues to mark exceptional conditions, such as the infamous beep sound of the internal speakers or the various "ding" sounds of certain operational systems. `MUSE` by Lodha, Beahan, Heppe, Joseph, and Zane-Ulman [22] is one example for a system that allows the mapping of musical sounds to any kind of scientific data.

- There is also a special category of visualizations, that are specifically developed as web applications. While these have of course much the same problems as any other category, they also have some additional ones, like having to be content with less space, a more primitive user interface and all the other little nitpicks of having to run in a browser window. One big issue is of course platform independence. This is the reason why the overwhelming majority of tools for web animations are either directly Java based, or they use some scripting language and have a Java interface to the browser. `JHAVÉ` (by Naps, Eagan, and Norton [24]) is one example of such a tool. One might wonder if these visualizations even deserve an extra category, but the enormous popularity, or at least, the enormous quantity of these visualizations speak for themselves: At the time of this writing, AlgoViz.org, a web portal that collects algorithm animations, counts 540 animations, of those, 303 are classified as java applets, 95 as java applications and another 56 as either java web start or java script.

There are many more ways to distinguish between visualizations:

- Are they interactive or not? Often interaction is considered to be "better" for learning than just looking at an animation. But even between interactive animations there are differences: Is the user just allowed to give the algorithm some input and then watch what happens? Or is the visualization more an exercise, that presents the users with a situation and then expects them to correctly predict what will happen?

- Is the visualization purely graphical, or is there also the possibility to add comments, annotations, whole explanations in help files that can be displayed depending on the situation?

- Does the animation portray a concrete program in a concrete language, or is it based on a kind of pseudo code?

- Is it a visualization for a program written in an imperative, functional or object oriented language?

- Is the visualization domain specific? I.e. the standard visualization of data structures may not make much sense, if the program manipulates geometric data and thus should be displayed in a geometric way. If there is a set of points, then a visualization should display them graphically in their space and not as number tuples.

- Is the goal to show concurrency and the problems that go with it? Again the visualization has to change to accommodate the intent of what should be learned by looking at it.

The list could go on, but it is pretty obvious, that there exists a desire for visualizations to help teach, learn, understand and debug algorithms. The question then is, how many visualizations are there, do they cover the interesting topics, how available are they and, of course, are they any good?

As stated in Shaffer, Cooper, and Edwards [31] and also in the project description of "Building a Community and Establishing Best Practices in Algorithm Visualization Through the AlgoViz Wiki", which references the former, there are some often encountered problems with using algorithm visualizations in teaching:

"On the side of the teachers the main impediments were difficulty finding worthwhile algorithm visualizations for use on a desired topic, difficulties adapting a given visualization to a teacher's classroom settings and a lack of knowledge of the best way to deploy algorithm visualizations."

On the developer's side, a study performed in Shaffer et al. [31] of a collection of over 350 algorithm visualizations "shows that many existing algorithm visualizations are of little pedagogical value, and there is poor distribution of topical coverage." To understand what is meant by the poor distribution of topical coverage, I will give some numbers from the above paper. Of the 350 algorithm visualizations, 134 were sorting algorithms, 60 were search structure algorithms, meaning mostly all kinds of search trees with binary search trees being the most prominently represented structure, another

48 were about lists, stacks and queues and 38 were about graph algorithms, mostly traversals, shortest paths and spanning trees. The rest becomes harder to categorize. The authors also state that they consider only roughly 25% of those visualizations to be of any value in teaching.

1.3 Our Approach

The AlgoViz portal still exists and has grown to 540 algorithm animations, but the percentages and imbalances on the covered topics have not changed significantly. At the time of this writing 97 of the 540 visualizations were recommended to assist in lectures and even fewer for self study purposes. There is another point not mentioned by Shaffer, that all visualizations in the AlgoViz (to our knowledge) share: They all just follow one concrete program execution and are fixed at one certain level of abstraction, thus it is often difficult to see meta information like invariants. We want to follow a different approach that helps with the main problems outlined earlier:

- Available: It should be reasonably easy to find or create a visualization for any given algorithm and not be necessary to find a different tool for every algorithm, which costs time and energy better spent elsewhere.
- Adaptable: The teachers should be able to easily adapt the visualization to their needs, instead of having to use the notation of the visualization. This means both having access to customization options and having access to source code.
- Familiar: On the side of the learner and sometimes also the teacher it is better to have to get used to just one tool for the whole course, than to have to adapt and refamiliarize with a different tool for every algorithm.
- Understandable: Show more than just the program execution, but include meta data, invariants and other knowledge beyond just the current numbers.

Note that the following subsections will first give an intuition of how shape analysis works and then go into some more technical details that are needed to understand many of the tools and concepts described in later sections. If you are just looking for a quick summary of how our approach does help with the points mentioned above, then you can skip ahead to Section 1.5.6.

Our approach differs from the usual (as seen in the AlgoViz collection) approach in two key points, that are strongly tied to each other:

- As already mentioned, the overwhelming majority of algorithm visualizations follow one concrete execution path of a given program. Whether the input is fixed, random or user generated, the visualization will choose a visual representation of the important data structures of the program and then simulate the steps of the program on that concrete input, updating the data structures for every step.

  Our approach is to look at all possible (valid) inputs for the given program and then simulate the program on all those inputs. Note that it usually does not make much sense to consider in fact all possible inputs, since a program that expects a certain input data structure will usually not work at all if given a different data structure. So unless the explicit purpose of the visualization is to see what would happen with an invalid data structure as input, we will always assume the input to be a valid one.

- The second point is a difference in the fundamental philosophy behind the visualization. Usually visualizations focus on what changes from step to step to better allow a user to understand just what the algorithm does in that special case. Our emphasis lies on showing what actually does not change, or, in other words, what is invariant. To better understand why this might be desirable, let us imagine an AVL-tree insertion. Sure, it is interesting to see a concrete example of an insertion, but usually when in class the focus will be on what stays invariant: The AVL-balance constraint. No matter how the input is chosen, there are only two main cases: Either the balance constraint after insertion is preserved, then the program finishes, or it is broken by the insertion and it will backtrack to do a single rotation. And even for the rotations, there really are only two cases, the single and the double rotation in one direction or the other. So if the users look at all possible inputs at once, they can directly show that only those cases exist and that after a single rotation the balance will always be restored and thus that the resulting tree fulfills the AVL-constraints once more. This meta level invariants cannot be deduced by a single execution, but by looking at the program on a more abstract level and that is exactly what our approach does.

Our method looks at all valid inputs at the same time and then simulates the program on all of them to find the already mentioned invariants and meta information. How is that even possible? Even if the input data structure is just a simple list of arbitrary length, then there exists infinitely many lists that are perfectly valid. Obviously it is not possible to simulate the algorithm for each of them separately, but that is not necessary either. Imagine for the moment a standard classroom situation, and the problem of drawing a list of arbitrary length comes up, then it is quite certain that the result would look like the head of the list, followed by the next pointer, followed by dot, dot, dot followed by the last element of the list or something quite similar.

What happens there is that we argue over a whole set (possibly infinite) of data structures that have similar properties, at least according to what is important for the algorithm at that point. No matter if the dot, dot, dot notation is used or symbols like little triangles to represents trees, what really happens is an abstraction of each graph in a set of graphs, such that new equivalence sets are created. We might call these graphs "*summary graphs*", because they summarize a lot of concrete graphs into something more abstract. To illustrate the point consider Figure 2 as an example of the usual visualization. Here we show one concrete example of list. Conversely look at Figure 3 to see an example of our approach. At first glance they seem to be very similar, both have a similar structure, the same pointers, only the length differs. But in fact, they are very different in that Figure 3 does not represent one single concrete list, but a whole set of lists. Think of the double outlined nodes as a symbol similar to dot, dot, dot or the

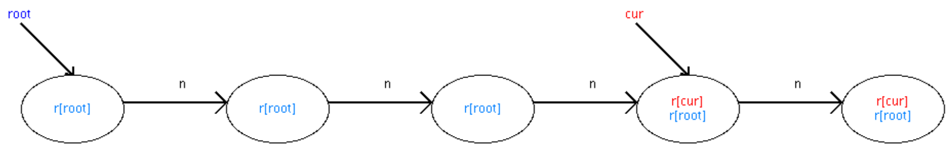


Figure 2: This graph is a typical example of what we expect a list to be visualized as. It shows one concrete data structure: A list with five nodes, the first node has the `root` pointer and the second to last node has the `cur` pointer on it. Every oval stands for one concrete list node and the `n` pointer is the next pointer, that connects the list together. This or something similar is what most usual visualizations will use to represent a list.

triangle that represents a subtree. We will call these kind of nodes *summary nodes.* In this case they stand for at least one, but arbitrary many list nodes. It should be obvious, that the list from Figure 2 is included in the set of lists that are represented by Figure 3, all we have to do is instantiate the leftmost summary node with two list nodes and the right one with a single list node. Our approach works with these summary graphs,

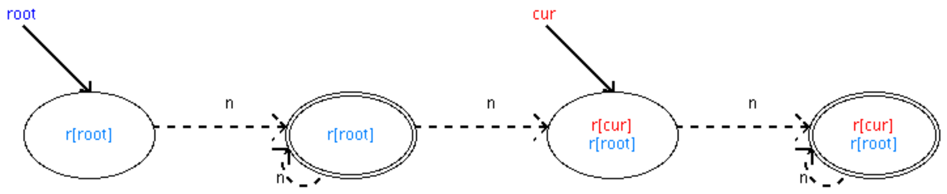


Figure 3: A singly linked list with `root` as head element and `cur` as iteration pointer. Summary nodes are marked by their double outline. The `n` pointers represent the next pointer in the list structure. This graph represents a whole set of lists, including the one from Figure 2.

but they alone are not enough. In order to actually define a sensible abstraction, we need some mechanism to assign and compare properties. Canonically this can be done by using logical predicates to represent the properties and logical definitions and constraints to give them a meaning (usually called a semantics). In the example figures we use the `r[x]` predicates to indicate reachability from pointer `x` over the `n` pointer, each node can of course reach itself.

There is still the problem of guaranteeing that our input (and output) is finite. After all just having abstract graphs does not mean, that there are only finitely many of them. To this end, we have to define our abstraction in such a way, that there can be only finite many graphs. So let us look closer at what exactly our abstraction does: Whenever we use dot, dot, dot notation or something to symbolize a subtree or something similar, then there are some implicit assumptions we make in our minds:

- The abstracted elements are all similar in their properties. E.g. all are list nodes, all have next pointers, all are predecessors of some node and successor of another, all have some data element and so forth.

- What we do not see due to the abstraction is either not relevant at all, or it can be easily reconstructed due to some constraints. E.g. if our sequence of nodes is supposed to be part of an acyclic list, then we know, that there will not be a cycle, or if it is a subtree that is part of an AVL-tree then we assume it will satisfy the AVL-balance property.

This means that our abstraction should work along the same principles: Nodes with the same properties get abstracted into a single entity, something we call a "*summary node*" for the same reason as the summary graphs. We already mentioned, that logic predicates are a canonical choice for modeling properties, so we could abstract nodes whose logical predicates evaluate to the same logical values. Since we already mentioned, that logical definitions and constraints can be used to give those predicates a semantics it is also quite easy to apply them if it is ever needed to get a node back out of a summary node. An example of this can be seen in Figure 3: We only have four predicates here to consider, that is `root`, `cur`, `r[root]` and `r[cur]`. Since the pointer predicates are unique (a pointer can only ever point to one node at a time), these nodes will never be abstracted into a summary node. Since the `root` pointer marks the head of the list, the other nodes in the list can only be either between the `root` and `cur` pointer or they are behind the `cur` pointer. This is in turn reflected by their predicates: Some are only reachable by `root` (`r[root]`) and others are reachable by both, thus two equivalence classes are created that correspond to the two summary nodes.

How does that solve the problem with the finite number of graphs? Since we only ever care about a very limited (and certainly finite) number of properties when we look at any algorithm, we also will only have a finite number of predicates and rules describing them. That means that each node has a vector of truth values that correspond to each predicate, since the predicates are finite and the truth values are finite too, there can only ever be a finite number of different nodes. Remember, that nodes with the same truth vectors will be abstracted together. That in turn means, that our summary graphs have an upper limit of how many nodes there can be, which puts an upper bound on the number of different summary graphs. Thus the size of our input and output sets is bounded and certainly finite.

What remains to be shown is that our method is well-founded. For that we have to look again at how our visualization is supposed to work: We give our program a set of input summary graphs, that define all valid inputs for this program. Then those graphs are modified, according to the program statements. Since we already use logical predicates, we can relatively easily use the operational semantics of the programming language to compute how the graphs (i.e., the data structures they represent) are changed. We already showed, that there can be only finitely many graphs, so all we have to show now is that we do not look at the same graph over and over again.

Our method avoids that by checking each newly generated graph for graph isomorphism with already generated graphs. That means we start with the set of input summary graphs and generate new graphs according to the operational semantics of the first program statement. Then we check for graph isomorphisms in the new set and remove duplicates. If we iterate through a program loop, then there might already be summary graphs in there from previous iterations, so we check those for graph isomorphisms too. If there is no isomorphic graph present, then the new summary graph will be moved into the to-do list. Our method will abort, if the to-do list runs empty. Since there are only a finite number of summary graphs possible, at some point all graphs will be generated and thus there will be no new graphs for the to-do list, thus this method is also well founded.

What we described above is the basis for a static program analysis method called *shape analysis*, which is usually implemented to verify certain program properties. A more formal introduction to shape analysis will be presented in the next section.

1.4 Shape Analysis

Shape analysis is a static program analysis, meaning that it is run at compile time rather than run-time. As a type of verification method it can prove or disprove certain well-specified properties of a given program. It can be thought of as a kind of storeless pointer analysis, in the sense that it is not interested in concrete memory locations, stack frames or numerical values, but only looks at the structures in the program heap (as in the dynamic memory used by a program). That means certain memory areas are logically thought of as a single object (even if they are not in the actual implementation) and the analysis looks at and models the relationship between these objects, especially the pointers between these objects and those pointing from the stack into the heap, thus the name, shape analysis, because it describes the shape of the heap. The goal of shape analysis is to synthesize invariants from possible heap descriptions in order to prove or disprove properties. It should be noted here that this is also a limitation on what is actually practical to visualize with this method. Anything that requires more than very limited numerical analysis is not really recommended. The reason for this is that each numerical value, or, at the very least, value range, has to be modeled by a predicate. Thus even something as simple as basic addition of natural numbers in the range of one to one million would be nearly impossible to do, since it would need a million predicates, as well as the faculty of a million rules how to do the addition. Shape analysis has been applied to a variety of problems (list taken from the wikipedia page):

- Finding memory leaks, including Java-style leaks where a pointer to an unused object is not nulled out
- Discovering cases where a block of memory is freed more than once (in C)
- Finding dereferences of dangling pointers (pointers to freed memory in C)
- Finding array out-of-bounds errors
- Checking type-state properties (for example, ensuring that a file is open() before it is read())
- Ensuring that a method to reverse a linked list does not introduce cycles into the list

- Verifying that a sort method returns a result that is in sorted order

To solve those problems, shape analysis finds invariants about each program point, i.e. it describes all possible heap shapes at each program point. Those heap shape descriptions are also called *shape graphs* and are identical with what we called summary graphs in the previous section.

Before explaining how a visualization that is based on these kind of graphs can help with some of the algorithm visualization problems, I need to make clear what I mean with the term *concrete* as used above and respectively the term *abstract*. In any kind of algorithm visualization there is at least some kind of abstraction, since one hardly wants to picture the whole electronic interior workings of a computer for every algorithm. Thus there always exists some level of abstraction depending on the goal of the visualization. For my purposes I will make use of three abstraction levels:

The lowest level of abstraction is the one seen in the examples of common algorithm visualization, where each heap object is unique and all interesting information is available. Usually this can be thought of as looking at one specific execution of a program, where every single bit of data can be accessed and visualized if necessary. It could also be seen as a kind of data view, where concrete data values and even address space are still available informations.

The next level of abstraction is defined by concrete shape graphs, which means shape graphs that do not use abstraction to summarize nodes, i.e. there are no summary nodes and all relations are definite, that means there is no loss of information about the structure of the heap. This still can be seen as looking at one specific execution of a program, however now there is some information not available, like concrete data values or minutiae of the address space. Figure 2 from the previous section is an example of this.

The final level of abstraction consists of the abstract shape graphs, like the one in Figure 3 from the previous section. Abstract shape graphs are the norm for shape analysis and as a direct implication of the abstraction follows that there is also a loss of information, more often called a loss of precision.

So if I speak of a concrete example, then I mean a specific execution of a specific program on the lowest abstraction level. A concrete shape graph is a description of the heap state of such a concrete execution at some program point, with the added abstraction of shape analysis, that does not care about concrete numbers (or values), variable sizes, memory addresses and similar things. An abstract shape graph finally is part of the output of a shape analysis describing a usually infinite number of possible heap states at a specific program point. All abstract shape graphs of a specific program point define the invariant of all possible heap states at that program point.

The first step for performing shape analysis is to define a function $f : H \to (O, R)$, that is given a heap and will yield a set of objects and a set of relations between these objects. If we stay with our list example, then the list nodes are the heap objects and the `n` pointer as well as the reachability `r[x]` are relations. The function f can be arbitrary, but must remain fixed for a given analysis. There is also a special kind of unary predicates: The (stack) pointer predicates. Pointer predicates model the program pointers that point into the heap and are thus the link between stack and heap. It may be canonical to define heap objects as instances of classes in object oriented languages, but it does not have to be like that. Depending on the goal of the analysis it may make sense to map different classes to the same objects, to not model certain classes at all or to disregard class structure entirely. Choosing this function f is also a means of controlling the level of abstraction in the analysis.

Now I used the term shape graph without clearly defining it. Generally a shape graph is an abstract description of possible objects in the heap and their relations to each other, especially the pointer connections. Usually this means, that once we have the heap objects and relations defined by the function f, the objects are mapped to nodes in the graph and the relations are mapped to edges or annotations, depending on their arity. After that, the chosen abstraction is applied to detect similar nodes, which are then abstracted into a single summary node. At this point I cannot be more specific without choosing a specific tool that computes the shape analysis, as the expressiveness and representation of shape graphs varies quite a bit.

It was mentioned before, that there was an inevitable loss of information associated with the abstraction that we perform. Formally this follows simply from the fact, that the halting problem is not decidable. If we had an analysis that could model every property of a given program with absolute accuracy, then we could apply this knowledge to decide whether or not said program would accept an input or not. More intuitively, we can consider again the process of abstraction: We look at properties and summarize those nodes that have the same truth values on those properties. So far, so good, but if we look again at Figure 3, then we see that there is a little problem when we abstract nodes into a summary node. Let us consider the `n` pointer: All the `n` pointers in the graph are drawn as dotted lines and with good reason. The `n` pointer is modeled by a binary predicate (i.e. it takes two arguments), if `n(u,v)` evaluates to true, then v is the successor of u in the list. But what happens if either u or v is a summary node? The summary node abstracts an arbitrary number of nodes, but only one of them can be successor or predecessor of another node, not all at the same time. That means setting the predicate to one would be wrong, but setting it to zero would also be wrong, since that would mean the list was not connected anymore. Clearly what is needed is some mechanism to express uncertainty or partial truth and that is what I will present in the next section.

1.4.1 3-valued Logic

Three valued logic may be unfamiliar to the reader, yet it is a central feature and also a central problem in this thesis. Thus I will give some intuition as to how the third logical value works and why it is a good choice to use this kind of logic in shape analysis. Note that this won't be a formal logical definition, for that see Kleene [17], Sagiv, Reps, and Wilhelm [30].

So how shall we imagine the logical value 1/2 aka `unknown`? Intuitively the easiest way is to think of it as "maybe `true` or maybe `false` and no way to know for sure". This is also why the `unknown` value is called an *indefinite* value, whereas `true` and `false` are *definite* values. More formally, `indefinite` can be considered as the union of the two definite truth values.

All the usual two-valued tables for the logical operators are of course extended for the new logical value. See Figure 4 for examples of how the $\vee$ and $\wedge$ operators behave in 3-valued logic. Note that for definite values the operators are unchanged, this will be very important for abstract shape graphs later on. In order to compute this tables, whenever there is the value 1/2 involved, split it in two cases: One time instantiate 1/2 with 0 and the other with 1, take the union over both cases and if it is the same truth value in both cases, then the result will be that definite truth value, if there are different truth values, i.e. both 0 and 1, then the result will be 1/2. Thus computing the truth values in the 3-valued case can be reduced to computing truth values in the 2-valued way.

$\wedge$	0	1	1/2
0	0	0	0
1	0	1	1/2
1/2	0	1/2	1/2

$\vee$	0	1	1/2
0	0	1	1/2
1	1	1	1
1/2	1/2	1	1/2

Figure 4: 3-valued truth tables for the $\vee$ and $\wedge$ operators.

As a special case I want to point out the equality operator and the formula φ : `1/2 = 1/2`. As a first reflex one is tempted to say, that φ evaluates to true, but it holds: $[\![\varphi]\!] = 1/2$. To understand why this is, think back to where I gave the intuition for what 1/2 means. In essence φ stands for the union of four cases:

$$1/2 = 1/2 \begin{cases} 0 = 0 \\ 0 = 1 \\ 1 = 0 \\ 1 = 1 \end{cases}$$

Since some of them evaluate to `true` and some evaluate to `false`, it ends up with the indefinite value. This means that there is no way to specify something like: “If this predicate is indefinite, then set the value of another predicate to something”. As already mentioned above, this also means there can be no testing (by logical formula) whether a node is a summary node or not.

Three-valued logic is a great way to model the information loss caused by abstraction. Whenever two nodes are abstracted into one single node, the algorithm looks at the predicates of both nodes, if a predicate has the same truth value for both nodes, then this truth value will be taken to the summary node, otherwise the value will be set to 1/2. This works both for unary and binary predicates. The direct implication of this is, that using 3-valued logic will result in only erring on the “safe” side. That means if a logical value is definite in an abstract shape graph, then it indeed holds for all its possible concrete instantiations and that in turn implies that if a logical formula holds for a shape graph, then it holds for all its possible instantiations. In this way shape graphs describe invariants for the heap state they model.

This fact is proven in the so called "embedding theorem" (see Sagiv et al. [30]). The name comes from embedding 2-valued shape graphs into a 3-valued one in such a way, that all formulas will evaluate in a "safe" way, i.e. if a formula evaluates to a definite truth value on the 3-valued graph, then the respective concrete shape graphs will all have the same definite truth value. Only if a formula evaluates to `unknown` in the 3-valued shape graph, then there is no guarantee to what the formula will evaluate in all possible concrete shape graphs. It might even be the case that it evaluates to the same definite value. Thus the indefinite value represents the loss of information.

The importance behind all this is of course, that it is possible to prove correctness of programs with 3-valued logic simply by looking at the shape graphs. If all abstract shape graphs at a given program point have definite values for certain relations, then it means, that the definite truth values are actually invariants, holding for all possible heap configurations at that program point. Looking at all program points, it is thus possible to prove properties that hold for every program execution at every program point. If the analysis models some C-program then we can prove for example, that there can never be a null-pointer dereferentiation, simply by looking at all abstract shape graphs and evaluating a formula that represents the problem. If the formula can be evaluated to `true` in all shape graphs then the proof is complete.

1.5 TVLA - A Tool For Realizing Shape Analyses

There are quite a few algorithms for performing shape analyses, a selection of them can be found in the following papers: Assmann and Weinhardt [1], Banerjee, Gelernter, Nicolau, and Padua [3], Chase, Wegman, and Zadeck [9], Horwitz, Pfeiffer, and Reps [12], Jones and Muchnick [14], Larus and Hilfinger [18], Sagiv, Reps, and Wilhelm [29], Stransky [32]. According to Sagiv et al. [30], all of these use shape graphs as "shape descriptors", meaning the description of the heap configuration. Also they all compute shape analyses for specific properties of a certain program or a class of similar algorithms, much like most traditional visualizers can only visualize a very specific algorithm or class thereof.

This is where parametric shape analysis comes in. Instead of being confined to a single program or even a single class of algorithms, the parameters allow for generating shape analyzes for arbitrary programs and properties. TVLA, short for "Three Valued Logical Analyzer" is our tool of choice for this kind of shape analysis. According to Sagiv et al. [30], all of the above tools can be either seen as instantiations of TVLA or in some cases TVLA actually yields higher-precision analyses for the same properties. TVLA being parametric not only means, that it can emulate of lot of other tools, but that we can freely choose both the heap abstraction and the similarity criterion for the abstract shape graphs. This in turn means that it is possible to freely scale the level of abstraction and thus the precision of the analysis! Another feature, that sets TVLA further apart from other shape analysis tools, is the use of 3-valued logic, as made obvious by the name of the tool. I already explained in the previous chapter why it is a very canonical way of modeling information loss and uncertainty. It is, however, also one of the main reasons why TVLA has made huge headway in the power and precision of materialization, as will be explained in a later section. The actual implementation of TVLA is described in Lev-Ami [19], Lev-Ami and Sagiv [20], Lev-Ami, Reps, Sagiv, and Wilhelm [21].

It should also be noted here that today TVLA is no longer the only parametric shape analysis tool. There is also the work of Chang, Rival, and Necula [8] where they introduce a parametric shape analysis based on separation logic. The specification is done by what they call "checkers", which are small functions that in theory would be used by the programmer to check their data structure and which their tool then uses to derive correctness information from. In this original version it was only possible to check structural invariants, i.e. checking properties that are (mostly) independent of each other (this is also why separation logic works so well here). In Chang and Rival [7] they extended this to allow a combination with numerical analyses allowing them to prove properties like sortedness. In total it seems that the tool is still not as powerful as TVLA in terms of expressiveness and the general look of their shape graphs is a bit more formula heavy which makes it in my opinion not a good choice for visualization for teaching purposes.

I will summarize the most important features and mechanisms of TVLA here, insofar they are needed for understanding this thesis and only in the precision needed for said understanding. For deeper knowledge and formal proofs and definitions see the above cited papers. The notations concerning logical structures that are used here come from Sagiv et al. [30], which is also a good place to start for a more logic-based view on TVLA.

1.5.1 Shape Graphs

One of the key features of TVLA is the use of three valued logic as interpreted by Kleene(Kleene [17]). For TVLA heap shapes are described as logical Structures. The following is taken directly from Sagiv et al. [30]:"Each logical structure is associated with a vocabulary of predicates with given arities ranging from zero to two. Each logical structure S, denoted by $\langle U^S, \iota^S \rangle$ has a Universe of individuals U^S, these will represent the heap objects, and a function ι^S, that in the two valued case maps every predicate symbol p and every k-tuple of individuals $(u_1, \ldots, u_k)$ with $u_i \in U^S$ to 0 or 1, i.e. `false` or `true` respectively. In the three valued case they are mapped to 0, 1 or 1/2, i.e. `false`, `true` or `unknown`, the last one is sometimes also called `maybe`."

To get from the logical structure to an actual graph is pretty straightforward: Every individual in U^S is mapped to a node. Every binary predicate can be naturally interpreted as an edge between two nodes, that is annotated with the predicate name. Every unary predicate describes a property of some node and is mapped to a node label and finally every nullary predicate describes a property of the whole shape graph and is mapped to a label outside of any node and edge. Note also, that due to their importance pointers from the stack into the heap are also always modeled as special unary predicates, they have no start node, only the target node.

Formally, let L be a set of labels, $O = \{\phi\} \cup V \cup E$ and $l : O \rightarrow L$ the relation, that assigns labels to graph objects. A graph with labels $G(V, E, L, l)$ is called a shape graph for a logical structure $S = \langle U^S, \iota^S \rangle$ with vocabulary W if and only if there exists a bijective function $m : U^S \to V$ with the following conditions:

- $L = W \times \{0, 1, 1/2\}$ consists of all possible predicates in the vocabulary of S, combined with all possible truth values (0,1,1/2). Note that we canonically identify predicates with predicate names.
- $l : O \times L$ is a relation for which holds $(o, (p, t)) \in l$ if and only if $[\![p(u')]\!]^S_U = t$,

  $$u' = \begin{cases} (m^{-1}(o)) & \text{if } o \in V, \\ (m^{-1}(o_1), m^{-1}(o_2)) & \text{if } o = (o_1, o_2) \in E, \\ () & \text{if } o = \phi \end{cases}$$

  Note that these are the tuples needed for the ι function.

So now that the exact definition of a shape graph is presented, it remains to define how to draw them:

- Nodes are usually drawn as oval shapes, edges as arrows between those as usual.
- Labels are written on the respective edge or inside the respective node or in one corner of the graph for nullary predicates.
- In order to increase the readability of the graph predicates that evaluate to `false` are omitted, i.e. what can't be seen is always `false` as a convention. If a predicate is true then it is common to just write the predicate name without the logical value behind it.
- Another matter to increase readability is to draw only one edge for each truth value between any pair of nodes and concatenate the predicate labels with commas.
- Summary nodes are drawn as normal nodes, but with a double outline.
- Binary predicates, that have value `unknown` (1/2), are drawn as dotted lines.
- Pointers from the stack into the heap are drawn as arrows towards the node they point to, without an origin object.

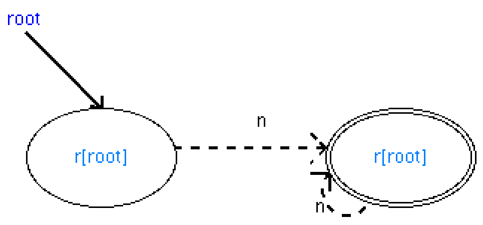


Figure 5: The standard general case of a single linked acyclic list with at least two elements.

The shape graph in Figure 5 depicts the general case of a single linked acyclic list, that is at least length two and can be arbitrarily long. I will use this example to show how this shape graph is represented as logical structure by TVLA:

```
%n = {u0, u1}
%p = {
        sm = {u1:1/2}
        n = {u0->u1:1/2, u1->u1:1/2}
        root = {u0}
        r[root] = {u1, u2}
}
```

The %n list the nodes and in %p all predicates that have at least one non-`false` value are listed. The format (called tvs, aka three valued structure) is straightforward, there are however two things to note:

- As with drawing shape graphs, predicate values of `false` are omitted, i.e. everything not listed is defaulted to `false` and if there is no truth value explicitly given after a node or edge, then it defaults to `true`, which again makes things slightly more readable.

- The `sm` predicate is a special predicate, that denotes whether or not a node is a summary node. The interesting thing to note here, is that the value for every node is either `false` or `unknown`. That means it is only possible to be sure that a node is *not* a summary node, but there is no way to tell that a node is definitely a summary node. This might seem strange, it has however a valid and important reason, to understand one has to look on the subject from two sides. On the one hand, if there is a given concrete graph and we apply some abstraction on it, then we of course know how many nodes were abstracted in each summary node. On the other hand, shape graphs describe a usually infinite number of "similar" concrete graphs. Figure 5 abstracts all single linked list with at least two nodes. Now imagine we iterate over the list, for the iteration to ever stop, there needs to be a case where a materialization of the summary node yields, that this was it, that was the last node in there. In other words, a summary node always stands for *at least* one node, but it *may* stand for arbitrarily many nodes, thus we have both the case of "is a concrete node" and "is definitely a summary node" which in 3-valued logic yields `unknown`. More on this further down. Another important implication of this is, that there is no special treatment possible for summary nodes, one cannot make a logical formula, that test whether a node is a summary node or not. This in turn has serious implications for constructing specifications for TVLA.

1.5.2 Abstraction and Materialization

We already mentioned, that in order for the number of shape graphs to be bounded, there needs to be some kind of abstraction of nodes, this works by abstracting similar nodes into a single summary node. For this purpose TVLA allows us to specify some unary predicates as so called *abstraction predicates*.

It is of course possible to enumerate all those abstraction predicates and then define a vector of truth values for each node, such that the i-th position in the vector represents the truth value of the i-th abstraction predicate. This vector is called the *canonical name* of the node. The abstraction is straightforward, if two nodes have the same canonical name then and only then they will be abstracted into the same summary node. Note that this directly implies that abstraction predicates always have to be definite.

That is the way how TVLA abstracts nodes into summary nodes, but how does it get them out again[5]? Suppose we want to iterate through our standard list from Figure 5. Let `cur` be our cursor pointer that we initialize with the `root` pointer as shown in Figure 6.

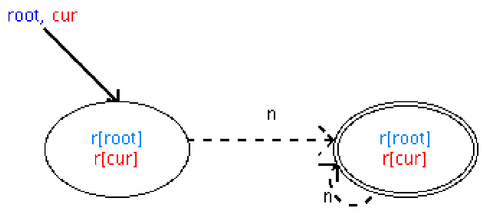


Figure 6: The standard List with the `cur` pointer initialized with the `root` pointer.

The next step is to set the `cur` pointer to `cur = cur->next`. To be able to do this, TVLA has to materialize a node out of the summary node. This sounds trivial at first, but it is actually one of the hardest parts of shape analysis. It was already shown how summary nodes lose information, like number of nodes or truth values of non-abstraction predicates. Even the pointer structure between abstracted nodes is mostly lost, so getting a node back out will usually yield very poor results due to indefinite predicates and lost information. Figures 7-9 show some examples of how the materialization may look like and some of the common problems faced even with something as simple as a list data structure. It is quite obvious that the alternatives shown are not satisfying and getting worse from top down.

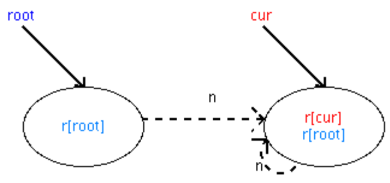


Figure 7: One possible outcome after we set `cur = cur->next`. This is the case where the summary node was instantiated to only one concrete node.

Figure 7 shows the alternative of the summary node being instantiated to just one single node. So far so good, however, there are two imprecise predicates that should

[5]Getting a node back out of a summary node is called *materialization*.

be sharpened (i.e. their information value increased, i.e. becoming definite). The predicates in question are the two `n` pointers. Since both nodes are now definite the `n` pointer between them should be definite too, if the list is linked then there is no choice but for the n predicate between the two nodes to be `true`. The self loop `n` on the other hand should be `false`, since our list is supposed to be acyclic.

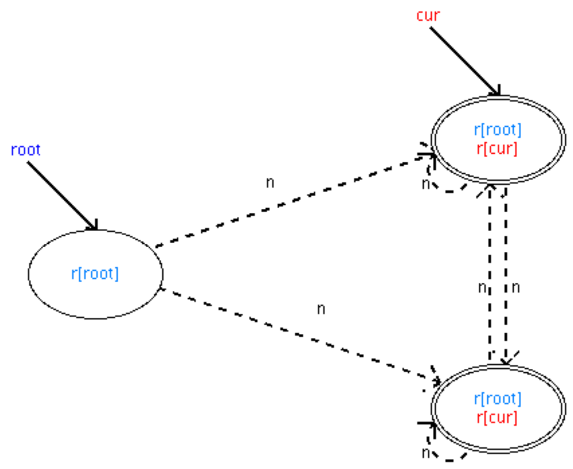


Figure 8: One possible outcome after we set `cur = cur->next`. This does look strange and not entirely right.

Figure 8 has even worse problems. The `cur` pointer seems to point to every node in the summary node at once, which is of course not possible with more than one node, thus the pointed node should really be a concrete node. The other serious problem is, that the list does not even look like a list anymore, the structure is totally lost with all the possible indefinite `n` pointers. It should not be possible for the `root` to reach both summary nodes directly by `n`.

Figure 9 is probably the worst of the three alternatives, it still has all the problems of Figure 8, but it also adds one more thing, that breaks the data structure: Since the `n` pointers between the two summary nodes are missing, it means that either both summary nodes are reachable by the same `n` pointer from the `root`, which is clearly impossible, or one of the two summary nodes is not reachable from `root` at all, which is also clearly impossible, since it wouldn't have been a linked list in the first place then. Thus the whole shape graph should be discarded. TVLA has mechanisms to improve and sometimes even eliminate those problems:

In order to get any precision at all, shape analyses need meta information. I.e. knowing that the analyzed program works with a single linked acyclic list, I already argued, that a shape graph like in Figure 9 can never happen. The mechanism of how to provide this meta information varies, in the case of TVLA it is through constraints. The user is allowed to specify a list of constraints that have to look like implications: $\varphi \Rightarrow p$. Where φ is a logical formula in 3-valued first-order predicate logic and p is either a predicate or

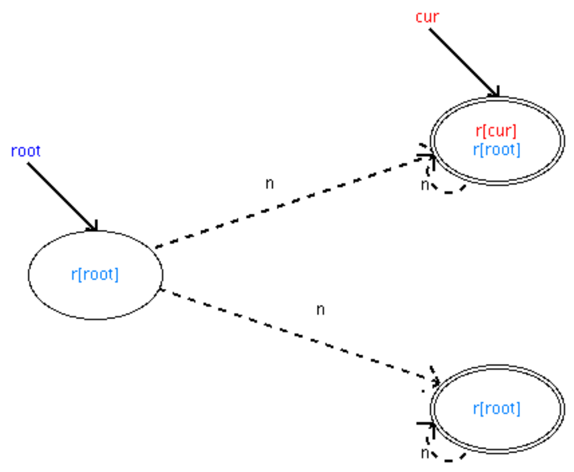


Figure 9: One possible outcome after we set `cur = cur->next`. This one is impossible to get with a list.

the negation of one. The semantic is clear: If the formula φ holds (evaluates to `true`), then p must hold too. All constraints must hold for every shape graph or the shape graph is invalid and subsequently discarded from the analysis.

The first step is to define, by constraints, the acyclicity of the example list structure, thus eliminating the unnecessary `n` self loop. However for that reachability information is needed and we cannot dynamically define reachability with first order predicate logic, three valued or not. Therefore another feature of TVLA adds a limited ability to define transitive, reflexive hulls additionally to first order logic. For more information on the how and the limits see the aforementioned TVLA papers (e.g. Sagiv et al. [30]), for the purposes of this thesis it is enough to know, that it works for all practical purposes. An acyclicity constraint then could look like the following: "transitive closure of n"$(v_1, v_2) \Rightarrow \neg n(v_2, v_1)$

Let us look closer at what a typical TVLA config file looks like. The following is the standard predicate definition file for single linked acyclic lists, taken from the example folder of TVLA.

```
//////////////////
// Core Predicates

// For every program variable z there is a unary predicate that holds
// for list elements pointed by z.
// The unique property is used to convey the fact that the predicate
// can hold for at most one individual.
// The pointer property is a visualization hint for graphical renderers.
foreach (z in PVar) {
  %p z(v_1) unique pointer
}
```

```
// The predicate n represents the n field of the list data type.
%p n(v_1, v_2) function acyclic


////////////////////////////////////////////
// Instrumentation (i.e., derived) predicates

// The is[n] predicate holds for list elements pointed by two different
// list elements.
%i is[n](v) = E(v_1, v_2) (v_1 != v_2 & n(v_1, v) & n(v_2, v))

// The t[n] predicate records transitive reflexive reachability between
// list elements along the n field.
%i t[n](v_1, v_2) = n*(v_1, v_2) transitive reflexive

// Integrity constraints for transitive reachability
%r !t[n](v_1, v_2) ==> !n(v_1, v_2)
%r !t[n](v_1, v_2) ==> v_1 != v_2
%r E(v_1) (t[n](v_1, v_2) & t[n](v_1, v_3) & !t[n](v_2, v_3))
   ==> t[n](v_3, v_2)

// For every program variable z the predicate r[z] holds for individual
// v when v is reachable from variable z along the n field (more
// formally, the corresponding list element is reachable from z).
foreach (z in PVar) {
  %i r[z](v) = E(v_1) (z(v_1) & t[n](v_1, v))
}
```

The first thing to note is, that the file is split in so called *core* and *instrumentation* predicates, denoted by the `%p` and `%i` tags respectively. The difference is, that *core* predicates have to be managed manually, they will not change without being explicitly modified by *actions*[6]. *Instrumentation* predicates are derived from other predicates by evaluating the formula given in their definition, whenever a shape graph is modified or newly created they will be reevaluated.

Another unusual thing are the `foreach` statements. These are just for convenience, in this case allowing to save time and space not having to write the same line for each pointer, they are purely syntactic.

A much more interesting and convenient feature of TVLA are the attributes behind the predicate definitions. Like the `unique, function, acyclic, reflexive` etc., these enable TVLA to create its own constraints for those predicates. E.g. the `function` and `acyclic` attributes behind the definition of the `n` predicate tell TVLA to make sure,

[6]Actions will be explained shortly, they can be thought of as the operational semantics of program statements.

that there can be only one definite `n` edge emanate from any given node (`function`) and that there must never be an `n` cycle.

It should be clear how the formulas are to be read, `|,&,!,E(),A()` stand for the logical $\vee, \wedge, \neg, \exists, \forall$ respectively and the implications are also easy to spot. I will use the TVLA notation whenever I speak about analyses or concrete examples.

There is one more tag not explained yet, namely the `%r` tag. This specifies a constraint formula, as described earlier. Note, that there are usually free variables in these formulas, these can be thought of as all-quantified, they must hold for all possible instantiations within a shape graph.

Lastly we see how the transitive reflexive hull of `n` is defined here, the predicate `t[n]`. This new predicate together with the `function` and `acyclic` attributes of the `n` predicate will allow TVLA to yield much better results than in Figures 7-9. Let us have another look at those shape graphs this time with the `t[n]` predicate present, marked in green color.

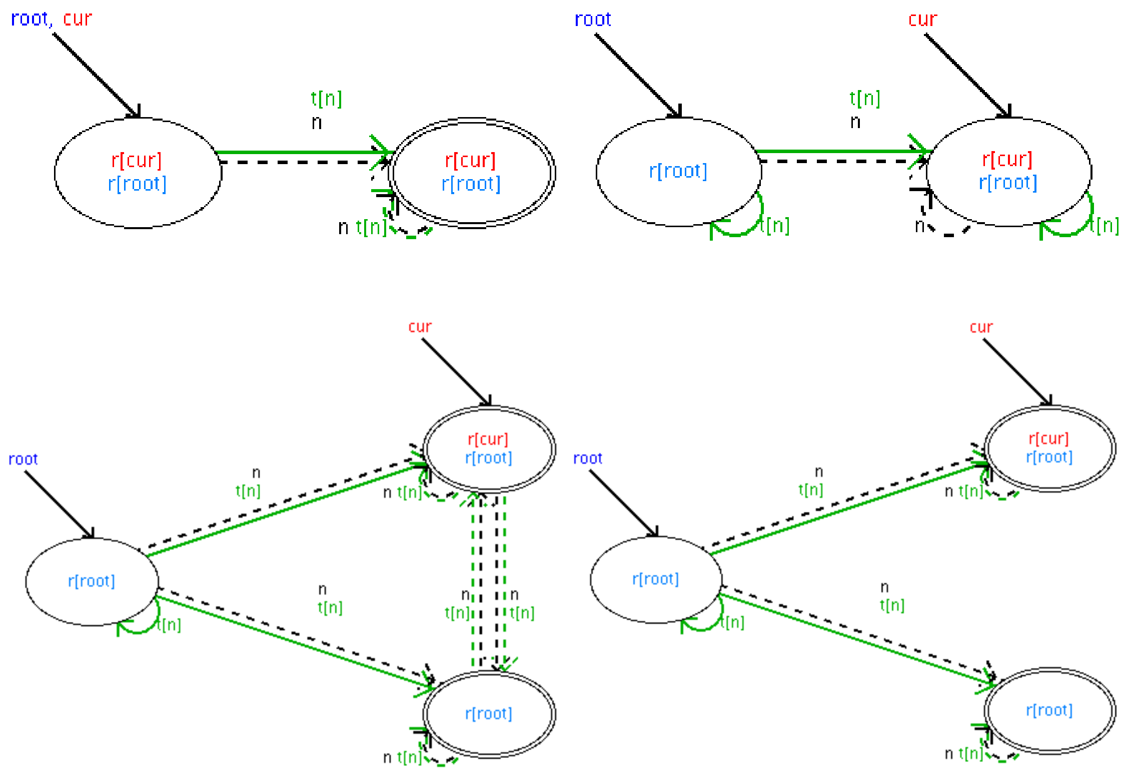


Figure 10: The example shape graphs again. The top left one is the original shape graph, the other three are possible instantiations of what happens after the `cur` pointer is set to the next element in the list.

Thanks to the constraints and the transitive closure we can now refine the graphs: While looking better with the `n` self loop gone Figure 11 still has the problem that the `n` pointer between the two nodes is indefinite. There is no consistency rule in the

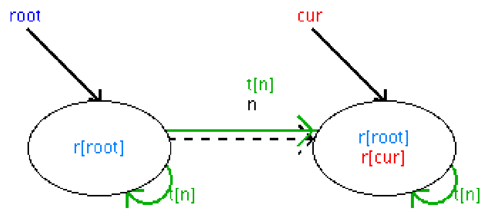


Figure 11: TVLA has successfully removed the `n` self loop. But the other `n` edge is still indefinite.

specification that does sharpen it. The question is, can we make such a rule? The answer is, no, we can not. At least not without breaking other things again. One could try a rule, that if a node is reachable from another, then there must be an `n` pointer between them, but that obviously is wrong. Even if we somehow introduced a predicate for direct neighbors, we still could not do it, because it would also sharpen it for summary nodes and that makes it logically false. Since there is no way to distinguish between summary nodes and non summary nodes in the specification formulas, as I argued before, there is no way to sharpen the `n` predicate just using constraints. Again in Figure 12 there

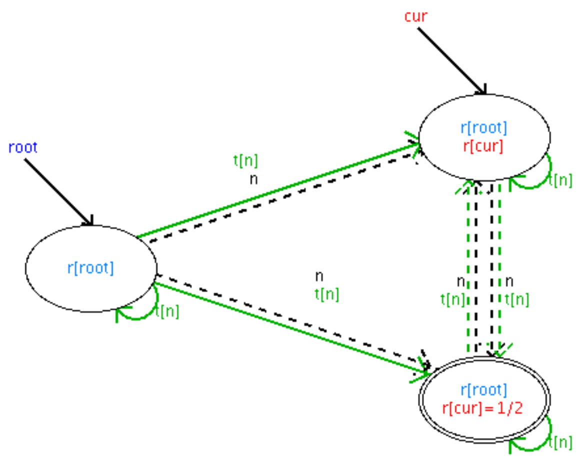


Figure 12: The constraints allowed TVLA to conclude correctly, that since pointers can only point to individuals, the node which `cur` points to is a concrete node and no summary node.

is some improvement. Thanks to the `unique` attribute of pointers, TVLA can conclude that the node `cur` points to must be concrete. Also the self loop of `n` on the same node is gone. Unfortunately the graph still does not look very much like a list. Again with the constraint as they are given in the specification, there is no better information to be had. If it were not for a unique feature of TVLA, this information loss would have to be accepted. To make things even sharper, TVLA introduces focus formulas!

1.5.3 Actions and Coercion

Before I show what focus formula are and can do, there is one more thing, that has not yet been explained: How does TVLA even know what certain program statements do, when it analyzes a program? The answer to this is: It does not know, the knowledge is part of the specification.

The specification of an analysis for TVLA consists of four major parts:

- The main tvp-File containing the annotated control flow graph for the program to be analyzed, as well as the include commands for the other specification files. I will show an example further down.
- The start shape graphs in a separate tvs-File. These are the shape graphs, that specify what the heap looks like when the program starts, from these all other shape graphs are derived. I already showed what tvs-Files looked like when talking about shape graphs. These shape graph must fulfill all the constraints in the predicate definitions.
- The predicate definitions as well as the constraints. It should be noted here that these are highly dependent on the data structure and algorithm used in the program that is to be analyzed. Reusing these definitions for other purposes should be done with a careful and watchful eye.
- The actions file. In this file so called *actions* are defined, that specify an operational semantics for certain program statements. More intuitively: They describe how the heap is changed by any given program line. It is imperative to note here, that just because the code line `cur = cur->next` may be used in practically any data structure and program with pointers called `next`, those code lines will be translated to very different actions, since there is some meta knowledge needed about the data structure and sometimes also the algorithm. This fact makes building an automatic front end for TVLA a very hard task.

Now let us have a look at an action:

```
%action Get_Next_L(lhs, rhs) {
  %t lhs + " = " + rhs + "->" + n
  %f { E(v_1, v_2) rhs(v_1) & n(v_1, v_2) &  t[n](v_2, v) }
  %message (!E(v) rhs(v)) ->
          "Illegal dereference to\n" + n + " component of " + rhs
  {
    lhs(v) = E(v_1) rhs(v_1) & n(v_1, v)
  }
}
```

This is the action that describes what happens when executing the `cur = cur->next` program line, note that the `L` at the end marks that this action is intended for lists (and single linked ones at that). Each action may have up to six tags and a main body that may be empty. Formulas in the main body (denoted by the extra $\{\ldots\}$) always have the same structure: predicate p = formula φ, where φ may be any formula. This will recompute all p values in the shape graph according to the formula φ. Note that the formula is always evaluated in the environment of the old, unmodified shape graph, i.e. modifying p_1 and then modifying p_2 with the use of p_1 in its formula will use the unmodified p_1 values. Thus the order of predicates in the body is irrelevant. Also predicates without an update formula will simply retain their former values, unless they are instrumentation predicates.

The possible tags are:

- %t: Allows to specify a string, that better illustrates the pseudo code meaning of the action. In this example `Get_Next_L(cur,cur)` would print `cur = cur->next`.
- %message Allows to specify a message string, that is printed if the corresponding formula is evaluated to `true`.
- %p While not shown in this example, this tag allows to set a precondition in the form of another formula. The semantic is, that "if the formula is closed, then a result of `true` or `unknown` will let the shape graph pass. If the formula contains free variables, then the action is performed for each assignment into these variables, potentially satisfying the formula." (taken from the TVLA manual) This is often useful to model conditionals, only letting through the shape graphs fitting the respective case.
- %f This allows to specify a focus formula, which helps to increase precision.
- %new This allows for the creation of new nodes. Optionally a formula can be supplied to duplicate existing nodes that match said formula (i.e. formula evaluates to `true` for them), the duplicated nodes will be matched with the originals by a special binary predicate. Also newly created nodes get the unary special predicate `isNew`, which can be used in the following update block.
- %retain In a sense, this is the opposite of %new. By default all nodes persist in the heap, with %retain it is possible to specify a formula, that retains only those nodes, that fulfill the given formula and all else are destroyed. This is most often used to model the freeing of a memory region or a pointer. For example `%retain !cur(v1)` would result in destroying the node where the `cur` pointer is on, effectively simulating a `free(cur)` in a C-like language.

The most interesting and innovative feature is the focus formula: It gives TVLA hints what is important, where to materialize and helps with precision. I already mentioned, that due to indefinite values and information loss, TVLA has to cover all possible cases when materializing a node back out of a summary node. The focus formula has the decisive role to help decide which cases to consider. For all instantiations within a shape graph the focus formula has to evaluate to a definite value. That means that TVLA has to create new shape graphs in such a way that they are consistent with the parent shape graph that is modified and that are consistent with the focus formula.

Let's consider what this means for our running example. The original shape graph can be seen in the top left corner of Figure 10. For the summary node the focus formula can never evaluate to anything else than `false`, since `rhs(v_1)` (which in the example case would be `cur(v_1)`) can only be `true` for one node, due to the `unique` attribute and that one node is the `root` node. This method of anchoring the formula is actually a very common and easy way to focus on a very specific spot in the shape graphs.

So, for the `root` node the formula can be reduced to `E(v_1, v_2) n(v_1, v_2) & t[n](v_2, v)`. What the free variables in the focus formula mean is quite complex and not relevant unless trying to create a specification, therefore I shall just give the rule of thumb, that a free variable prompts TVLA to create a concrete node. Note, that TVLA can come to the conclusion of creating a concrete node even without this, but this way it is forced.

If we remember the constraints, then we can reduce the formula even further. The important constraint is this one:

```
% r !t[n](v_1, v_2) ==> v_1 != v_2
```

It means, that every node must have a `t[n]` self loop, definite or indefinite. This in turn means, that the last part of the focus formula must become `true`, leaving only the `n` predicate! For our example, this means the `n` pointer between the nodes can no longer remain indefinite and there are only two cases left for each of the two remaining shape graphs.

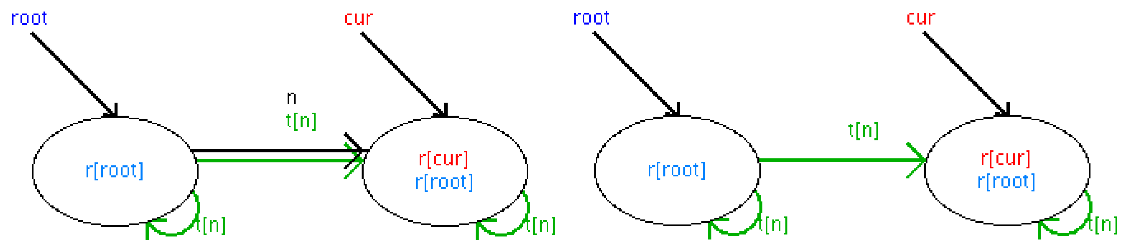


Figure 13: The two possible cases after the focus formula for the shape graph where the summary node gets instantiated as containing only the one concrete node.

Figure 13 shows the two possible shape graphs, for the case that the summary node contains only the one concrete node. It is quite obvious, that the alternative with the missing `n` pointer makes no sense and TVLA will discard it, because it collides with the definition of `t[n]`. The inconsistency is, that `t[n]` says definitely, that the node can be reached by `n` pointer from `root`, but at the same time the `n` predicate says definitely that there is no `n` pointer between the two nodes. This is an unrecoverable contradiction, that will get the shape graph discarded.

Figure 14 shows the analogous graphs for the more general case and of course the right shape graph will get discarded because of the same reasoning as given for Figure 13. It seems rather disheartening that on first glance the remaining valid shape graph still does not look like a list at all. But that is about to change! After the focus formula, TVLA

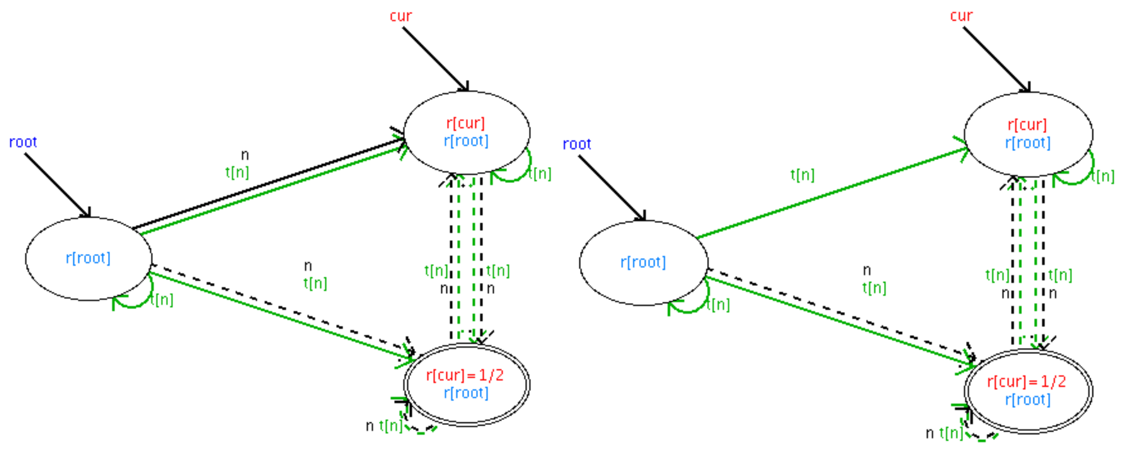


Figure 14: The more general case, that the list contains more than just two elements.

runs over all constraints and instrumentation formulas again, to check for inconsistencies like those that got the unwanted shape graphs discarded. However TVLA does not only search inconsistencies, but it also tried to sharpen predicates if possible.

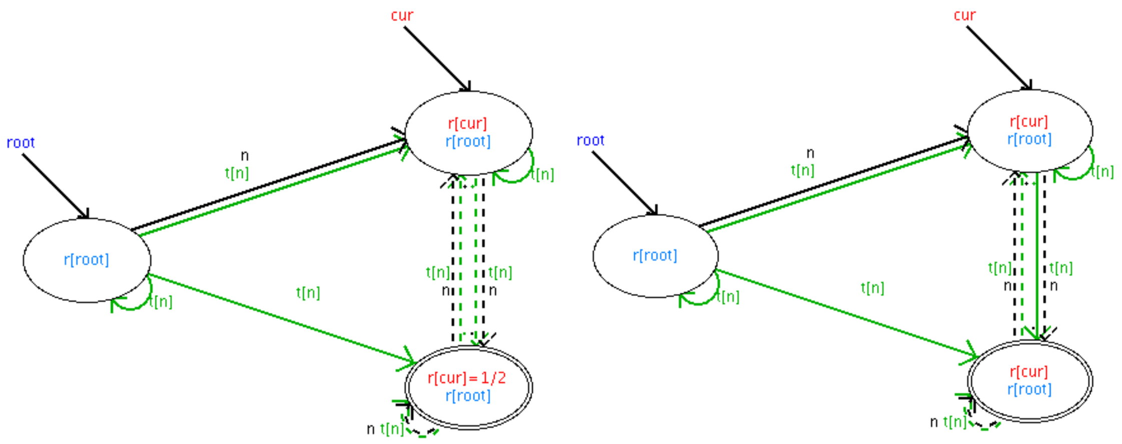


Figure 15: Here it is shown in two steps what happens if TVLA uses the constraints and instrumentation predicate formulas on the valid shape graph after focusing. The right shape graph is after the `t[n]` between `cur` and the summary node was sharpened.

Look at the left shape graph in Figure 15: Due to the simple fact, that the `n` predicate is defined as a function in the specification and that the `n` pointer from `root` to `cur` is definite, it follows immediately that all indefinite `n` pointers originating in `root` can be sharpened to `false`.

This leaves just one more issue, so far it looks like the rest of the list may still reach the `cur` node, which of course cannot be, since we are looking at a single linked acyclic list. Here the instrumentation formula for `t[n]` becomes important: There is now only one route to the summary node, the one over the newly materialized node. Since both the `n` and `t[n]` predicates between `cur` and the summary node are indefinite, the value of `t[n]` from `root` to summary node should be indefinite too (according to the instrumentation formula) but the predicate already has a definite value! TVLA can only ever sharpen indefinite values to definite values while checking constraints and instrumentation predicates, the other way around, weakening information is not allowed and will get the shape graph discarded. In this case however, it can make things right again, by sharpening the `t[n]` between `cur` and the summary node to `true` as shown in Figure 15 on the right side.

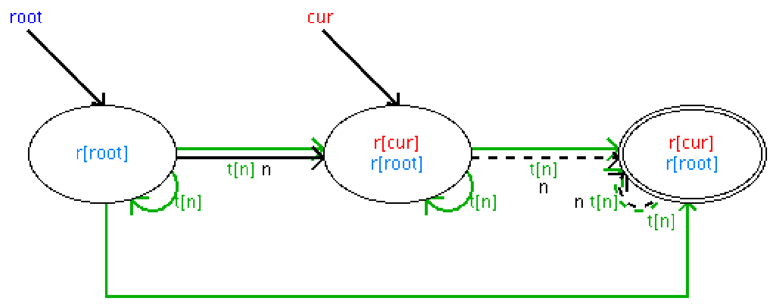


Figure 16: The final sharpened shape graph after all instrumentation formulas and constraints are evaluated. Now it finally looks like a list again.

Now that the line of reachability is definite from `root` to the summary node, it is possible to sharpen things even more: Due to the acyclicity of predicate `n`, it can be concluded, that a node v_1 that can definitely reach a node v_2 due to the `t[n]` predicate, must in turn definitely not be reachable from v_2, else that would imply a path of `n` pointers going from v_1 to v_2 and back again, thus forming a cycle. This means the `t[n]` between the summary node and `cur` can be sharpened to `false`. Now we just have to remember the following constraint:

```
%r !t[n](v_1, v_2) ==> !n(v_1, v_2)
```

It says, if there is no `t[n]` between two nodes, then there can be no `n` either. A direct consequence of `t[n]` being the transitive reflexive closure of `n`. Thus we can now sharpen the `n` between the summary node and `cur` to `false`, leading to the much more intuitive looking shape graph in Figure 16.

This relatively simple example does not only illustrate how TVLA handles materialization, but also shows, that despite the abstraction of summary nodes it is still possible to specify the analysis in such a way, that the resulting shape graphs look very close to what one would intuitively expect when drawing it on a black board for example. The ability to directly influence the look and the precision of a shape analysis simple by adjusting the specification is another big advantage of using TVLA.

1.5.4 The Control Flow

There is only one thing missing before we can piece it all together in a meta overview of how TVLA works and that is the specification of the actual program that is to be analyzed. The following is the main tvp file for a program, that reverses a single linked acyclic list, taken from the examples in the TVLA distribution package:

```
///////
// Sets

%s PVar {x, y, t}

#include "predicates.tvp"

%%

#include "actions.tvp"

%%

//////////////////////////////////////////////////////////////////
// Transition system for a function that reverses a singly-linked
// list in-situ.

L1 Set_Null_L(y)      L2   // y = NULL;
L2 Is_Null_Var(x)     exit // while (x != NULL) {
L2 Is_Not_Null_Var(x) L3
L3 Copy_Var_L(t, y)   L4   //   t = y;
L4 Copy_Var_L(y, x)   L5   //   y = x;
L5 Get_Next_L(x, x)   L6   //   x = x->n;
L6 Set_Next_Null_L(y) L7   //   y->n = NULL;
L7 Set_Next_L(y, t)   L2   //   y->n = t;
                           // }
exit Assert_ListInvariants(y) error
exit Assert_No_Leak(y)        error
```

The includes really just copy the text of the respective file at the place where the include is written. Here we also encounter the %s tag for the first time, which is used to define sets, as mentioned earlier already, these can be used with the `foreach` statement. It is also convenient to see on just on glance what the pointer variables are called in this specification.

The file itself is split in three parts by the %% lines. The first part holds the predicate definitions, set definitions and the constraints. The second part is reserved for the action definitions and the last part specifies the control flow graph.

The control flow graph is specified in a way reminiscent of `goto` style programs. Each lines must consist of three parts, separated by white-spaces: The first string is interpreted as label for the current program point, the second string denotes the action that is to be taken at this point and the third string is again a label, this one denoting the next program point (i.e. a `goto`). There is one thing however that breaks with conventional programming: There may be multiple possible actions taken at each program point. This can be seen in the example tvp file at program point `L2`. This mechanic is used to simulate conditionals that split the program flow, but it is important to keep in mind, that this is not an entirely correct way to think of it. TVLA uses abstract shape graphs, that care not about concrete numbers and values, but still we also want to be able to simulate program statements like "if `(el->data > help->data)` do something". In an abstract shape graph such a condition cannot be absolutely answered, instead the shape graph will be sharpened in such a way that fits the one truth value or the other. A very simple example of this is shown in Figure 17.

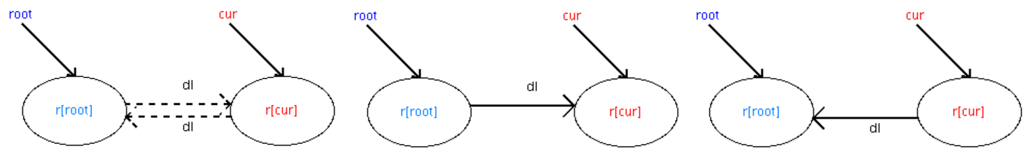


Figure 17: Two arbitrary elements with no information about their data values are to be compared, shown in the leftmost shape graph. This is done by focusing on the `dl` predicate, which yields the two possible results shown here.

The situation depicted in Figure 17 is quite simple: We are at some program point where a conditional is simulated that branches off depending one whether `root` data value is smaller than `cur` data value or vice versa. This is done by two actions:

```
L1 Data_Greater_Than(root,cur) L2
L1 Data_Less_Than(root,cur) L8
```

Both of these actions will be nearly identical, they both will focus on the `dl` predicate, which stands for "data less than". At program point $L1$ there will be shape graphs that look basically like the leftmost graph in Figure 17, all of these will go through both actions. Both times the focus will produce the other two graphs shown in Figure 17. But the first action will have as precondition `dl(cur,root)` and the second one will use the reverse `dl(root,cur)`. Because of these different preconditions, each action will only let one of those two shape graphs through to the next program point.

What that means in general is, that it is not good to think of these multiple labels as `if-then-else`, because that insinuates, that one shape graph can only fit into one such action. This is wrong. *All shape graphs will go through all actions that are defined for a certain program point* and potentially can go to many different successor program points. It is up to the specification to make sure by constraints and preconditions, that only the right shape graphs go to the right destinations. This may be better seen at the `exit` program point. The actual algorithm is done once the program flow reaches `exit`, but there are still two actions performed on those shape graphs: One makes sure, that the reversed list is still a single linked acyclic list and the other that there are no unreachable memory cells floating around. Both together guarantee correctness for the program. They work with simple preconditions, only shape graphs, that are *not* fulfilling the list or no-leak conditions will go through into the `error` program point, that exists purely for reasons of proving correctness. Note that both these tests are independent of each other and are performed on the same shape graphs.

There is not much more to be said about the syntax of the specification itself. Pretty much all label names are allowed, as long as they don't contain white-spaces, the same is true for action names as well. There is no limit on how many actions are performed in a single program point (there are instances where it makes sense to have more than the maximum of two shown in this example) and also no limit on where to jump to after an action. It should be noted here, that when I will later show shape graphs at a certain program point L, it is always from the set of shape graphs at program point L *before* any of the actions at that program point are performed.

Now that the technical details are out of the way there remains one important question: Where does the specification come from? The answer to this at this point in time is unfortunately that they all have to be done manually. While a java front end for TVLA is planned, a practical automated translation of a java program to a TVLA specification is still a long way off.

1.5.5 Piecing It All Together

So now that we know all the different pieces of TVLA, it is time to put them all together into a big picture of how TVLA works. Let's list the steps:

- The first step of creating a shape analysis with TVLA is to think about what kind of data structures there are in the program that is to be analyzed. These data structures then have to be modeled in 3-valued first order predicate logic with the addition of transitive closure. After both the predicates and the constraints are done, it's time to move to the next step.
- Define the actions that will simulate the actual code of the program, using the predicates already specified.
- Translate the program code into a flow graph for TVLA using the recently defined actions.
- Specify all possible inputs of the program by a set of shape graphs.
- Run the analysis.

What happens when the analysis is run? This is a loaded question with a plethora of details that are not all that interesting at this point. There are a lot of options that can be set, including the kind of traversal through the control flow graph, how shape graphs are joined (i.e. when are two shape graphs considered isomorphic), debugging, generation of automatic constraints and many more technical details.

For us however it is of interest how it works in principle: There is some order in which the control flow graph is traversed, whenever TVLA gets to a new program point, it will look at the set of shape graphs there and for each of them perform the actions specified at that program point. The resulting shape graphs are propagated to their destination program points and once there a join is performed. This basically means it is tested whether or not there already is an isomorphic shape graph, if there is then nothing is done, if not, then the new shape graph is added and the program point is flagged as changed. Changed program points will be added to the toDo list. Once there are no more new shape graphs to be found and thus no more changed program points the analysis will stop. The creators of TVLA have proven in the various papers that TVLA is well founded as well as bounded and thus will always terminate. In other words the program will be iterated over till the shape graph sets at each program point are invariant and thus describe all possible heap states for the program.

Let us also consider very broadly the running time of the analysis: For a set A of abstraction predicates, it is possible to have $2^{|A|}$ many different nodes. This is because of the canonical name, that we defined as a vector of definite truth values for the elements of A. This also means we could have that many different shape graphs per program point. Actually the number is a lot bigger since two valued predicates also count for many of the isomorphism tests of shape graphs, which means for every binary predicate we could have every combination of edges between those nodes, which pushes the run time squarely in the $O(2^{2^{|A|}})$.

Performing an action is separated into different phases (the order of which can be configured too):

- Focus: This is what happens when the focus formula is evaluated and where materialization happens. Here is usually where "new" shape graphs are created.

- Coerce: This is what I described as checking all the invariants and instrumentation predicates in Section 1.5.3. The important thing to remember here is, that coerce can only sharpen indefinite predicates into definite values, never the other way around. If there is a conflict that cannot be solved by sharpening, i.e. one constraint says `true` and the other `false` then the shape graph is considered to be inconsistent and will be discarded.

- Precondition: This is simply the check of the precondition formula, if one exists. As explained before for closed formulas a `true` or `unknown` will let the shape graph through, a `false` will get it discarded.

- Update: This is the phase where the body of the action is actually executed, i.e. the predicates listed there will be assigned new values according to the formula given.

- Coerce: Another one, which is to insure, that the update didn't inadvertently break some constraints and also to update some predicate values that can now be sharpened due to what happened in the update.

- Blur: This performs the abstraction algorithm. I.e. it checks the canonical names of all entities and then abstracts them accordingly as described in Section 1.4.1. This is important to keep the analysis bounded and thus make sure that the analysis will come to an end.

1.5.6 Strengths and Weaknesses

Now that we have given a good overview of what TVLA and shape analysis in general can do, let us compare the strength and weaknesses of TVLA with those of normal visualizations (see also Section 1.2) and with our list at the beginning of this chapter:

- Available: It is often difficult to find a worthwhile visualization for a desired topic, let alone a similar or consistent visualizations for multiple topics. Here TVLA has a clear advantage, it is one tool for everything, the same program, the same interface. Saves the time to search for and get used to a "new" visualization for every new topic. Free availability also guarantees less hassle of getting to visualization to students as opposed to web based visualizations that may or may not be there anymore next semester or even next week. Another important point here is that while for some data structures and algorithms there are plenty of visualizations, for others there may be none. (see Section 1.2) Unfortunately there is at the moment also still a problem with the extend of TVLA analyzes, which means that not all standard data structures and algorithms are covered yet.

- Adaptable: It is certainly true: When someone designs a visualization with a very narrow specific purpose and algorithm in mind, the visualization can be great and very detailed and use visuals and gimmicks very specific to that algorithm. This can not be matched with an all purpose visualization that potentially has to handle very different algorithms and data structures. But it also means, that the user is usually stuck with what they get with little to no customization options. This is great if the visualization shows exactly what you want to show in the way you want to show it, but it is unlikely that this happens very often. As a teacher you want to adapt the visualization to your teaching style and not vice versa. TVLA and in extension my visualizer allow a great deal of adaption, as will be shown in Section 2. In extreme cases it is even possible to design your own analysis to fit what you want to show, down to the very name of the relations.

- Usability: There may be many pretty visualizations, but pretty does not mean anything in regards to pedagogical value. In Shaffer et al. [31] they estimate that only 25% of the over 350 visualizations they checked were of a quality that they would feel comfortable using in class. Having just one visualization client makes it that much easier to adapt mechanisms and strategies approved by the visualization community.

- Understandable: Here shape analysis has a clear advantage, it can provide both the meta level view, i.e. the invariants, thus conveying a lot more meta information than any concrete example could ever hope to. Also I will present a compromise in Section 3.3, that allows us to use *concrete* shape graphs to simulate a concrete program execution, combining the abstract and concrete view. This should help those students, that prefer to learn from concrete examples, rather than looking at an abstract definition. We allow both views to coexist, to maximize the learning effect.

- Familiar: As already mentioned in the first point, our Visualizer is just one single tool, that can potentially handle any algorithm, that students are going to see in a lecture. Thus, they only have to familiarize themselves once with the tool and can then use it for the rest of the lecture. Even better, if they have enough understanding, they will be able to customize the view to their liking.

1.6 Motivation and Extent of the Thesis

Originally, the visualization of abstract shape graphs was done solely to make the output of shape analysis tools more readable for humans, thus easier to understand. It was mainly used for debugging of analyses and easier presentation of correctness proofs.

The first prototype of my Visualizer for TVLA was taking a step further and already explored the possibility of using the abstract shape graphs for teaching purposes by adding some new features and a new analysis for AVL-trees. After all, since shape analysis already proved partial correctness for programs as well as providing invariants there was the hope that this additional information could be of use for teaching purposes too.

As already explained at the end of the previous section, it is quite clear that there is great potential in abstract shape graphs due to the fact that they provide a more formal and complete view of a program, as well as having just a single tool to visualize everything, making it much easier to find and adapt visualization to personal style.

Since the abstract shape graphs are really logical structures, it is possible to use the layout of the graph itself, as well as certain other visual features, such as shape or color of nodes, to convey the values and/or meaning of logical predicates without actually drawing the predicate into the graph. A simple example of this is sortedness: When drawing a sorted binary tree then we often implicitly assume, that nodes that are drawn to the left of another node have a smaller data value according to the sortation relation. Or at the very least, there is the convention for sorted trees, that left subtrees contain the smaller data values and right one the bigger. Because this is such a natural assumption, our Visualizer allows the option to simulate it and many others. The order (even a partial order) for each axis can be adjusted by the user. It is generally possible to define a formula for each axis, that induces a sortation such that a node is drawn on a smaller coordinate than another if the formula holds when instantiating it with those two nodes.

One of the most important goals for the Visualizer is, that any and all meaning that is insinuated by layout or graph features must be *safe*. That is, if a feature suggests that a certain predicate or formula holds true for the graph or some nodes then it indeed must hold true according to the logical structure. This is in a way an extension to the *embedding theorem* from TVLA, which states, that a formula evaluated in a 3-valued logical structure must be "compatible" with any results of that formula evaluated in any 2-valued logical structure that is a valid concretization of the 3-valued structure. In other words, TVLA and our Visualizer by extension, are only allowed to err on the safe side, if there is something definite shown, then it must hold for all valid concrete cases. Due to abstraction it is not possible to keep everything definite and this in turn also influences the visualization: Sometimes it just is not possible to compute a definite layout of a shape graph, because of indefinite values in the defining formula. This is a major problem of visualizing abstract shape graphs and will be addressed in great detail in Section 3.1, where I define exactly how the layout of my Visualizer is computed and why it is generally not possible to have a complete ordering of nodes in a shape graph, no matter how the abstraction is chosen. I will also present ways to reduce the problem.

Sometimes an algorithm is best and easiest explained with just an abstract meta view, these are the algorithms that are usually best suited for visualizing by abstract shape graphs. Sometimes, however, it may be easier for the understanding of the students if they are presented with a couple of concrete examples that illustrate the most common cases. These kind of programs or algorithms are a lot less suited for abstract shape graph visualization, because they usually can only provide a very abstract view of the possible cases. To address this issue I have developed the notion of *concrete* shape graphs. These shape graphs can be considered as 2-valued logical structures, or alternatively as 3-valued logical structures with only definite values. The basic idea is that while keeping the same look and logical notation of abstract shape graphs, there just is no blurring performed. That means all nodes will always stay concrete, as will all predicate values and nodes will never be abstracted into summary nodes. This allows for the computation of a "concrete" execution of the program, while at the same time we can match the concrete shape graph to the respective abstract shape graph, thus giving something of a hybrid concrete and abstract view. It should be noted that by deactivating the blur operation, there is no more guarantee that the analysis will stop, i.e. if the algorithm diverges in the concrete example, then the analysis will too. The analysis will of course stay partially correct. Section 3.3 will cover this feature in detail.

Shape analysis will compute invariants and depending on the specification of the analysis these invariants will be more or less detailed. The more detail and thus the more information that is definite, the more shape graphs are usually needed to describe the possible heap shapes. This can be illustrated by a binary tree again. If the only goal is to know information about the node under the `cur` pointer, then the rest of the tree can be abstracted into a single summary node with very little information value and it may be possible to have just one shape graph describing the heap state. But if the goal was to also know whether or not the `cur` node is a left or right child, then we would need to distinguish these two cases, effectively doubling the number of shape graphs. If we continue this example, such that now the goal was to know whether or not the last n ancestor nodes of `cur` where left or right children, then the number of cases rises exponentially.

If it were always possible to specify the analysis as precise then this would not be much of an issue, unfortunately, these specifications are often quite complex and not that easy to direct. It may be necessary to have a certain precision in some case and another in the other case. In short, in practice it is not uncommon that there are unwanted case distinctions in shape graphs, that do not help with understanding, but can not be easily removed because they are needed elsewhere. In such analyzes it may happen that there are hundreds of shape graphs in a single program point. That is clearly not very feasible as an explanation of an algorithm and intimidating for students. Dierk Johannes has worked on solutions to this problem in his Phd-Thesis (see Johannes [13]) and I will present the partitioning methods that were incorporated into my Visualizer in Section 3.2. The general idea is that the core structure of a shape graph is defined by predicate values and/or a logical formula and all shape graphs with the same core structure will then be abstracted into a isomorphic class, not unlike the abstraction process of nodes.

The names and specifications of TVLA actions are often not suited for use as pseudo code. Moreover different teachers may prefer different styles of pseudo code. Thus in Section 3.4 I will show how my Visualizer allows anyone to easily specify their own pseudo code, including even the possibility to subsume more than just one action into a single line of pseudo code.

Section 4 serves as a proof of concept. Here I present the specification of another analysis, this time for pairing heaps, using all the theoretical foundations from Section 3. The intent was to see if using them would make a noticeable difference in the output complexity and judging from the numbers it certainly did. I will try to use only the output of TVLA to describe how pairing heaps work.

While all the theory is certainly important, it should be noted there that a significant part of the work was also of a much more practical nature. The Visualizer code has an extend in the order of twenty thousand lines of code, with several thousand more than were used as experiments and did not survive in the current version. There are many little tools and features that may seem trivial or commonplace, but since shape graphs are not the average, everyday graphs I will use Section 2 to both show the common little problems with shape graphs and the common little tools that can help with them.

2 Common Problems and Tools to Help with them

The shape graphs presented up to now were all very simple, neat and showed only what information was needed for the example case. This in turn summarizes very well the rules for visualization in teaching:

1. Simplicity: For a better overview, keep the example as simple as possible while still showing all the important details. How much detail ought to be shown depends of course a lot on what the teacher actually wants to achieve with the example.
2. Cluttering: Is to be avoided as much as possible. Only the information should be shown, that is actually needed to understand and explain the example algorithm. This is especially important for shape graphs, since there often are construction and helping predicates, which are needed for the proper functioning of the analysis, but are not helpful at all for human viewers and indeed only add confusion.
3. Disambiguity: When information is shown, then it should have a clearly defined meaning. If rules are set then they should always be followed. If it is not possible to follow some rule for some reason then it should be very clearly and unmistakeably stated that the rule is broken at that point. This is also of great importance for shape graphs: Because of abstraction there will always be uncertainty for some predicates, this uncertainty poses a problem when trying to set clear rules. How deep this uncertainty runs will be explored in Section 3.1.

In order to help us to abide by these rules, my Visualizer provides some basic tools, which will be presented in the following sections. In order to provide more meaningful examples for the application of those tools, I will introduce another data structure first: Binary search trees.

```
%s TSel {left, right}

//////////////////
// Core Predicates

...

// For every field there is a corresponding binary predicate.
foreach (sel in TSel) {
  %p sel(v_1, v_2) function invfunction // assume treeness
}

foreach (x in PVar) {
  %p ancestor[x](v) abs
}

/////////////////////////////
```

```
// Instrumentation Predicates

// The down predicate represents the union of selector predicates.
%i down(v1, v2) = |/{ sel(v1, v2) : sel in TSel } invfunction acyclic

// The downStar predicate records reflexive transitive reachability
// between tree nodes along the union of the selector fields.
%i downStar(v1, v2) = down*(v1, v2) transitive reflexive antisymmetric

// For every program variable z the predicate r[z] holds for individual
// v when v is reachable from variable z along the selector fields.
foreach (x in PVar) {
  %i r[x](v) = E(v1) (x(v1) & downStar(v1, v))
}
...
```

The above excerpt from TVLA's examples shows the most important new predicates for binary trees. The first thing to notice is, that there is now a new set of predicates defined called `left` and `right`. These model the child pointers for the left and right child of a node respectively. They are defined as `function` and `invfunction` because every tree node can only have one left child and one right child and every node may also have at most one parent.

The `ancestor[x]` predicates are not part of the original TVLA specification, but are my own addition. They are used to keep nodes that lie on the path from the root to a pointer x separate from non-path nodes. This will lead to separate summary nodes for ancestors and non-ancestors. This distinction becomes especially useful once we go beyond simple binary trees and look at balanced binary trees like AVL-trees.

The `down` predicate is a help predicate to make defining reachability easier, its meaning is to specify a parent to child relation without the added information of whether it is a right or left child. The predicate `downStar` is the reflexive transitive hull of `down` and is the tree version of reachability: If there is a `downStar` from node v_1 to v_2 then it means node v_2 is reachable over the `left` and `right` pointers from v_1 or in other words v_1 is an ancestor of v_2.

The `r[x]` predicates we already know. They are used again to specify reachability of a node from a certain pointer `x`. Even the definition is nearly identical, except for using `downStar` instead of `t[n]`. Note that there are of course a number of constraints needed to fully specify trees, such as making sure that there are no cycles and that different subtrees cannot reach each other. However for the coming examples these rules are of no importance and I will skip them in accordance with teaching rule number two. What is needed, however, is the specification for a simple sortation, taken again from the example folder of TVLA:

```
/*********************************************/
/*************** Core Predicates *************/
```

```
%p dle(v_1, v_2) transitive reflexive

/**********************************************************/
/****************** Instrumentation Predicates *************/
%i cmp[dle,left](v_1, v_2) = dle(v_2, v_1)
%i cmp[dle,right](v_1, v_2) = dle(v_1, v_2)

%i inOrder[dle,left](v)  = A(v_1)  left(v, v_1) -> dle(v_1, v) nonabs
%i inOrder[dle,right](v) = A(v_1) right(v, v_1) -> dle(v, v_1) nonabs
%i inOrder[dle](v) = inOrder[dle,left](v) & inOrder[dle,right](v) nonabs
...
```

The core predicate `dle` is an abbreviation for "data less or equal" and that is exactly what is models: The usual less or equal relation on data values of tree nodes. So far that seems simple enough, so why are there five more predicates defined? To understand that, recall that shape analysis deals with abstract shape graphs: Once nodes are abstracted into summary nodes the binary predicate `dle` will always become indefinite, unless all nodes in the summary node have the same data value. But if we consider the general case, then there must be some way to specify, that a given subtree, although abstracted into a single summary node, is completely ordered according to the `dle` predicate. This is what the other five predicates are for.

The most important predicate is `inOrder[dle]`, if this one is true for a node v, then it means, that the subtree with root v is a completely ordered (according to the `dle` relation) binary tree, i.e., that every left child of a node in the subtree has a smaller data value than its parent and vice versa for every right child. Predicate `inOrder[dle]` has in turn two defining help predicates. One says that for all nodes v_1 must hold: If v_1 is a left child of v then the `dle` predicate must hold `true` between v_1 and v, i.e., it makes sure that a left child has a smaller data value than its parent. The other says a right child must have a bigger data value than its parent.

The `cmp` predicates are a bit more involved and are only needed for the update block in actions. Simply said they allow to easily and elegantly formulate the `pointer = pointer->left` and `pointer = pointer->right` statements in a single TVLA action. Without them there would need to be two separate actions, because the "correct" direction of the `dle` relation depends on whether it's a right or left child. I won't go into further detail as it will only distract from my examples. The important thing to remember is, that these help predicates do make sense and all have their uses.

2.1 Too many/confusing Predicates

Now I said that those help predicates all have their uses, but does that mean they always need to be shown? Let's have a look at Figure 18. Note the numbers beginning with an underscore inside the nodes are the TVLA internal names for these nodes. This shape graph is a good example for one of the most mundane problems with TVLA output:

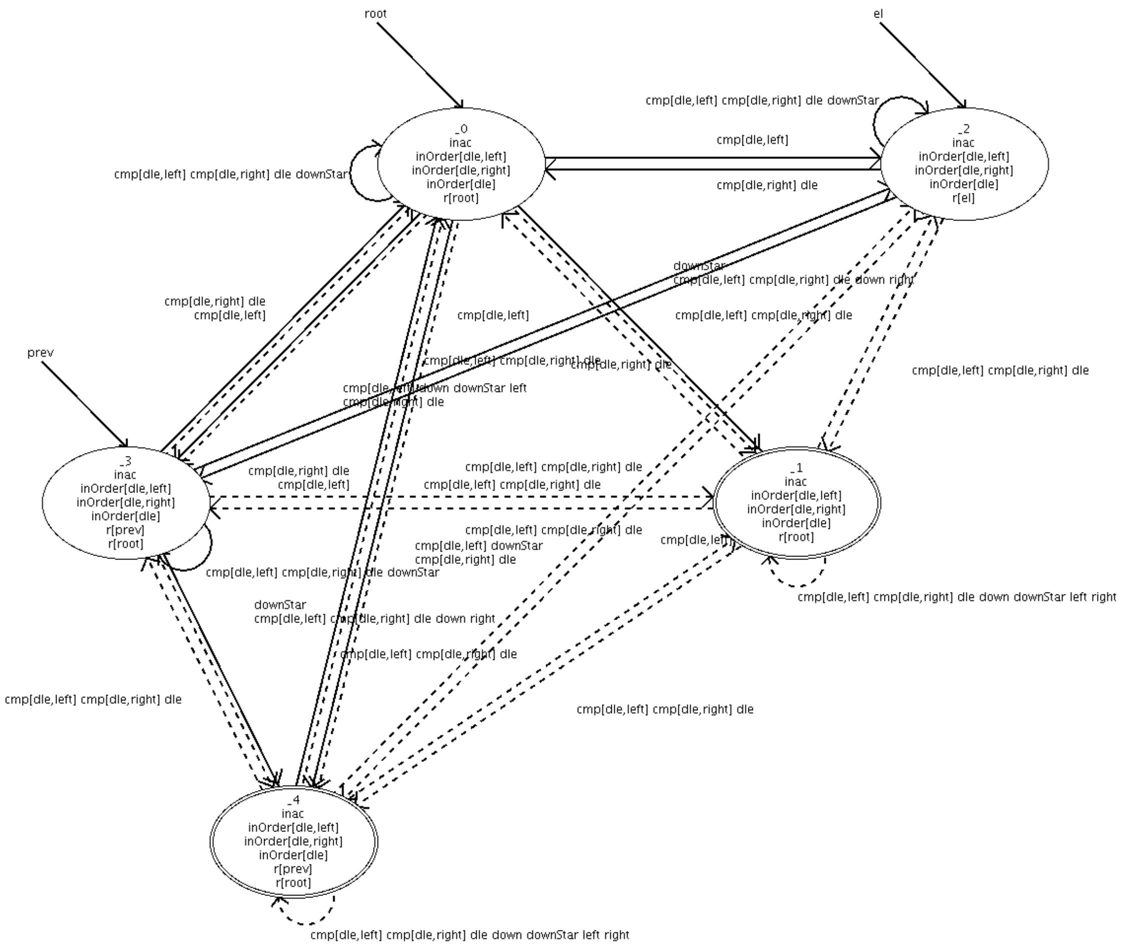


Figure 18: A sorted binary tree in the middle of a standard insertion algorithm, with all predicates shown. This is an example shape graph taken from one of the tree example analyses of TVLA called "insertSorted". It is quite obviously overloaded with edges and labels.

There are simply too many predicates. One could argue that the direct line edges are not the best choice here. That may or may not be true, but trying with orthogonal edges or curved edges will also ultimately result in a big ugly pile of edges that are hard to follow, much less easy to identify on just a glance. The reason for this is of course that the graph is highly non planar, because it is completely connected by at least three separate predicates (`dle` and both `cmp`) and nearly completely connected by `downStar`. It is in fact not even obvious, that this is supposed to be a binary tree plus a single not connected node that is to be inserted.

No one in their right mind would use this graph in an explanation of how insertion into a binary tree works. There are, however, a couple of features in the Visualizer that will make the graph much easier to look at.

The first and maybe most obvious is, that all those help predicates are not really needed for human understanding at all. What is of interest for such a simple insertion is the structure of the tree and whether or not it is ordered. So let's remove all those predicates that are not needed as seen in Figure 19.

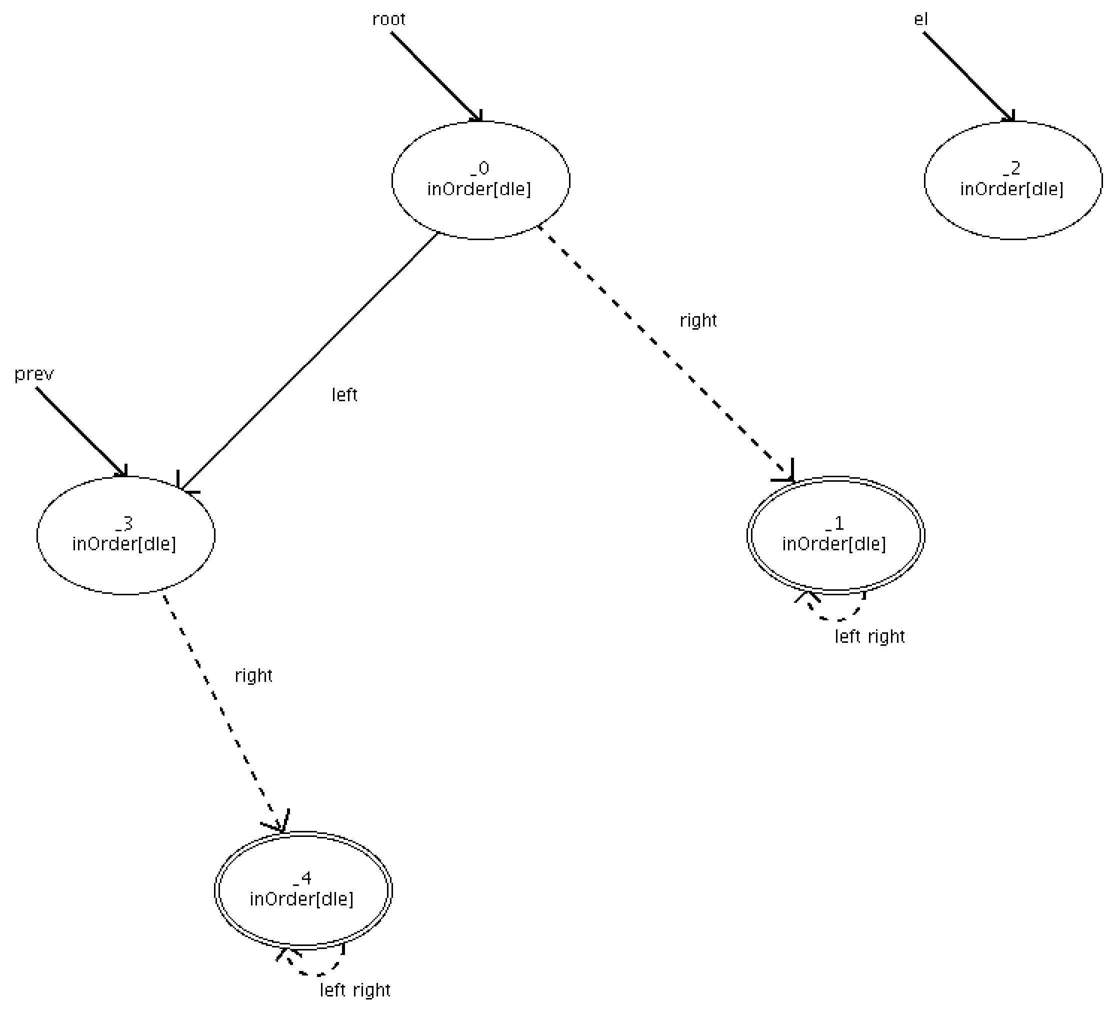


Figure 19: The same shape graph as in Figure 18, but without all the "unneeded" predicates. Without the clutter the structure of the tree becomes clear and the algorithm much easier to follow.

The change is very dramatic, as is the increase in readability of the graph. It is now clear that we see a tree, that there is a cursor pointer called `prev` that sits on the left child of the root node and that there is an isolated node that is to be inserted and pointed to by the `el` pointer. The question is, did I remove too much? Can everything important still be observed? And the answer is, that I may have omitted one predicate that is still needed, namely the `dle` predicate. As a reminder this predicate indicates whether the data values of two nodes are less or equal to each other. We do not need this predicate to tell whether or not the tree is still correctly sorted, because we can conclude that from the `inOrder[dle]` predicate. What we cannot observe without the `dle` predicate however, is what happens at the test if the element to be inserted is bigger or smaller than the current tree node and thus whether the algorithm goes into the left or right subtree in the next iteration. So we add the `dle` predicates back to the graph, but to preserve the easy understanding, color it differently to the structure pointers as seen in Figure 20.

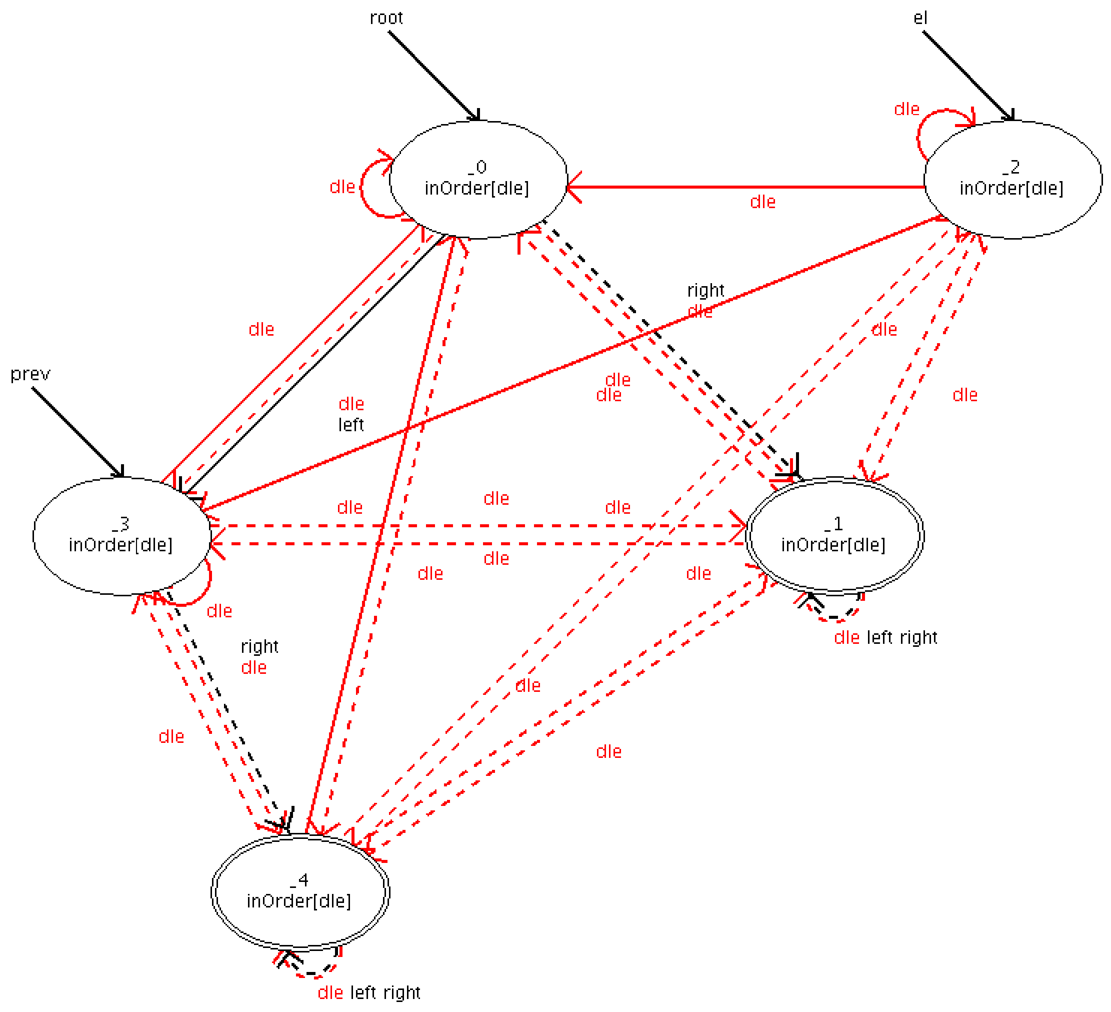


Figure 20: The same shape graph as in Figure 19, but the `dle` predicate was added back. To preserve some of the overview it was colored red, as opposed to the black structure edges.

Thanks to the different colors, it is now possible to easily distinguish between structure edges in black and `dle` edges in red. However the graph feels a bit cluttered again.

While not nearly as bad as Figure 18, there still is a lot of information, that is of no great interest. In fact we might decide, that we only really care about the `dle` relation between the `prev` node and the `el` node at this point in the algorithm. So why not just mark these two nodes as important and ignore all edges that are not between those two, this is shown in Figure 21.

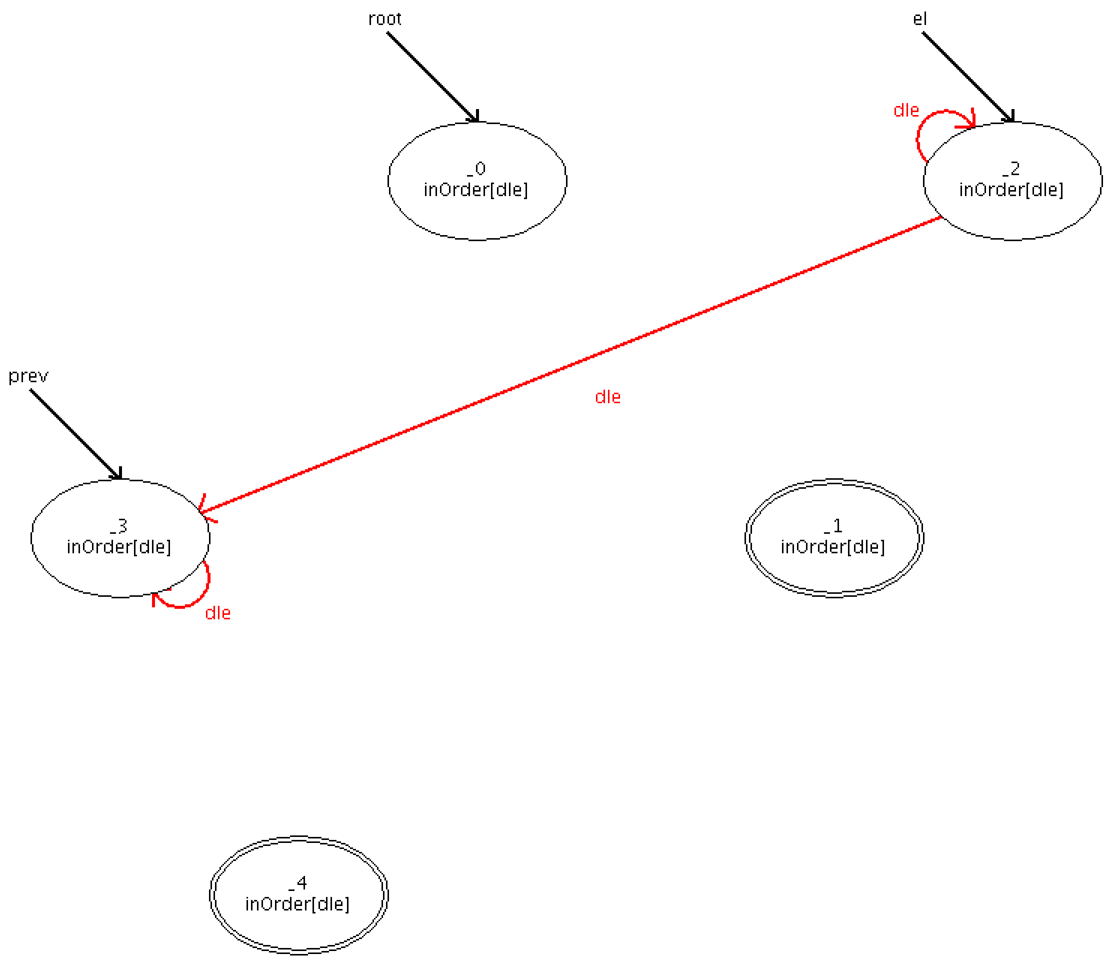


Figure 21: Another version of the shape graph from Figure 18. The nodes with the `prev` and `el` pointers were marked as important and only those edges are drawn that go between marked nodes.

In the example this might seem a bit minimalistic, but in bigger graphs this can save a lot of confusion.

Another useful feature is the possibility to give nodes different shapes. To do this the user can specify a formula for each shape, that when instantiated for the node evaluates to `true` will change the node to the respective shape. At this time there are two additional node shapes possible: Rectangles and octagons. These are useful to see certain important properties of nodes on first glance. Figure 22 gives an example for this. In the Section 4, we will see another useful application for this mechanism, namely to distinguish between nodes representing tree nodes and nodes representing list nodes.

For this simple example I chose the formula `r[el]`(v_1) for the rectangle shape and the formula `r[prev]`(v_1) for the octagon shape. The variable v_1 has to remain a free variable in the formulas, as this variable will be instantiated with the node in question. Note how the shapes add another possibility of uncluttering the graph: Although the reachability predicates `r[x]` are not shown, they are implicitly made visible by the shape of the nodes.

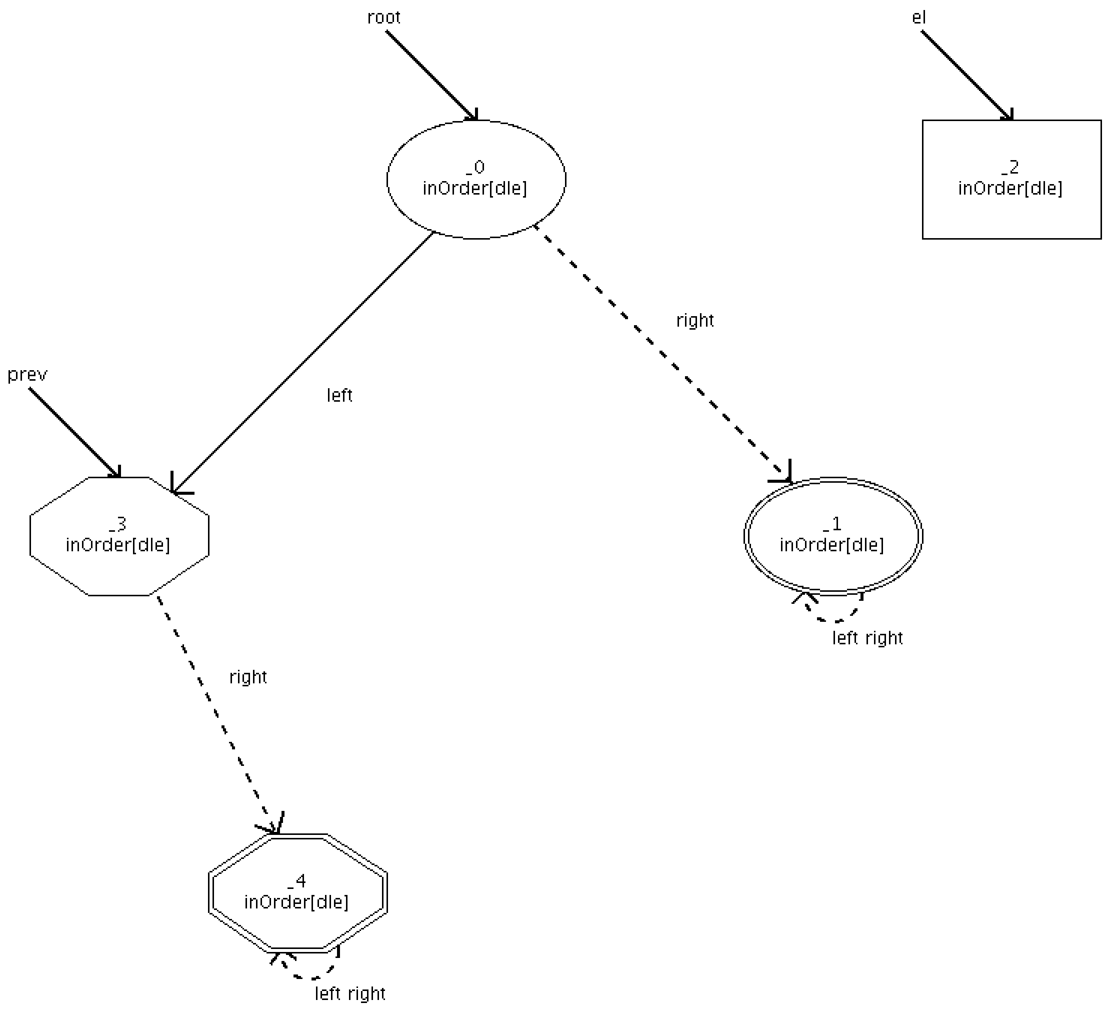


Figure 22: Yet again a version of the shape graph from Figure 18. This time the nodes reachable from `prev` will be drawn as octagons and the nodes reachable from `el` as rectangles. Note that the reachability predicate does not have to be shown for this to work.

Here is a short summary of the features affecting single shape graphs:

- Blending predicates in or out depending on what needs to be shown. This is a major way of uncluttering the graph by omitting help predicates that are of no use to human onlookers.
- Coloring predicates for easy recognition. While a great way to mark important or special predicates against others, it should be used with caution. There should never be more than four to six different colors, else it gets confusing again.
- Different node shapes to easily distinguish between important node properties. This also works when the defining predicates are not shown in the graph, thus giving another possibility to unclutter the graph.
- Marking of Nodes to show only those edges between marked nodes or edges that come from or go to those nodes. This is good to see only certain edges in the graph if it cannot be uncluttered by other methods.
- Although not shown in the examples here, there is also the feature to invert the edges. This is more of a debugging tool, but it can sometimes also be a good way to show what is not there, if the graph is cluttered again.

2.2 Navigating through Shape Graphs

So far we have only seen some of the common display problems with single shape graphs. Now let us take a step back and remind ourselves, that an analysis consists of whole sets of shape graphs. How do we order and navigate through these shape graphs?

In the default tvs-output of TVLA shape graphs are grouped by program point, i.e., we first get all shape graphs belonging to program point 1, then all of the next program point and so on. There is no way to tell when a specific shape graphs was created or even in what order they were created. Also there is no way to tell which shape graph is a successor of another shape graph, i.e., was created by applying a TVLA action on another shape graph. The only order and identification we can reliably use here is the appearance in the output file, i.e., we enumerate the shape graphs in the order they appear in the output file. This number we call the *index* of the shape graph. Shape graphs can be jumped to by their index in our visualizer and we also have a previous and next button to decrement or increment the index respectively.

While we now can argue over certain shape graphs, it should be immediately clear, that this kind of ordering is not very satisfying, neither for debugging, nor for teaching purposes. In very small analyses you can still reasonably easily see which shape graphs are in a successor and predecessor relationship, but this gets harder very quickly with a rising number of program points and shape graphs. No one wants to check through hundreds of shape graphs to see which one might fit.

The developers of TVLA also saw this problem and implemented a second kind of output, the so called tr-output, which is short for trace. While the basic structure of the file is still the same (graphs are still ordered by program point), the whole structure

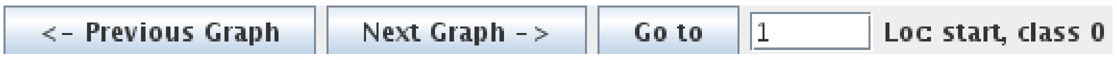


Figure 23: The buttons that allow navigation through shape graphs by their index. Also on the right is a label that shows which program point the shape graph belongs to. The class index means the position of the shape graph within the program point, i.e., class 0 means it is the first one in the output file that belongs to that program point. The class index also has a deeper meaning that will be explained in Section 3.2.

is now in XML-format. Every shape graph now gets a unique label and at the end of the file there now exists an edge listing. The edges define that shape graph SG_2 was created by using action a on shape graph SG_1. With this information it is now possible to actually follow an abstract program execution. The key word here is *abstract*, as with summary nodes, the abstraction can cause some seemingly strange behavior. It is therefore possible to run indefinitely through a loop without anything ever changing. This is due to the fact, that every shape graph can have multiple successors (we already saw that when explaining how focus works in Section 1.3), i.e., if there are multiple valid instantiations for the focus formula then the shape graph will have multiple successors if the resulting shape graphs are not also isomorphic. It is also possible, that there are multiple predecessors for a given shape graph. This is due to the fact that isomorphic shape graphs are considered the same shape graph, thus if different actions or different loop executions yield the same abstract shape graphs then that abstract shape graph will have multiple predecessors.

With the information in the tr-file we can now properly compute the so called *trace graph*. In this graph nodes represent whole shape graphs and edges stand for the successor relation and are annotated with the action that was taken. Figure 24 shows a small example of such a trace graph. The y-coordinates in the trace graph represent the program locations in the order they appear in the output file, i.e., all shape graphs on the same coordinate belong to the same program location. Not shown in the screen-shot is the small preview window, that shows the currently selected shape graph (marked in red in the trace graph). The successors of the currently selected shape graph are marked in green and its predecessors in yellow. It is possible by simply clicking on the nodes to move through the shape graphs.

The problem with the trace graph is that it gets unwieldy very quickly when the analysis is big. In Figure 25 we can see an example with 3573 shape graphs. While an experienced user can see some structure in there, like loops and special cases, the graph is a lot too big to be of use if we want to see local details around a certain shape graph.

With this problem in mind, I implemented a more local version of the trace graph, called *trace window*. The trace window is always centered on the current shape graph and will show the trace graph around the current node up to a configurable depth (default is 2). That means it will show the direct successors and predecessors and their respective successors and predecessors etc. It will not show other shape graphs at the same program point. An example can be seen in Figure 26. As in the trace graph itself, this window

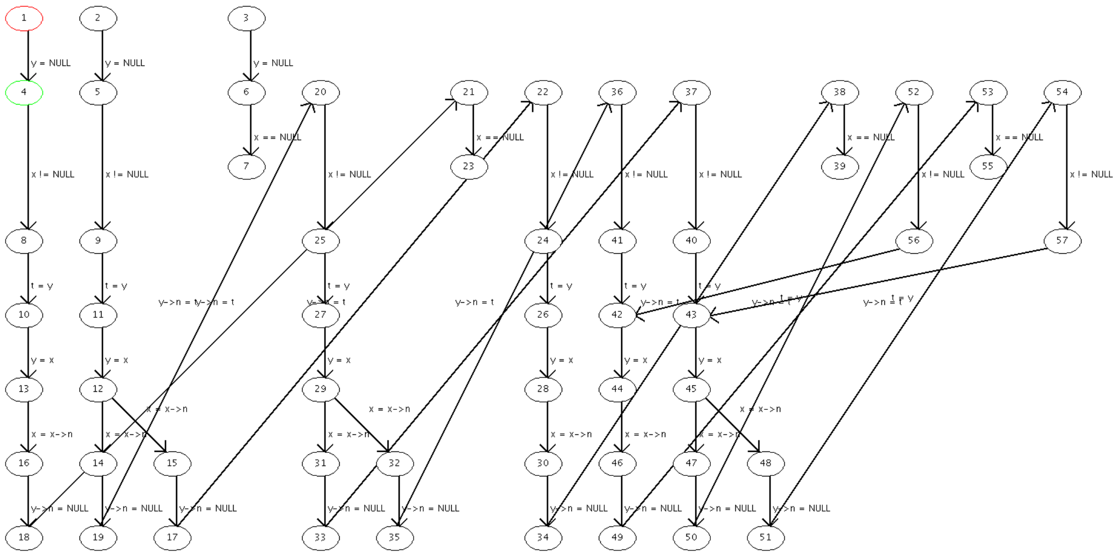


Figure 24: The trace graph for a small reverse list example analysis. Edges portray the successor relationship, annotated with the action that was taken. Nodes with the same y-coordinate belong to the same program point.

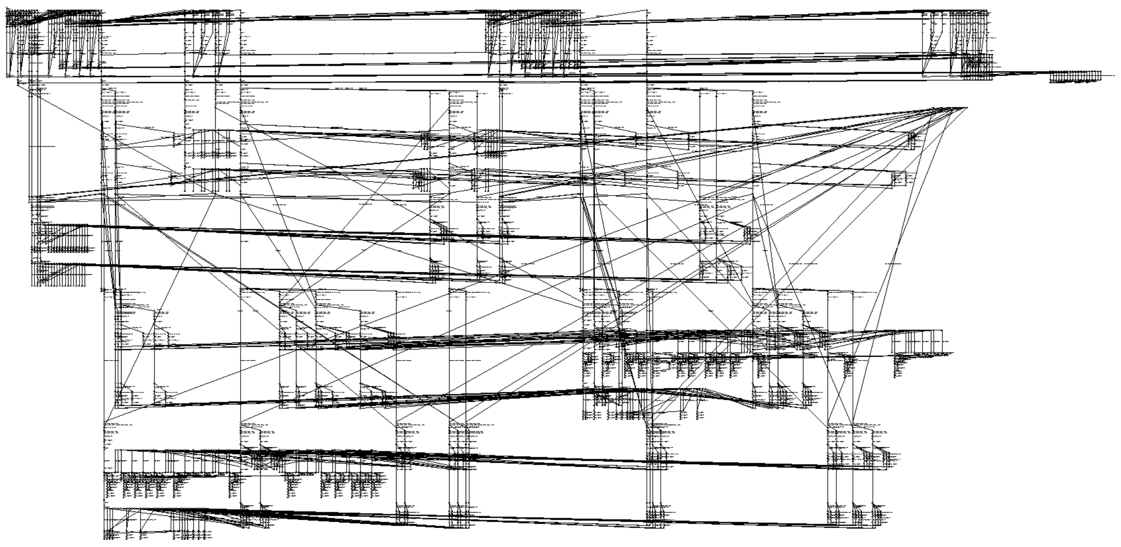


Figure 25: The trace graph for an AVL-tree insertion algorithm with over 3500 shape graphs. While it is still possible to see some structure, the local details are lost in the huge graph.

can be used to navigate directly through the shape graphs by simply clicking on them. That the successors and predecessors are not simply drawn as nodes, but as mini shape graphs is another useful feature, because it helps to decide which path to take through the algorithm. This is especially useful to see on a glance which are the special cases and which are the more general ones.

As a further convenience there are the "<<" and ">>" buttons, that allow to quickly switch between shape graphs in the current location, that share the same predecessor as the current shape graph. If the current shape graph is the only successor of its predecessor then these buttons have no effect.

In Addition to jumping to the index of the shape graph it is also possible to jump to the unique label of a shape graph. Sometimes, however, the user may not know the label or index of a shape graph, but rather can specify the shape graph by its logical properties, i.e., by logical formulas that hold for it. Conversely sometimes a user may want to check that there is no shape graph in the analysis where a certain formula holds (because this may indicate a possible error). For this purpose there is a search tool in our Visualizer, that allows the user to search for shape graphs where a certain formula holds, for empty shape graphs, or - in the case of debugging - for shape graphs with warning messages.

2.3 Miscellaneous Features

There are some more features in the Visualizer that do not exactly fit into the previous categories:

- Zoom: A useful standard feature in many programs. It is of even more importance here, because the shape graphs are dynamic and can change their display size rapidly and unpredictably depending on the number of nodes and the number of non-`false` unary predicates. This is due to the fact that unary predicates are drawn inside the nodes, the more there are the smaller the text or the bigger the node has to be. Therefore the user has the option to always automatically fit every shape graph to window size. Of course fitting the whole shape graph in the window may yield unreadable small nodes and text. Thus it is possible to manually zoom in and out using the mouse-wheel or to hold down the right mouse button and draw a box that will then be zoomed to window size.

- Magnify single nodes: Tied to the problem described above, when it is important to see the whole structure of the shape graph, but also the details of some nodes. There is the option to have a magnification effect when hovering the mouse over a node. A bigger and clearly readable version of the node will appear right next to it. It is of course configurable how long the magnification node hovers and the maximum scale it will appear at all.

- Font size: Also a standard feature in many programs. In my Visualizer its main purpose is for shape graphs with many nodes but few predicates. The many nodes mean that we have to zoom out if we want to see them all, but since there are only few predicates we can just increase the font size, so we can still read them.

- Manually color nodes: Pressing the C-key while hovering the mouse over a node will bring up a color slider menu, that allows the user to set a new background

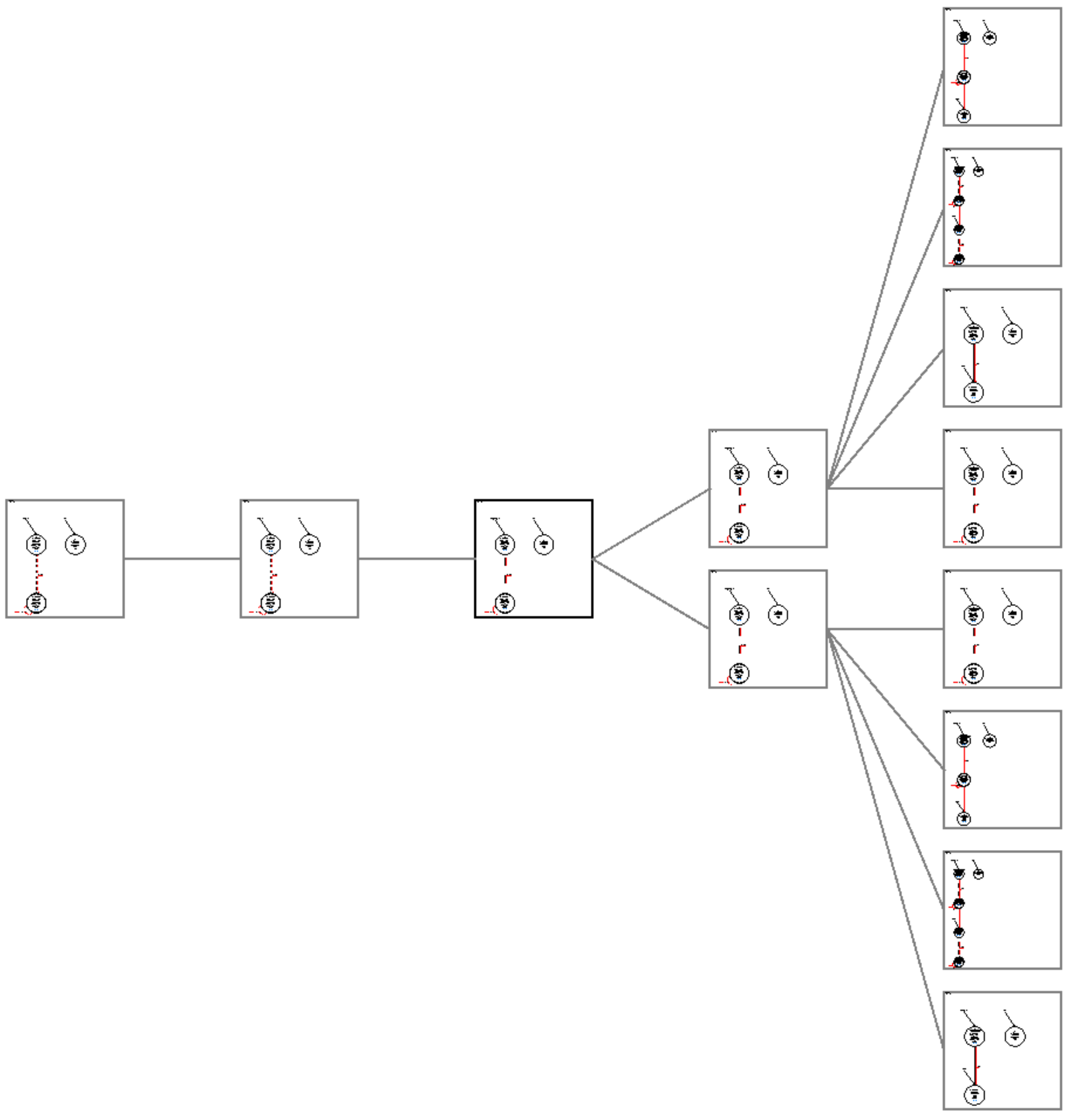


Figure 26: The trace window is a more local view of the trace graph and only shows reachable successors or predecessors up to a configurable depth. The successors and predecessors are drawn as mini shape graphs, so the user can already see the structure of the resulting shape graphs.

color for that node. This is a nice feature to mark nodes in a presentation. Note that the outline color of the node will not change, because these are used for a variety of other purposes as will be explained in later sections.

- Manually move nodes: While there is a very powerful specification tool in place for the layout of the shape graphs, there always can be problematic nodes that just do not look good where they are. Thus it is possible to just drag and drop nodes anywhere on the grid. Note that once you dragged a node somewhere the layout of the graph will be changed to “manual” and that graph layout will no longer be automatically computed. In other words the Visualizer will remember your layout even if you move to other shape graphs and come back again. Of course there is a button to reset the node layout to automatic.

- Cosmetic changes: Right clicking on the outline of a node will move the self loops to that point. Note that indefinite self loops are always 90 degree away from definite self loops. Holding shift and left clicking on a node will mirror the pointer arrow to the other side. This makes it easier to keep edges from going through each other.
- Different Edge layouts: The Visualizer supports two kinds of edge layouts right now. The standard one are just straight edges from middle point of one nodes to middle point of the target node. While the most primitive and the most clutter prone, experience has shown that this is indeed a much preferred method in our shape graphs, which mostly depict lists and trees. There is also the option of using a channel based orthogonal grid layout for edges, similar to CAD programs.

2.4 Shape Graph Creation Tool

There is one more tool that I will describe here in detail due to its great versatility and importance in creating new shape analysis specifications. I call it the Shape Graph Creation Tool, because that was the initial motivation of coding it. To understand this motivation properly I will remind the reader on how shape graphs are specified for TVLA by posting another example from their folder:

```
%n = {u, us}
%p = {
        sm = {us:1/2}
        root = {u}
        varSel[root,left] = {us:1/2}
        varSel[root,right] = {us:1/2}
        left = {u->us:1/2, us->us:1/2}
        right = {u->us:1/2, us->us:1/2}
        down = {u->us:1/2,us->us:1/2}
        downStar = {u->u, u->us, us->us:1/2}
        r[root] = {u,us}

        //Sortation predicates

        inOrder[dle,left] = {u, us}
        inOrder[dle,right] = {u, us}
        inOrder[dle] = {u, us}
        dle = {u->u:1, u->us:1/2, us->u:1/2, us->us:1/2}
        cmp[dle,right] = {u->u:1, u->us:1/2, us->u:1/2, us->us:1/2}
        cmp[dle,left] = {u->u:1, u->us:1/2, us->u:1/2, us->us:1/2}
}
```

This tiny shape graph is the input shape graph for a deletion algorithm in a sorted binary tree. Note that there are only two nodes in this shape graph and yet we already have four entries in `dle` and the `cmp` predicates. All those predicates that model a relationship between all nodes, like a sortation will have quadratic many entries in the specification and to a slightly lesser degree this is also true for the transitive closure of the reachability predicates. It should be immediately obvious, that is it very unintuitive to specify shape graphs like this and of course it is also very error prone. Forgetting to set one of these predicates right for even one case will invalidate the whole analysis and will usually produce an error somewhere down the line that is hard to track back to the input shape graph.

Since the Visualizer already could display shape graphs and because doing things visually greatly reduces the error chance, I implemented a tool that just lets the user create their own shape graph from scratch or editing an existing one by just doing a few mouse clicks.

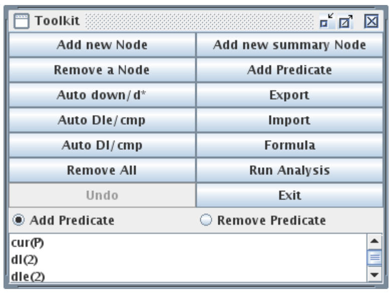


Figure 27: The Shape Graph Creation Toolbox. With these few buttons it is easy to create your own shape graph directly in the Visualizer, where you can check visually whether or not it looks like it should.

Figure 27 shows the toolbox for the shape graph creation window. It is divided in the button panel and the predicate list.

The predicate list contains all predicates in the current analysis in alphabetical order. The tag in parentheses indicates whether it is a nullary(0), unary(1) or binary(2) predicate and also makes a special distinction for predicates that simulate stack pointers into the heap(P). This distinction is made solely for visual reasons (since stack pointer predicates are drawn differently than pure heap pointers), pointer predicates can otherwise be thought of as a normal unary predicate. The user can add predicates into the shape graph simply by selecting the predicate in the list and then clicking on a node in the unary case or clicking on two nodes in the binary case. In the case of a nullary predicate they can click anywhere in the graph. The same technique works for removing predicates, the only difference is that the radio button must be set to remove predicate.

The button panel allows to user basic functionality like adding or removing nodes from the shape graph, but it also adds some powerful and convenient features:

- Add Predicate: This simple feature allows the user to add new predicates into the predicate list. All that is needed is the name of the new predicate followed by the arity in parentheses.
- Auto down/d*: This is a very convenient feature for specification of standard binary trees. Once the user has build a binary tree using just the `left` and `right` predicates, this function will automatically compute all the necessary `down` and `downStar` predicates that result from this tree structure. Thus it eliminates one of the most error prone parts of specifying a shape graph, it is very easy to forget a self loop or some reachability.
- Auto dle/comp: Similar to the auto down/d*, this function computes all the necessary predicates for sortation, i.e., the `dle`, `comp` and `inOrder` predicates. This is an even greater source of possible errors, since every two nodes in the graph need to have a relation set. Please note: This function needs both the `left`, `right` structure predicates and the `down`, prdownStar predicates set before it is invoked. Also this will only yield the desired result if the tree is actually supposed to be a standard sorted tree.
- Auto dl/comp: The same as above, just for the `dl` predicate, which was introduced for the AVL-tree analysis.
- Remove All: This function allows the user to completely remove all occurrences of the currently selected predicate in the predicate list from the shape graph. This is useful if the structure has become so hopelessly messed up, that a fresh start is needed.
- Formula: One of the most powerful tools in the arsenal. This allows the user to specify a kind of update formula for the currently selected predicate in the predicate list. As is usual for the Visualizer, the variables `v1` and `v2` are reserved and will be instantiated with all possible combinations of nodes in the shape graph. The predicate will be set according to the result of the evaluation. There is also the possibility to force the algorithm to use only definite values, i.e., if the formula evaluates to `unknown` the predicate will default to `false`. This tool is the generalization of the "auto" functions above and while powerful should be used with caution: It might require some trial and error till the right formula is found.
- Import/Export: Allows the user to export their newly created shape graph into the usual tvs format that then can be directly used as input for TVLA. Conversely import can be used to continue editing a saved shape graph. There is one more use for the import though: Depending on the TVLA version, the debug output or error message may not be in proper tvs format or the output may be really big, although the one thing of real interest is the shape graph at the very end where

something went really wrong. Copy and pasting the offensive shape graph into a separate file which is then imported is then usually the most convenient and time saving way to display the error graphically.

- Run Analysis: This is a special feature, that will call TVLA with the currently loaded specification and the current shape graph as input while deactivating the blur operation. Thus it can be used to compute a “concrete” example of the current program. More on the intuition behind this and its use can be found in Section 3.3.

The process of editing or creating a shape graph can be aborted at any time. Note, that a modified or newly created shape graph will be lost unless a modified shape graph was exported or the run analysis feature was used. While the intent with the shape graph creation tool was solely on helping the user create input shape graphs, it is also a simple way of playing around a bit with shape graphs.

2.5 Formulas in the Visualizer

Logical formulas are an important specification method for our Visualizer: They are used for partitioning, sortation, node shapes and search tools, to name just the most important ones. Thus this section will discuss the syntax and some peculiarities of our formulas. Since the Visualizer was developed with TVLA in mind, we adjusted the notation, such that it is as similar as possible to TVLA notation. The goal was that it is possible to copy and paste a formula from TVLA directly into the Visualizer and have it evaluate in the same way. There are some important exceptions to this:

- In our Visualizer there is no transitive reflexive closure, we use strictly first order predicate logic. The closure is needed in TVLA to define predicates that can express reachability, these predicates can be used in Visualizer formulas, but not their definition.

- There is no possibility at the moment to define sets of variables and thus no need to implement the `FOREACH` construct present in TVLA. We found no real application for this mechanism that would help with visualization, but this may be a feature in future updates if the need arises.

- The variable names `v1` and `v2` should be considered as special reserved names that should only be used in formulas that actually are intended to use them and they should always be used as free variables. For example, if the sortation formula evaluates to `true`, then the node instanced as `v1` will have a smaller coordinate than the node instanced as `v2`. If a formula is not intended for sorting or node shape, then the variable names `v1` and `v2` should be completely avoided or side effects might occur.

- Looking down at the grammar in line (8), there is the >= operator, this operator does not exist in the original TVLA notation. The point of this operator is pure syntactic convenience: Instead of the actual logical values, we consider the values as numbers, thus they are ordered 1 >= 1/2 >= 0. That means writing `reach(x)`>=1/2 is just a convenient short for `reach(x)`==1/2 | `reach(x)`==1.

- Also in line (8), the operator == behaves differently than in TVLA. In TVLA and normal three valued logic the formula 1/2==1/2 would evaluate to `unknown`, whereas in our Visualizer it will evaluate to `true`. This has a very practical reason: If we want to specify what should happen visually if a predicate or formula is `unknown`, then we need to have some way to test whether a predicate (or formula) has that value or not. A test in traditional three valued logic, however, would always evaluate to `unknown`, not true or false. In other words, 1==1/2, 0==1/2 and 1/2==1/2 logically always evaluate to `unknown`. Our == operator will yield `true` if both truth values are syntactically the same and `false` else. Note that the equivalence operator from line (3) will provide the same results than the original == operator from TVLA. So if it is desired that the results of the formula stay the same as in TVLA, then the == operator should be replaced by <->.

The grammar below shows the syntax of our logical formulas. The start symbol is Σ_0, characters between quotation marks are the actual characters as they appear on the keyboard. *id* is the name of a logical variable, these names can consist of any combination of characters, including special characters, with the exception, that they may not contain the logical operators and may not start with 0 or 1. The exception to the exception are the two quantors, it is of course allowed to use the letters 'e' and 'a' in variable names. *idList* is a comma separated list of logical variables. *p* is the name of a predicate, the rules for predicate names are the same as those for *id*.

$$\Sigma_0 \rightarrow \text{'E'}(idList)\Sigma_0 \mid \Sigma_1 \mid \Sigma_2 \tag{1}$$
$$\Sigma_1 \rightarrow \text{'A'}(idList)\Sigma_0 \tag{2}$$
$$\Sigma_2 \rightarrow \Sigma_3 \text{ '<->' } \Sigma_3 \mid \Sigma_3 \tag{3}$$
$$\Sigma_3 \rightarrow \Sigma_4 \text{ '->' } \Sigma_4 \mid \Sigma_4 \tag{4}$$
$$\Sigma_4 \rightarrow \Sigma_5 \text{ '|' } \Sigma_4 \mid \Sigma_5 \tag{5}$$
$$\Sigma_5 \rightarrow \Sigma_6 \text{ '\&' } \Sigma_5 \mid \Sigma_6 \tag{6}$$
$$\Sigma_6 \rightarrow id \text{ '!=' } id \mid id \text{ '==' } id \mid \Sigma_7 \text{ '!=' } \Sigma_7 \mid \Sigma_7 \text{ '==' } \Sigma_7 \mid \Sigma_7 \text{ '>=' } \Sigma_7 \mid \Sigma_7 \tag{7}$$
$$\Sigma_7 \rightarrow !\Sigma_8 \mid \Sigma_8 \tag{8}$$
$$\Sigma_8 \rightarrow \text{'0'} \mid \text{'1'} \mid \text{'1/2'} \mid \text{'('}\Sigma_0\text{')'} \mid p\text{'('}id\text{')'} \mid p\text{'('}id\text{','}id\text{')'} \mid p \tag{9}$$

3 Understanding Shape Graphs

In the previous section we have seen a number of tools and their applications and in the process we gained some insight into some of the common problems with general shape graphs. In this section we will adress the more fundamental problems inherent in using shape analysis and shape graphs for visualization. There are a few very basic mechanisms that spawn all the other problems and those shall be adressed in the following sections. It should also be noted, that while I continue with examples from TVLA, the problems and mechanisms are mostly inherent in shape analysis and thus are independent from the tool used for computing the analysis. Only some of the more practical solutions described here may have to be modified to fit the respective tool.

- Abstraction: While absolutely necessary for shape analysis, this is one of the major troublemakers. Abstracting nodes all over a given data structure into a single node will usually cause big problems if the goal is to layout the nodes in a meaningful manner. This will be the main focus of Section 3.1.
- Number of predicates: While of course necessary for conducting a meaningful analysis, predicates should be used with forethought and caution. The more there are, the more confusing the analysis becomes and the more potential shape graphs may accumulate, which again makes the analysis harder to understand and debug. Thus the general rule is to use as few predicates as possible. Section 3.1 shows some examples of this.
- Number of shape graphs: This is of course somewhat tied to the previous issues, but there comes a point when you just cannot remove any more predicates. If there are still too many shape graphs in the analysis, then another method to reduce their mass is needed. This is where partitioning comes into play. It allows the user to abstract whole sets of shape graphs into equivalence classes. Section 3.2 will explain this mechanic.
- Intuition: There are many arguments over what the best learning method is and it usually ends with different people prefering different methods. Some people may prefer abstract and formal thinking, while others prefer intuitive thinking and deduction from examples. Shape analysis is great for abstract people, but it is very hard on the intuitive side of things. That is why we provide a way to show semi-concrete examples of program executions. Section 3.3 explains this specific approach.
- Pseudo Code: While nothing new or spectacular, Section 3.4 will give a short overview of why generic pseudo code is especially useful for our Visualizer and how it is implemented.

3.1 Inherent Problems of Abstraction

In this section I will show the basic limitations of laying out and visualizing shape graphs, as well as propose strategies that can be employed to work around some of the problems.

But before I can talk about the problems of visualization, I first have to give a good idea what I actually mean with that word.

3.1.1 Visualization

Visualization is a very broad term that can mean anything from vague handsigns to very clearly defined mathematical graphs. Even if we just look at our Visualizer, there are different kinds of visualization present.

On the one hand there are all the things that have nothing to do with shape graphs at all. Everything related to the user interface falls into this category. There are buttons, windows, menus and a thousand other little things that are so common, that we often do not even consciously think of them as visualizations anymore. Yet their good or bad arrangement can have quite a big impact on the usability of an application. This is, however, exactly what we do not care about in this section.

On the other hand there is the data that is to be visualized, i.e., the shape graphs. A shape graph is just a logical construct, that consists of logical entities (the nodes) and of a set of truth values for logical predicates, that are defined on those logical entities. The basic problem is to read the logical description of a shape graph and create a visualization that respects the logical properties as much as possible. That means we have to assign some visual properties to the nodes, e.g., coordinates, shape, size, colors and many more. Also we have to assign properties to the predicates, again things like size and colors, but also whether or not they are drawn at all, what kind of edges to use (straight, rectangle, curved) and so on. Some of those properties may also depend on the currently viewed shape graph: The coordinates of a node is a good example of this, the same node may be drawn in different positions depending on the rest of the graph.

Some of these properties may not be easy to define formally, but a few are very straight forward. Coordinates, colors and shape are good examples of this. Given a shape graph and a function that assigns these properties to nodes and predicates, we may actually create a well defined visualization in the sense, that we can deduce logical values simply from visual cues. This is what we will discuss in great detail in the next sections.

3.1.2 Limitations and Problems of Abstract Visualization

Since our Visualizer is intended for teaching purposes, the representation of the data on the screen should be as accurate and unambiguous as possible, we don't want to goad the user into wrong conclusions.

To generate screen coordinates for the nodes, the user assigns a formula $\varphi(v_1, v_2)$ to an axis and then a partial order is computed:

$\forall u_1, u_2 \in V_{SG}, u_1 \neq u_2 : u_1 < u_2 \Leftrightarrow [\![\varphi(v_1 \leftarrow u_1, v_2 \leftarrow u_2)]\!]^{SG} \equiv true$

So for every two nodes of a given shape graph the formula is evaluated and if the result is *true* then u_1 will have a smaller coordinate than u_2 on the respective axis. In the general case this will only produce partial orders, however if φ describes a complete order relation on the nodes, then the computation will of course also yield a unique and complete order on the coordinates (we need not concern ourselves with reflexivity, since all our nodes are unique). That means if we only consider concrete shape graphs, i.e., shape graphs without summary nodes, then our visualization will display the nodes in the logical correct order: If node u_1 is drawn left of another node u_2 then $\varphi_x(u_1, u_2)$ evaluates to *true* in the corresponding shape graph.

Note that for two different shape graphs SG_1 and SG_2 we can *not* conclude that $[\![\varphi(v_1, v_2)]\!]^{SG_1} \equiv [\![\varphi(v_1, v_2)]\!]^{SG_2}$ even if v_1 and v_2 have the same canonical names in both shape graphs. This is due to the fact that data values are in general not directly reflected in predicates and that binary predicates are not part of canonical names, but can be used in φ. A simple example of this would be the order of two nodes v_1 and v_2 that are ordered by a binary relation, that represents their data values. They both can switch their data values, thus inverting the binary relation between them, without changing their canonical names in any way. See Figure 28 for an illustration.

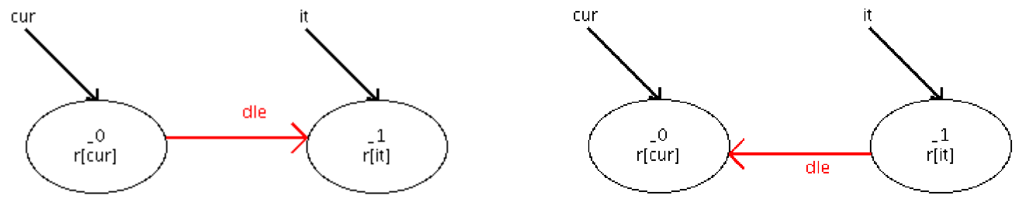


Figure 28: This Figure shows two nearly identical shape graphs. Both nodes have the exact same canonical name in either shape graph. The only difference is the `dle` predicate, that signifies that in the left shape graph the `cur` node's data value is smaller than the `it` node's data value and vice versa in the other.

So if we want to model the common less-than-relation on the x-axis we can do so and expect it to look like it should for concrete shape graphs. Unfortunately in a non trivial analysis, concrete shape graphs are usually only special cases, if they exist at all. Generally we have to work with abstract shape graphs. We will now look at how and when abstraction will cause problems.
Consider a sorted single linked list and once again we use the common less-than-relation for one axis. We choose reachability as abstraction predicates, as well as pointers, as usual, i.e., nodes that are reachable from the same pointers will be abstracted into the same summary node with the exception of the actual node that the pointer is on, which will remain unique. No matter how many pointer we use and where we place them, the abstract shape graph will still be completely ordered.
`Proof:` First we can merge all pointers that point to the same node, since all following nodes will be reachable from both, the merging (or removal of one) won't change the abstraction. This leaves us with distinct concrete nodes pointed to by a pointer and possibly summary nodes in between, that are reachable by one pointer, but not by the next one. If there are two consecutive concrete nodes, then the order is of course preserved since it is the same as in the concrete shape graph case. So the interesting case is that we have a concrete node v_1, pointed to by pointer p followed by a summary node v_2, followed by another concrete node v_3, pointed to by another pointer q as is shown in Figure 29. We have to make sure, that the summary node v_2 does not abstract

a concrete node that was before v_1 or after v_3, since that would destroy the ordering. Due to the choosen abstraction this cannot happen: Only nodes reachable by p and not reachable by q can be abstracted into v_2, so there can be no nodes before v_1 and none after v_3 be abstracted into v_2, thus the order is preserved.

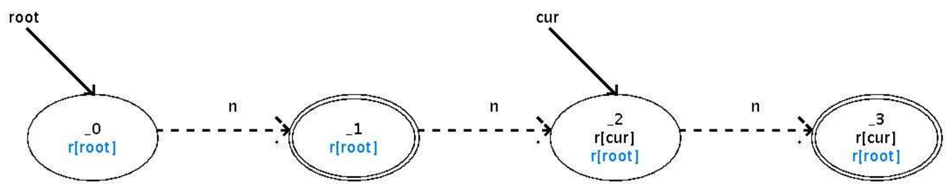


Figure 29: A sorted single linked list with `root` as head element and `cur` as iteration pointer. Due to using reachability (depicted as r[pointer] predicate) it is always possible to completely order the nodes.

Unfortunately this will not work for every abstraction. If we take the above example and extend the abstraction such, that even and uneven data values get abstracted into different summary nodes then it obviously is no longer possible to order the abstract nodes completely by the less-than-relation. If there are both even and uneven data values in the nodes between two pointers, the abstraction would generate two summary nodes between these pointers (one for even data values, one for uneven). And obviously there would be no right way to order these two summary nodes according to the less-than-relation.

We introduce the notion of a *concretization* of a shape graph, and, in extension, of a node. A concretization $c(G)$ of an abstract shape graph G is a new graph, where every summary node has been replaced by a set of definite nodes(at least one), we call such graphs concrete graphs. The new graph has to satisfy the following conditions:

- all predicates must be definite
- if the abstraction function is applied to the concretized graph(i.e., a blur operation of TVLA), then the resulting graph must be isomorphic to the original shape graph
- the concretized graph must satisfy all predicate definitions and restriction formulae of the analysis specification, i.e., a coerce operation should accept the graph and return it unchanged

We define $C(G)$ as the set of all possible concretizations of G. Analogously a concretization $c(v)$ of a node v is a set of nodes, that v could be replaced with and where the resulting graph satisfies the same three criteria as the concretization of a shape graph. We define $C(v)$ as all such sets of possible concretizations. Note: If v is not a summary node then the only concretization is the node itself. For every $G' \in C(G)$ exists exactly one corresponding $c(v)$ for all $v \in V_G$.

Definition 1. *An abstract shape graph SG is completely ordered under φ if and only if for every pair of abstract nodes*

1. $\forall v_1, v_2 \in SG, v_1 \neq v_2 : [\![\varphi(v_1, v_2)]\!]^{SG} \equiv true \vee [\![\varphi(v_1, v_2)]\!]^{SG} \equiv false$

2. $\forall v_1, v_2 \in SG, v_1 \neq v_2 : [\![\varphi(v_1, v_2)]\!]^{SG} \equiv true \Rightarrow [\![\varphi(v_2, v_1)]\!]^{SG} \equiv false$

3. $\forall v_1, v_2, v_3 \in SG, v_i \neq v_j : i \neq j : [\![\varphi(v_1, v_2)]\!]^{SG} \equiv true \ \wedge \ [\![\varphi(v_2, v_3)]\!]^{SG} \equiv true \Rightarrow [\![\varphi(v_1, v_3)]\!]^{SG} \equiv true$

A set S of abstract shape graphs is completely ordered under φ if and only if every shape graph $SG \in S$ is completely ordered.

Definition 2. *We call an abstraction order preserving under φ for a set S of shape graphs if and only if φ induces a complete ordering in every concretization of every shape graph $G \in S$ and if S is completely ordered under φ.*

In other words, if a concrete shape graph was completely ordered under the formula φ then the resulting abstract shape graph must still be completely ordered under φ. Formally for a summary node that means it can be ordered if all of its abstracted nodes are either smaller or bigger than any given other node of the shape graph(according to φ). If it is clear what order relation is meant, we will omit the under φ part.

For the following theorem we assume that when the nodes of a graph are completely ordered, then we will enumerate them in that exact order from first to last. If we refer to an interval of nodes then we mean all nodes from a start index through to the end index.

Theorem 1. *An abstraction is order preserving under φ for a set of abstract shape graphs S if and only if every summary node in every abstract shape graph $G \in S$ can be described as a single interval of ordered nodes in all concretizations of G and all concretizations of G are completely ordered by φ.*

Proof. *"$\Rightarrow$": All concretizations of G must be completely ordered by φ due to the definition of order preserving.*

To show that all summary nodes can be described as a single interval, we shall assume that there was one summary node s_1 that consists of at least two intervals I_1 and I_2. Let $j \in I_1$ and $l \in I_2$. There must exist at least one k, such that $j < k < l$ and $k \notin I_1, I_2$. Let k belong to a node s_2, that may be concrete or a summary node. We can observe three cases:

- *case 1: we order $s_1 < s_2$, this is a contradiction to $k < l$*
- *case 2: we order $s_1 > s_2$, this is a contradiction to $j < k$*
- *case 3: we order $s_1 = s_2$, this is a contradiction to $j < k < l$*

thus we have a contradiction to our assumption that the abstraction is order preserving.

"$\Leftarrow$": Again all concretizations of any G are completely ordered by φ due to our premise and we just have to show that having a single intervall for every summary node is enough to guarantee that we can order the abstract shape graphs according to φ.

Two summary nodes s_1 and s_2 with $s_1 \neq s_2$ can be ordered by φ iff
$\forall v_1 \in C(s_1), v_2 \in C(s_2) : [\![\varphi(v_1, v_2)]\!]^G \equiv true$
$\vee\ \forall v_1 \in C(s_1), v_2 \in C(s_2) : [\![\varphi(v_1, v_2)]\!]^G \equiv false.$
Note that this also includes the case that either of the nodes is actually a concrete node. Both s_1 and s_2 abstract nodes in one given index interval, these intervals must be disjunct, since every node can only be abstracted into one summary node, thus either all indices of s_1 are bigger than s_2 or vice versa. That directly implies the above condition is fulfilled (if $\varphi(v_1, v_2) \equiv true$ then v_1 must have a smaller index than v_2).

So now that we have a criterion for generating a good visualization, we have to ask if there always exists such an order preserving abstraction for a given φ for any algorithm and data structure. The answer to that question, unfortunately, is no, there does not. We will illustrate this by looking at a simple search algorithm in a sorted binary tree. We choose φ such that it uses the sortation of the tree to position the nodes, i.e., if a node v_1 is smaller than a node v_2 then and only then w.l.o.g. v_1 should be drawn left of v_2.

Theorem 2. *There exists no order preserving abstraction for an analysis of a standard search algorithm in a sorted binary tree with the following constraints:*

- *φ emulates the sortation of the tree, i.e., the node with the smaller data value has always a lower coordinate than any node with a higher data value.*
- *The nodes on the path from the root to the current node are clearly marked and may never be abstracted with non path nodes.*
- *A node that is pointed to by some program pointer (in contrast to heap pointers) will always be concrete, i.e., not be abstracted by any summary node.*

Proof. *Let us consider the case that our search algorithm always takes the left child in its search. Let us further assume that we are in the general case and every node except the leaves have two subtrees. We then have a situation looking like Figure 30.*
Note that this shape graph is just one (minimal) concrete example, the following argumentation holds true for all other examples as long as every i-th node (with $i \geq 1$ some constant) on the path has a right subtree.

The square nodes in Figure 30 are nodes on the path from root to the current pointer. Due to our precondition no square node may be abstracted together with a round node. Between each two (right) subtrees hanging from some path nodes, there is at least one path node that lies between the intervals of the subtrees. Since the path node itself may not be abstracted together with the subtrees, no two subtrees can be allowed to be abstracted together, or our abstraction is no longer order preserving. Furthermore, if we consider two path nodes and there exists at least one right subtree with its parent on that path (with the exception of the very first path node) then we again must not allow them to be abstracted together.

Thus in general we have to avoid abstracting both path nodes with each other and non path subtrees with each other. Now consider, that an abstraction always has a limit as

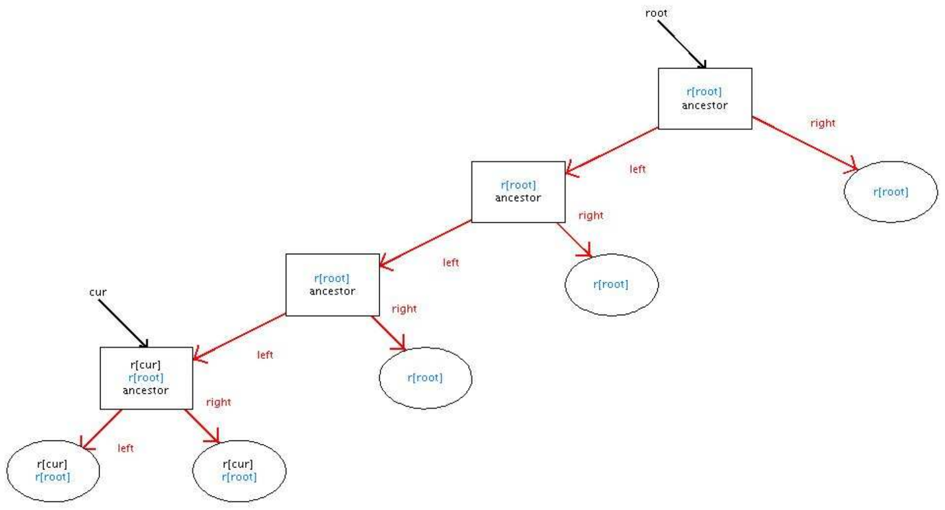


Figure 30: The resulting shape graph if the search algorithm always takes a left. Square nodes are path nodes.

to how many nodes can be present at most in any given shape graph. Since abstraction predicates are always definite, there are exactly $2^{|A|}$ *possible nodes in a shape graph, with* A *being the set of abstraction predicates. So no matter what abstraction predicates are chosen, we can always iterate the search algorithm some more until we come to the point where we have to abstract nodes together that are not allowed to.*

Since there exists no non-trivial order preserving abstraction for even such a relatively simple algorithm as searching in binary trees, it is reasonable to assume that we generally will not find one for non-trivial algorithms and data structures. Thus we have to deal with nodes that cannot be ordered with some other node, which we will call *conflicting nodes*.

First we consider how many conflicting nodes there can be in a given shape graph. Obviously the minimum amount is two, there simply cannot exist a single conflicting node. It is usually easy to construct examples where all the nodes in the shape graph are conflicting nodes. Unfortunately even in normal analyses, it is often the case that all or nearly all nodes are conflicting, due to catch-all summary nodes, i.e., everything that is of no interest to the analysis, but part of the data structures, will be summarized in that summary node, this often leads to abstraction of nodes that were previously scattered all over the data structure. In such a case the easiest solution is to mark the one catch-all summary node as the troublemaker, indicating, that the rest of the nodes would be fine if that summary node didn't exist.

This leads to the following idea of dealing with conflicting nodes: We determine the minimal set of conflicting nodes C, that need to be removed from the shape graph in order to resolve all conflicts and mark them. We can then lay out the rest of the nodes without problems and place the marked nodes at a convenient position. C is generally not unique, which means that we either pick an arbitrary set or we have to compute all possible minimal sets C. Consider the simple example in Figure 31, it's a binary tree which we traverse with the `cur` pointer. Node 0 is the root, Node 1 is the left subtree of the root, Node 2 are all the nodes lying on the path from the root to `cur`, Node 3 are all the subtrees hanging off the path, Node 4 is the node the algorithm is currently considering and Node 5 represents the yet not traversed subtrees of the aforementioned node.

We assume that we want to order the nodes by their data values on the x-axis (less-than-relation) and by father-child relation on the y-axis (children should always be drawn below their parents, we shall use the reachability predicate for this). Strictly speaking the second relation is no order relation at all, since it is not reflexive (we don't care much about that) and because it is not complete (we do care a lot about this), for example two subtrees are not comparable under father-child or reachability. For the sake of this example we will ignore that it is no real order relation and just relay what our algorithm would do, we will come back to this a bit further down. There are a number of conflicting nodes under these relations:

- Node 2 and 3 conflict in both the x- and y-axis. The reason for the x-axis conflict is the same as in Figure 30 und the y-axis conflict is because while it is true, that every root of every subtree hanging off the path is the child of some node on the path, it is not true that every root of every subtree is a child of all the nodes on the path and indeed many of those roots are not even reachable by all nodes on the path.

- Node 4 and 5 conflict in the x-axis, because node 5 abstracts both left and right subtree of node 4. They might even conflict in the y-axis if we just specified the `left` and `right` as our sortation formula, because both of these are indefinite values. It makes more sense to use reachability, i.e., node 5 is reachable from 4 and not vice versa. This will yield a defnite value. Note that reachability will not help with the conflict between node 2 and 3, because not all subtrees on the path are reachable from every node on the path and no node on the path is reachable from any subtree.

- Node 1 and all other nodes except node 0 conflict in the y-axis, as they are incomparable (not reachable from each other).

- Node 3 and nodes 4 and 5 conflict in the y-axis as these too are not reachable from each other.

- Nodes 2 and 3 conflict with Nodes 4 and 5 in the x-axis. Despite what the figure initially suggests, the path abstracted by node 2 has both left and right self pointers, that means the search path went zig-zag and thus that there are both nodes with bigger and smaller data values on the path.

So this means that every node except the root is marked as a conflicting node. Now consider which possibilites there are for minimal sets of conflicting nodes that allow us

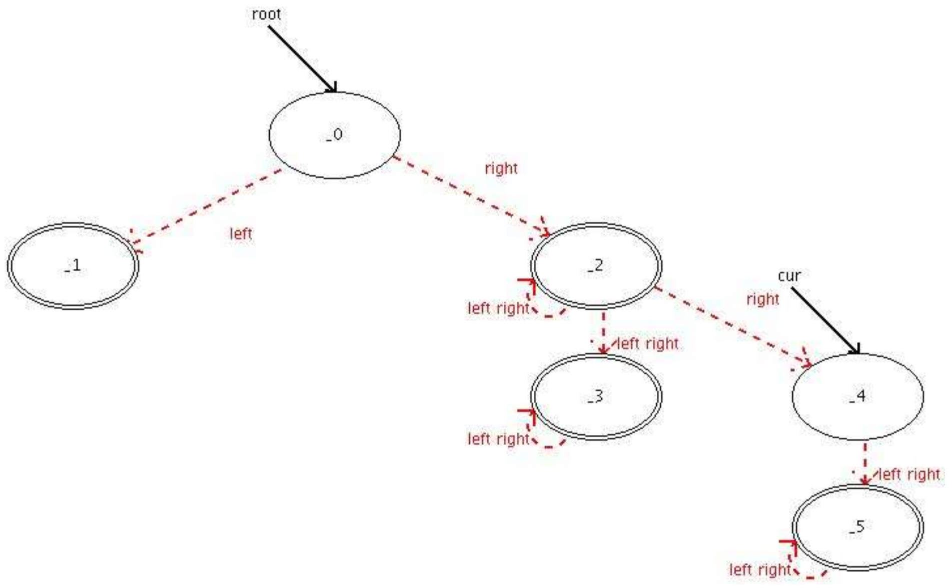


Figure 31: A shapegraph for an algorithm that traverses a binary tree. We use this example to illustrate how we solve conflicts.

the layout the rest without further problems. Since node 2 and 3 conflict with each other, at least one of them has to be in the minimal set, the same is true for 4 and 5. Also node 1 has to be included, or we would have to remove 4 other nodes. The minimum set in this simple example is already not unique: We can mark node 1 and then three out of 2, 3, 4 and 5. The latter is neccessary, because 2, 3, 4 and 5 are pairwise conflicting. In this example we decided to go for 1, 2, 3 and 5 as the minimal set to be marked. Once this is done we can layout the rest of the nodes without problems. Unfortunately in our example, that means all we know for sure is that node 1 is left of node 0 and that again is left of node 4 and we know that all nodes are below node 0 and that node 5 is below node 4. After that we place the marked nodes where we think they make the most sense, i.e., by obeying as much of the partial ordering as we can. Interestingly this leads to a quite natural looking tree, which means it is the desired layout for our purposes.

Deciding to compute all minimal sets C is the right approach, we formally phrase the problem as follows: Construct a graph G(V,E), such that the nodes that needs to be layouted are V and $(v_1, v_2) \in E \equiv [\![\varphi(v_1, v_2)]\!]^{SG} \equiv true$. I.e., we use our order relation as edges. Now we have to find the minimal node set, that when removed, would eliminate all circles. This is exactly the feedback vertex set problem, which is NP-complete as proven by Karp [15]. Additionally we have to make sure that there is exactly one edge between every two nodes, which amounts to the same problem again. Note that for purposes of helping us with the layout of the marked nodes, we also consider those edges where the evaluation of the relation only yields `unknown`.

Computing the minimal sets C is only half the work. We already stated, that those minimal sets are generally not unique. Since the example tree looked so natural, we might be inclined to flag the sortation relation for the y-axis as a partial order relation only. It actually makes sense to describe the visual layout of certain data structures as only a partial order, trees are one example of this. For us this means in terms of layout a slight change, we already described how we generate a partial ordering from evaluating the sortation formula for every two nodes in a graph. We say a node v_2 *depends* on another node v_1 if and only if $[\![\varphi(v_1, v_2)]\!]^{SG} \equiv true$. I.e., if the sortation formula says, that node v_1 should be drawn before node v_2 then v_2 depends on v_1. Our layout algorithm will start assigning coordinate 0 to those nodes without dependencies, then remove those nodes from all dependency lists, then increase the coordinate and repeat. Since it is only a partial ordering, there might be no nodes with zero dependencies, i.e., two nodes may depend on each other because the formula yields true both ways. In this case we take the set of nodes with the lowest number of dependencies and assign them the same coordinate. The only conflict in this kind of layout is, if two nodes depend on each other in both axis, thus getting the exact same coordinates assigned. We can solve these conflicts in the same way no matter whether it is supposed to be an order relation or not.

Consider Figure 31 again. Now assume, that the y-axis is flagged as partial order. The only conflicts left now are those between nodes 2 and 3 and the one between nodes 4 and 5. To solve the conflicts we have to mark 2 nodes, one from each pair. In other words our minimal sets C now contain 4 possible sets: $\{2, 4\}, \{2, 5\}, \{3, 4\}, \{3, 5\}$ each of those will solve all conflicts. The question is now how to decide which set to take. Which feels the most natural? We already mentioned that the way the graph is drawn in the figure is the most natural way that we can think of. It is of course hard to decide in general which way is more natural or sensible or efficient for a data structure to look and even more so how to abstract that look in a sensible way. Thus we use three heuristics which produced good results in our examples:

- First heuristic: We give higher weight to concrete nodes than to summary nodes. Given the choice between marking a concrete node or a summary node as "wrong", we will always chose the summary node. The reasoning behind this is straightforward: Summary nodes mean abstraction, abstraction usually means loss of information and loss of information will make logical values indefinite making evaluating the sortation formula indefinite too. Or in other words, the concrete nodes are usually those with pointers on them, which means they are at that time important for the algorithm and therefore interesting for the user and will contain the most information. Thus it would feel wrong to mark such a node over some summary node that abstracts an arbitrary number of concrete nodes.

- Second heuristic: We give more weight to nodes with more binary predicates than to nodes with less. That means we count the number of binary predicates for a node v with $p(v, v_2) \neq false$ or $p(v_1, v) \neq false$. The reason here is that given two summary nodes (the first heuristic eliminated the concrete nodes), one of them might have more structural predicates than the other. As an example see nodes 2 and 3. Node 2 has more edges (binary predicates) than node 3, because it represents the path taken by the algorithm. It links the concrete parts together.

The most problematic summary nodes are often those, that abstract all the odds and ends of data structure that are of no interest, but need to be represented by the logcial analysis never the less. This heuristic tries to separate the important summary nodes from the unimportant ones.

- Third heuristic: We give more weight to nodes with more unary predicates than to nodes with less. The reason is again similar to the above and comes down to how much information is available. Our basic premise is, that the more interesting some node is, the more information the analysis should have about it. Because if we want to prove certain properties then it is logical to make sure to have as much pertinent information as possible while ignoring unimportant information in order to not unnecessarily complicate and clutter the analysis.

Since these are only heuristics it is of course possible to find examples where the intuition is wrong and will lead to a bad layout. This is especially true if the analysis is actually constructed in such a way as to exploit the weaknesses of the heuristics. This however brings us to another important point: While it is certainly true, that we can construct the analysis in such a way, as to hinder our visualization, the reverse is also true. Specifying an analysis with the idea of visualization in mind can help to avoid a lot of the problems we discussed so far. As a simple example we compare the analysis for the insert operation of an AVL-tree with an analysis for all the usual heap operations on pairing heaps: The first one has some 20,000 to 30,000 shape graphs, the latter only around 200. It is clear by the mass of shape graphs alone, that the first analysis will be hard to visualize without confusing the user.

3.1.3 Visualization Friendly Analysis

One of the biggest problems of using shape analysis for teaching purposes is the sheer amount of information it can provide. On the one hand there are the various predicates used in the analysis: For every actually interesting (for the user) predicate, there are often two or more predicates that are only there to help with computing the interesting ones or keep their information as definite as possible. These helping predicates are often tied very intricately into the working of TVLA and the specification of the analysis. They have to be there to make it work, but to someone who does not know TVLA or the specific analysis they are often totally incomprehensible and massively confusing. Our visualizer can mend this problem by simply not drawing those predicates, this needs some user input (the visualizer cannot decide which predicates are the interesting ones), but since the number of interesting predicates is usually quite low it's easy to list the few predicates that should be shown and blacklist the others.

The other dimension of information amount is the number of shape graphs. It's a big difference if there are fewer than ten shape graphs at each program point or if there are over a hundred. It's reasonable to assume that a user can focus on and understand and remember the differences in four different shape graphs. It's also reasonable to assume that hardly any human being will be able to focus long enough to understand and remember four hundred shape graphs, for each program point that is. It's also not feasible to use the same method as with predicates and not show certain shape graphs. Because on the one hand that would either leave important parts of the algorithm out or if those parts were not important, why are they there in the first place? On the other hand someone would have to go manually over hundreds or thousands of shape graphs and mark which ones to show and which ones to omit. Our visualizer has a mechanism to help with this though and it is the partition option. This allows the user to partition the shape graphs into equivalence classes according to some similarity concept, details about this are in chapter 3.2. However partition is a tool that is not easy to use and the best way to deal with the number of shape graphs is of course to keep them low to start with.

To understand how to keep the number of shape graphs low, we have to go back to how TVLA works. Potentially, the number of shape graphs grows exponentially with the number of predicates. It's obvious that the number of nodes in a shape graph is bounded by $2^{|A|}$, where A is the set of (unary) abstraction predicates and between those nodes there can be any number of binary predicates (set B), two shape graphs with the same nodes but different binary predicates are still two different shape graphs, that we often don't want to abstract together. A simple example for this would be a single summary node that represents the path of a search algorithm in a binary tree: It may be that the search algorithm only took left children on its way down, or only right or a mixture of both. Now for some algorithms the distinction won't matter, but for other algorithms it might very well matter. If it does matter then we just tripled the number of shape graphs for each such summary node. That means a potential blow up to $2^{2^{2|A||B|}}$ possible shape graphs for each program point. Of course that worst case scenario will never happen in practice, since the predicates are not independent from each other and certain combinations just won't make sense. It is however still quite easy to get exponential blow up from adding important predicates and some helping predicates. Therefore one major rule should always be observed: If it doesn't absolutely have to be there for the algorithm to make sense, then it shouldn't be there.

We shall illustrate the design principles for a user friendly analysis by having a closer look at the pairing heap analysis. Pairing heaps are a self-adjusting kind of heap that is easy to implement and - in practise - fast. They were originally designed as an easy to implement alternative for Fibonacci heaps, however unlike them they are much harder to analyse in theory, but are faster in most practical examples, see also Fredman, Sedgewick, Sleator, and Tarjan [10].

A node in a pairing heap can have an arbitrary number of children, only the heap condition must be met at any node. The usual heap operations are given as the following pseudo code:

```
function merge(heap1, heap2)
  if heap1 == Empty
    return heap2
  elsif heap2 == Empty
    return heap1
  elsif heap1.elem < heap2.elem
```

```
    return Heap(heap1.elem, heap2 :: heap1.subheaps)
  else
    return Heap(heap2.elem, heap1 :: heap2.subheaps)


function insert(elem, heap)
  return merge(Heap(elem, []), heap)


fuction delete-min(heap)
  if heap == Empty
    error
  elsif length(heap.subheaps) == 0
    return Empty
  elsif length(heap.subheaps) == 1
    return heap.subheaps[0]
  else
    return merge-pairs(heap.subheaps)


function merge-pairs(l)
  if length(l) == 0
    return Empty
  elsif length(l) == 1
    return l[0]
  else
    return merge(merge(l[0], l[1]), merge-pairs(l[2.. ]))


funtion decrease-key(heap, elem, value)
  if heap == Empty || elem == Null
    error
  elem->key = value
  remove elem from children of parent
  return merge(heap, Heap(elem, []))
```

The most interesting function is the `delete-min`, which uses `merge-pairs`. The way the delete min is executed gave the pairing heap its name: The minimum (root) node is deleted and then the subtrees are divided into pairs and merged from left to right. After that is completed all resulting trees are merged from right to left.

The first and one of the most important things when designing an analysis for an algorithm is to decide which concrete implementation will be best for abstract visualization. In our example a tree like data structure is needed with no limitation on how many children a node can have and that can shrink and grow as needed. That means every node must have a pointer to a collection of pointers that represent its children. It should be easy to iterate over the collection since we need to do that for the `delete-min` function. The obvious choices are either an array or a list. We decided in favor of the list because of two reasons:

- It is very easy to insert or delete a list element. If we used an array then we would have to implement special cases for adding elements to an already filled array (doubling the array size for example). This would not be a problem if we were demonstrating the algorithm in a lecture, since we would just say that doubling the array size will solve the problem and won't cause much extra runtime in an amortized sense. In our static program analysis we would have to explicitly code this special case, which would result in extra shape graphs, which in turn would only clutter the visualization with a detail that is of no importance to the analysis at all. Because at any time we add an element to the children array, we would have the abstract case that there is either no child, some children or that the children array was full. With the list we only have the first two cases and as we shall see later we can reduce it even further.
- A list is one of the most basic and most easy to abstract constructs for TVLA. It can easily be represented by a single summary node with just the structural next pointer and reachability predicates. An array is much harder for TVLA to handle. As a reminder: Shape analysis is interested mostly in the pointer structure of the heap. In a list the next element is explicitly pointed to by the predecessor, in an array this information is implicit and thus hard to model with a pointer structure. Logically you can model it with a predicate that acts like the next pointer in a list, but it is very hard to model the actual size of the array, i.e., if we want to check for an array index out of bounds error. The strength of an array over the list is of course the direct access of elements by their index, this is however never used in the example algorithm, thus there is no reason to use the more confusing data structure.

So a list it is. The next step is to decide what kind of list. For every function except `decrease-key` a single linked list is sufficient and remembering our major rule, it makes sense to break the analysis up into parts for each of the functions separately (except for `delete-min` and `merge-pairs`). That means we can use single linked lists for all functions but `decrease-key` where we need it to efficiently remove a child from the parents children list.

So now that it is decided which kind of data structure to use it is time to start to think about programming. Keep in mind that specifying program code in TVLA is more akin to writing assembler code than any higher programming language. Standard statements like if-then-else may take several lines of code and even goto like statements to translate. That means it is generally a good idea to keep the number of statements as low as possible, which especially means trying not to do unnecessary conditionals as those usually don't do anything that is easily visualized, because often only secondary predicates are changed and sometimes nothing is changed at all but only a filter is applied to the set of shape graphs. This means that it can happen that there are two or more program points where nothing visually changes, which can be confusing.

With the above in mind let's consider our example some more: Only two cases are important when iterating over a list: Is the list empty or is there another element to iterate over? But if our algorithm does not look at children in a certain program point, then we really don't care at all if there are any children or not. Thus came the idea of using a sentinel. We shall illustrate our point by an example. Let's consider Figure 32. This is our basic, totally abstracted non empty heap. As a reminder: Summary nodes always represent at least one node, that means at least one node must exist in the heap. Another design decision was that every element, even the root node, are pointed to by a handle. These handles will also be the list elements in the children lists and they are drawn as rectangles, whereas the real heap elements are drawn as circles. It may seem strange that the `root` pointer actually points to the handle of the heap root node and not to the node itself, this is however with good reason. Such as it is we never have to create a handle, we expect every node to have a handle, even one that we just now want to insert. This also means that when we do an insertion and the previous root has to be inserted into the children list of the new element, that we do not have to waste code to first create a handle for it. Another reason for the handles is a lot more pragmatic: If we pointed directly to the heap elements, then those would need either a parent pointer to their heap parent, or a pointer to their list handle or else we couldn't do a `remove-key` or `decrease-key` operation. So in short we saved us the trouble of modelling yet another structural pointer that isn't really needed.

The graph itself may also look strange at first since it doesn't look like a tree at all. This is a necessary evil of abstraction: Every tree node is abstracted into the same summary node and every handle is also abstracted in the other summary node.

In Figure 33 we expanded the graph, now there is the `min` pointer on the actual root node of the heap, making it a unique node and also the children list of root is now seperate from all other due to reachability. There are two pointers on the first handle of the children list and here we see the first advantage of the sentinel. Without the sentinel we would have two shape graphs, one depicting that this is the only child and the other that there are still more children. This information is at this program point of no relevance to the algorithm or the understanding of the data structure. We don't need it, so we don't want it. Note that this means we have fewer shape graphs to worry about until a time where they are actually needed to see something important to the algorithm.

The figures 34 and 35 show the next step, where the `it` pointer moved to the next list element. Now we need to know if there is another child or not and thus we get the two shape graphs. Thanks to the sentinel we neither have to check for, nor work with null pointers. This is especially true in the next step of the algorithm after this, which would remove the `help` handle from the children list and then reconnect the list.

Another easy to see example is the `toDo` list. We use this list for the `merge-pairs` function. After each pair of children is merged they will be put into the list and after all pairs have been merged the list will be processed to merge all the elements into a single heap. Without the sentinel we have three distinct cases that TVLA has to describe with different shape graphs:

1. The empty list. There have been no elements added yet and thus the list pointer is null.

2. The case the we have exactly one element added. This means the list pointer will point on this one element and make it unique.

3. More than one element was added: The list pointer will point to the head of the list, making it unique and the rest of the list is abstracted into a single summary node.

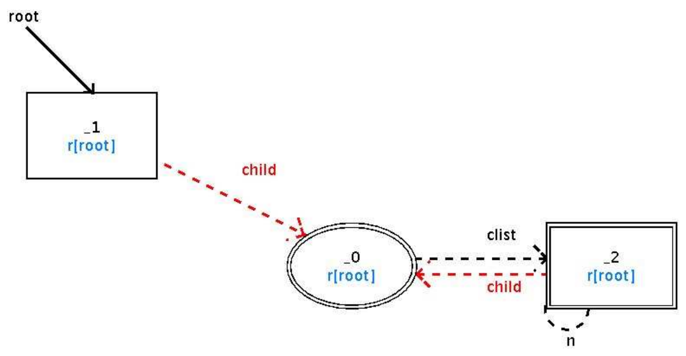


Figure 32: A shape graph depicting a maximally abstracted pairing heap. There is only the root pointer to the handle of the root element, which means the heap is non empty, but aside from that nothing is known. Square nodes represent handles to actual heap nodes, which are round.

With sentinel the empty list is represented by the list pointer pointing to the sentinel (shape graph looks like case 2 above) and all other cases fall under the second case, that there was at least one element added, making the shape graph look like case 3 above. So we saved another shape graph and lost no important information that way. That is, the information was not important, because all the algorithm cares about is whether or not there is another heap to be merged or not.

Aside from some logical advantages while actually performing and designing the analysis, which we shall not further elaborate here, the question may arise whether or not it is worth all this effort just to save one shape graph here and there or sometimes just postponing creating more shape graphs anyway. The answer to that is quite simply: Yes, it is worth the effort because shape graphs tend to blow up exponentially. For example: Having one instead of two shape graphs when expanding a child may not save much if you only ever expand one child at once, but if you expand two or three at once (this actually happens in the `delete-min` function) then the choices multiply: You have two choices for the first child, another two for the second etc. They are usually independent

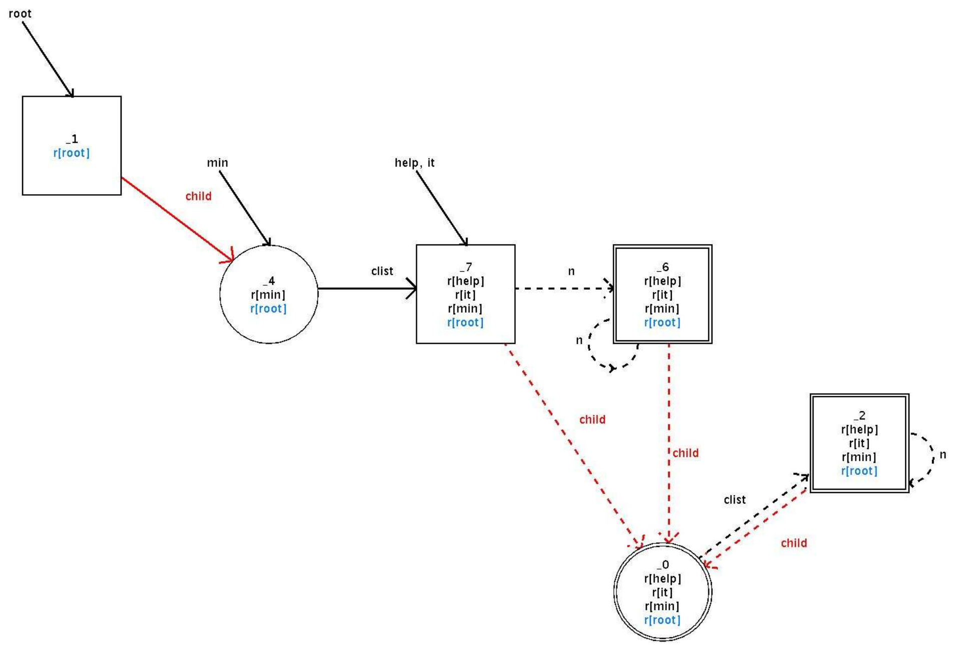


Figure 33: Here we are iterating over the children list of the root element of the heap. Note that at this point we do not know if there is another child after the `it` pointer and we don't have to know. Since there is always a sentinel though, we know that at least one node must follow.

of each other (in pairing heaps you cannot deduce anything about the number of children of a subtree by knowing the number of children of another subtree) and thus they multiply fast.

In summary, it is hard to formulate strict conditions and constraints for designing an analysis. But it is a good idea to first think about what is desirable to show, what is needed to do that and what really isn't needed and does nothing for the understanding. Keep the amount of code low as well as the number of shape graphs. The goal is not to be as precise as possible, but to be as precise as necessary.

3.2 Partitioning

In the previous section we have seen how careful planing of how exactly to implement data structures, coupled with well founded knowledge of TVLA (or the tool of choice for shape analysis) and the minimization of predicates used can in turn minimize the number of shape graphs produced. Fewer shape graphs means an easier learning and debugging experience. On the other hand, there are factors that stand against using the absolute minimum:

- While it may be most efficient (in terms of number of shape graphs) to implement a data structure in a certain way, that way may not fit at all with your style

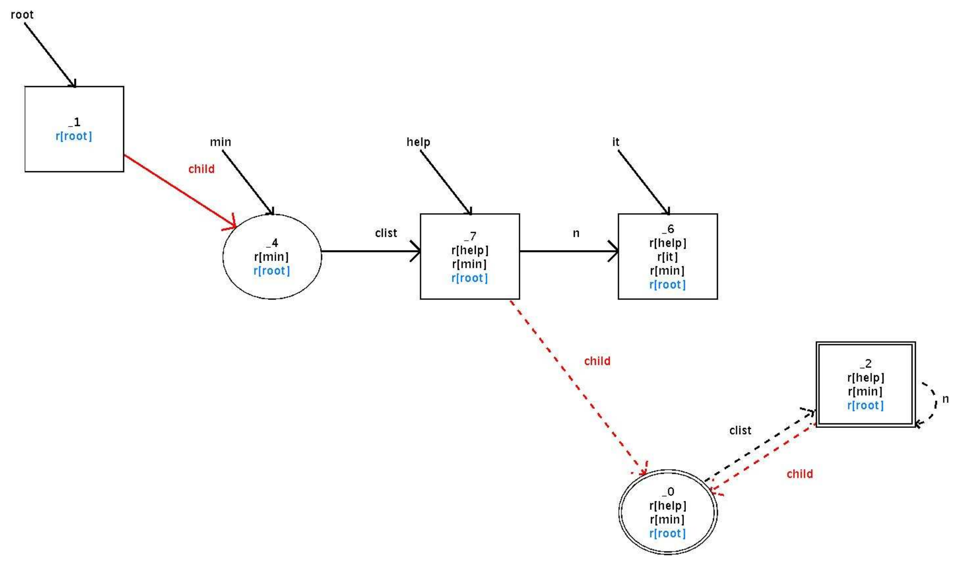


Figure 34: This is the next step in the algorithm after figure 33. The `it` pointer was moved to the next element in the children list and there were no more children only the sentinel.

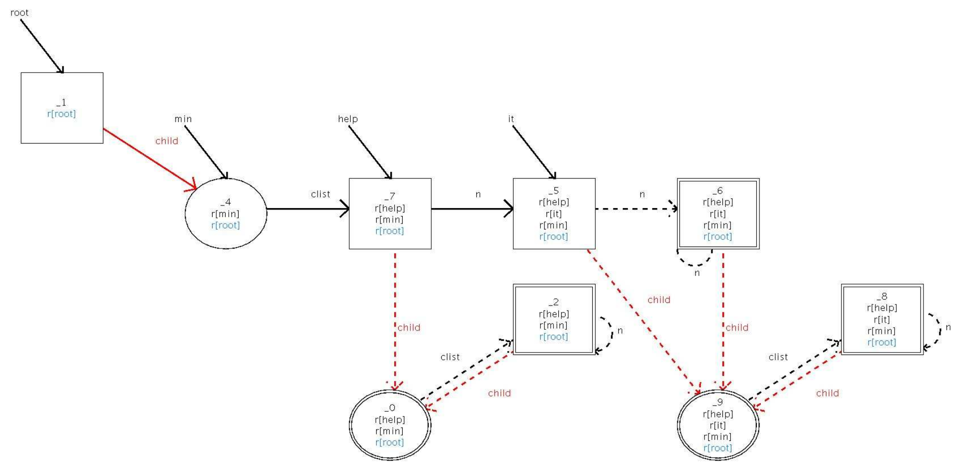


Figure 35: This is the alternative next step in the algorithm after figure 33. The `it` pointer was moved to the next element in the children list and there actually was at least one other child.

of teaching, or, worse, will yield suboptimal results in terms of running time or other constraints. Thus sometimes it might be necessary to fit the analysis to the teaching goals rather than to the visualization goals.

- While the rule of using as few predicates as possible may be sound advice, there are exceptions here too. The first factor to consider is always readability, too many predicates are bad for comprehension, but too few predicates can be equally bad. There has to be a balance between efficient visualization and ease of comprehension. The other point to consider is the level of detail wanted. By specifying the analysis, the user can, at least in the case of TVLA, also decide just how much detail should be shown. The more detail, the more predicates are often needed and the more different cases can be distinguished, leading to increasing (often exponentially) numbers of shape graphs. This can be as simple as deciding whether or not it is important to keep left and right subtrees separated at all times.

- Last but not least a very practical problem should also be kept in mind: Specifying analyses is not easy, specifying them optimally for a given task is most certainly very hard. Everyone who specifies a shape analysis only has a limited amount of time that they can devote to that task and the analysis will only be as good as it gets in that time.

All of the above are reasons why there may be more (sometimes a lot more) shape graphs in an analysis than there have to be. Additionally, even given only ideal analyses, there will be those that need shape graphs in numbers that are beyond what a human mind will be comfortable with. One solution that we propose here is based on what humans often did in such circumstances: We categorize the shape graphs into distinct classes, so we can argue over those classes instead of single shape graphs. The approach that I implemented in my Visualizer is based on the work of Dierk Johannes in his PhD-thesis (see Johannes [13]).

As was already mentioned the general approach is to partition a given set of shape graphs into equivalence classes. This is done by first reducing all original shape graphs to subgraphs, truth vectors or a combination of both and then checking which of the resulting reduced shape graphs are isomorphic. In order to preserve the structure of the analysis, as well as for easier understanding, it has to hold, that if two shape graphs are in the same equivalence class, then they belong to the same program point.

Johannes defines a number of what can be translated as *partitioners*, i.e., methods of reducing shape graphs to other entities. In my Visualizer I implemented two of those partitioners: Translated from German they are called *Formula Partitioner* and *Substructure Partitioner*.

The Formula Partitioner is simply a set of logical formulas, using the same syntax, semantic and predicates that is used in the shape analysis (3-valued first order predicate logic). These formulas are then evaluated for every shape graph. Two shape graphs are isomorphic if and only if those formulas evaluate to the same truth values in both of them. Alternatively, we can imagine a vector of truth values for each shape graph, like a canonical name of the shape graph, and shape graphs with the same canonical name belong to the same equivalence class.

The Substructure Partitioner is a little more involved. Here the shape graphs are reduced into a kind of substructure, that is again a shape graph: Nodes, predicates and edges may only be subsets of their original sets. The user has to specify a set of predicates P and a formula φ, both the set and the formula may be empty. The Partitioner will then remove all predicates from the shape graph that are not in P. If the predicate in question is a binary predicate then all the edges representing the predicate will vanish. If there are any nodes that now have zero non-`false` unary predicates, then they will be deleted, together with all the edges adjacent to them. After that the formula φ will be evaluated for every node (as usual the variable $v1$ will be reserved for this), if it yields `false` the node will be deleted, again with all adjacent edges. The resulting reduced shape graphs are then checked for isomorphism and are put into classes accordingly. Note that while graph isomorphism is usually hard to check, in this special case it is much easier because of the unique pointer variables, thus there is a clear matching for at least those nodes with stack pointers, making the check much easier.

Two shape graphs are isomorphic for this Partitioner, if and only if there exists a matching with the following conditions:

- The shape graphs have the same number of nodes.
- The nullary predicates of both shape graphs have the same truth values.
- Matched nodes have the same truth values in all their unary predicates.
- Edges between matched nodes have the same truth values.

Note, that this definition of isomorphism is not the same as the one used by TVLA to check if two shape graphs are isomorphic! For TVLA it is enough if one shape graph can be embedded into the other, along with a few more technical and implementation and version specific extra conditions.

The main purpose of the Substructure Partitioner is to focus on the structure of the shape graphs, grouping them according to common substructures, i.e., if the "unnecessary" predicates, meaning those that are not relevant to the working of the algorithm, are cut out of the graph, then the real basic cases appear. E.g., going down to the left or right child.

The Formula Partitioner on the other hand does not care so much about the graph structure and rather looks at the logical properties of the shape graphs, grouping them by similar behavior. I.e., the look of the graph is irrelevant, the important thing is whether or not these properties hold. It is easy to check for a leaf case with this for example: The structure of the graph may vary widely, but a leaf can always be easily defined by not having any children, which in turn is easily described by a logical formula.

Both kinds of partitioners have their uses, since some properties are easier described in logical form and others are easier to see in the structural form. There is, however, no reason to restrict ourselves to use just one of them. We allow to use both at the same time, partitioning first by one and then partitioning the resulting classes again by the other, thus having the best of both methods.

The following example illustrates the use of the Substructure Partitioner. Let us again consider the AVL-tree insertion algorithm. We are in the search phase still. I.e., the algorithm runs a cursor pointer down the tree, searching for the right spot to insert the new element. Figure 36 shows one shape graph that can happen in this search loop, which we will use to base the example on. In this example shape graph, we are looking at the second iteration of the search loop. Obviously the new, to be inserted, element has a data value that is smaller than the root and thus the cursor pointer cur now points to the left child of the root. We can ignore the prev pointer for this example, as it is only really needed for the backtracking after the actual insertion.

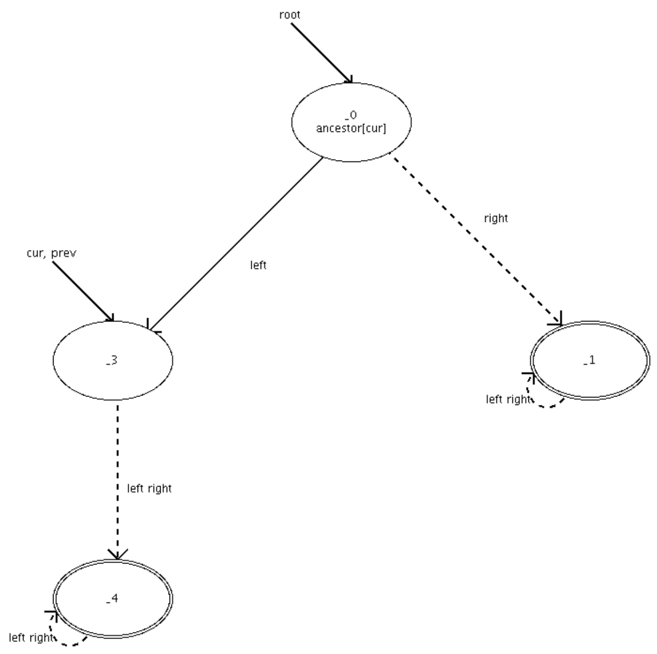


Figure 36: One example of a possible heap situation in the search loop of the AVL-insertion algorithm. This is the case where the cursor pointer cur is at the left child of the root and in the next program step the cur pointer will be either set to its left or right child depending on the data values of the current node and the node to be inserted.

The next step in the algorithm is to set the cur pointer to the left or right child of the current node, depending on how the data value comparison turns out. Thus it is easy to see that there should be at least two successor shape graphs, one for each general case. However due to how this particular analysis was designed, it is actually also possible, that there exists no left or right child and the algorithm has then to proceed to do the actual insertion. Taking this into account, we may then concede four possible cases and thus four shape graphs.

For those not so experienced with shape analysis and TVLA in particular it may come as a surprise, that the actual number of successor shape graphs is at least 12. Each of the four cases consisting of another three subcases at least. We show the possible three successors for Figure 36, for the case of going left and the left child actually existing in Figures 37 to 39.

The first thing of note here is, that the three distinctions are quite irrelevant to the working of the algorithm: At no point in the search phase will the algorithm check or care if there exist subtrees that are not traversed anyway. I.e., it is of no importance

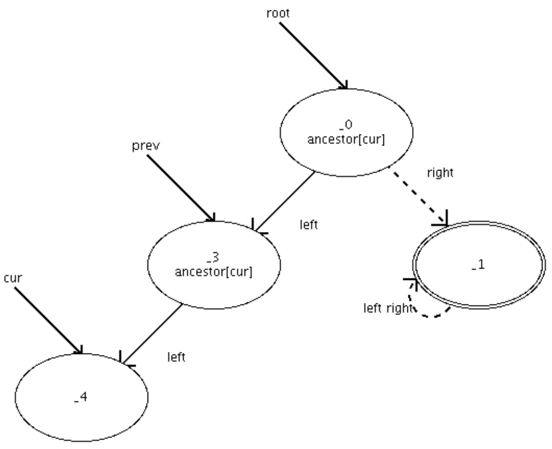


Figure 37: The simplest successor shape graph of Figure 36. Here the right subtree does not exists and the left subtree only consists of one node. Note that since the `cur` pointer now points to a leaf, in the next loop iteration the new element will be inserted.

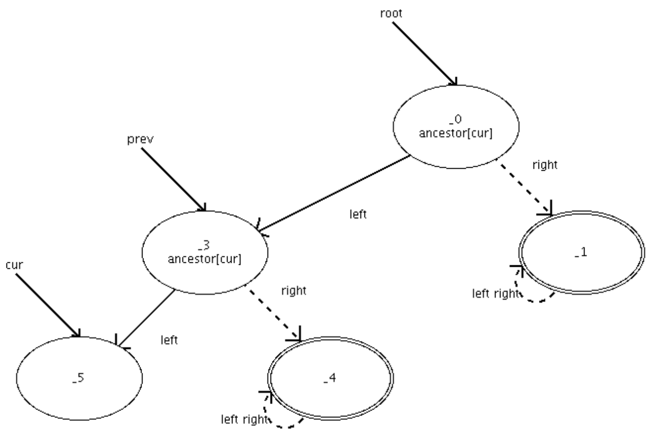


Figure 38: Another successor for the shape graph of Figure 36. Again, the left subtree only consists of a leaf, but additional to Figure 37 there exists also a right subtree of undetermined size. As with Figure 37, in the next iteration of the search loop the new element will be inserted.

for the understanding of the algorithm if the node with the `prev` pointer has a right subtree or not in the three example shape graphs. There are of course logical reasons why these distinctions are made and actually have to be made, since all possible heap states have to be represented by some shape graph at the program point. If we left out Figure 37 for example, then this would insinuate, that no AVL-tree could ever have a left leaf, without it being balanced by a right subtree (remember that summary nodes always stand for at least one node).

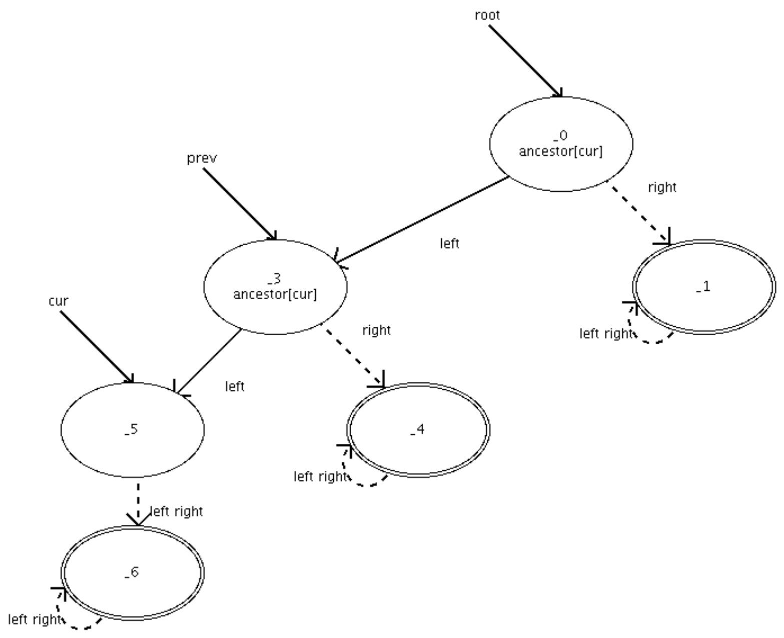


Figure 39: This is the most general successor of Figure 36. There exists both a left and right subtree and neither are leaves. This means in the next iteration of the loop there basically exist the same three cases again, aside from the slight changes to the insertion path. In particular it means that in the next step of the iteration there can be no insertion happening.

Since the three shape graphs in Figure 37 to 39 basically describe one and the same general case of the algorithm, it makes sense to put them into the same equivalence class. To do this, the Substructure Partitioner is the tool of choice. Remember, that the Substructure Partitioner is there to focus on the structure of the shape graph. What we are interested in is the shape of the search path taken so far, everything else is of no consequence (like subtrees hanging from the insertion path). We need two constraints to be fulfilled in order to put two shape graphs into the same class:

- The structure of the search path must be the same. Obviously we can only be in the same case if we took the same insertion path in both shape graphs.
- The pointers must point to the same nodes. If one or more pointers point to different nodes then we again are either not in the same case or not in the same iteration of the search loop. Remember that all shape graphs at the current program point are checked for partitioning, not only those with the same predecessors.

Thus we focus on the pointer predicates `root`, `cur` and `prev` as well as the `ancestor[cur]` predicate that represents the path from the root to the pointer `cur`. Of course we also need the `left` and `right` predicates to be able to compare the structure of the graphs. All other predicates are not relevant and can be deleted in the substructure. Finally, we want to make absolutely sure, that only nodes on the search path to `cur` remain in the substructure. To achieve this, the formula "`ancestor[cur](v1) | cur(v1)`" is added. The result of this Substructure Partitioner can be seen in Figure 40.

This figure shows the shape graph, that is the result of using the Substructure Partitioner on any of the three shape graphs from Figure 37 to 39. All the subtrees are cut off and only the basic path remains.

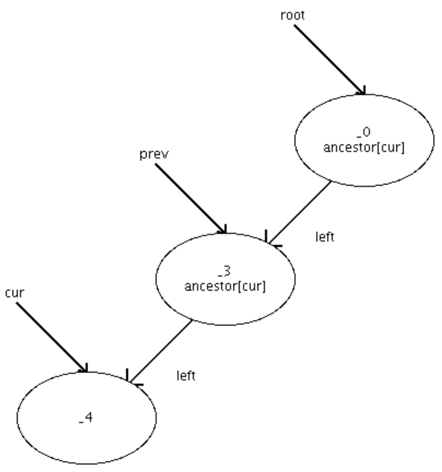


Figure 40: The reduced shape graph that remains after using the Substructure Partitioner with the specification described above on the three shape graphs shown in Figure 37 to 39.

The Visualizer has a number of features to help navigate the partitioned shape graphs:

- A label to see how many classes of shape graphs there are at the current program point.
- A label to see how many shape graphs are in the class that the currently viewed shape graph belongs to and what number the current shape graph has in that class.
- Buttons to navigate through the shape graphs within the same class, as well as jump to the next or previous class.
- The ability to show the reduced shape graph for the current class. I.e., what is shown in Figure 40

To continue with our example, remember that, while all three shape graphs are in the same case in the sense that the algorithm went to the left child, we might want to further fine tune the cases. The shape graphs in Figure 37 and 38 will result in an insertion in the next step, while the Figure in 39 are the most general case and can not go into the insertion mode without at least one more iteration. Thus assume we want to further subdivide the existing classes by distinguishing between leaf cases and general cases.

To do this we have two options: Either we further refine the Substructure Partitioner, risking that we change the existing classes in unintended ways, or we leave the existing classes because we are happy with them and use the Formula Partitioner to further subdivide them. We can achieve our goal by simply adding the following formula:

```
E(u1)A(u2) cur(u1) & !left(u1,u2) & !right(u1,u2)
```

This formula will evaluate to `true` for the shape graphs in Figure 37 and 38, but will evaluate to `unknown` for the shape graph in Figure 39 and thus will split the class just as we intended.

The same result could also be achieved by adding the `r[prev]` predicate to the Substructure Partitioner and changing the formula to "`ancestor[cur](v1) | r[cur](v1)`". For more information about partitioning see Johannes [13].

3.3 Concrete Versus Abstract

What is the best method to properly learn or teach? This is an important topic, that goes back to the dawn of mankind. It has been asked thousands of times and the answers have been equally as manifold. Many approaches have been tried and while some may statistically have been more successful than others, the one true way has yet to be found.

One of the most fundamental problems is the diversity of the human mind itself: What may work well for one must not necessarily be good for the next as well. One may try to partition the learners into two groups: Those that prefer to start with the formal, abstract view and who then try to adjust the abstraction, so that it fits the observed facts and on the other hand the ones those that do it the other way around and put more stock into intuition and try to deduce facts and invariants from examples and observations.

Note that this is not a real partition, as the two groups are not mutually exclusive. Usually there is a mixture of both for every person. Also both approaches share some similarities, as they both have to iterate between assumptions and verification of those assumptions.

What we saw so far from shape analysis, TVLA and the Visualizer was quite decidedly on the side of abstraction. The very output we visualize is abstract and in its entirety describes invariants. There is little in the form of concrete examples, or, more to the point, concrete program executions where we actually can see the whole data structure all the time in full detail (or at least as much detail as is needed to see what is going on). We will remedy this by introducing the concept of concrete shape graphs and then matching the concrete to the abstract shape graphs. This allows the user to seamlessly blend both views: Look at a concrete program execution and at the same time see the equivalent abstract shape graphs that subsume these concrete cases.

We shall further elaborate the point by looking at the following figures. Figure 41 shows a standard concrete representation of a binary tree. All nodes are concrete, no part of the tree is obscured or abstracted. This also means that all information is (or at least can be) definite, e.g., node v_1 is definitely bigger or smaller then v_2 depending

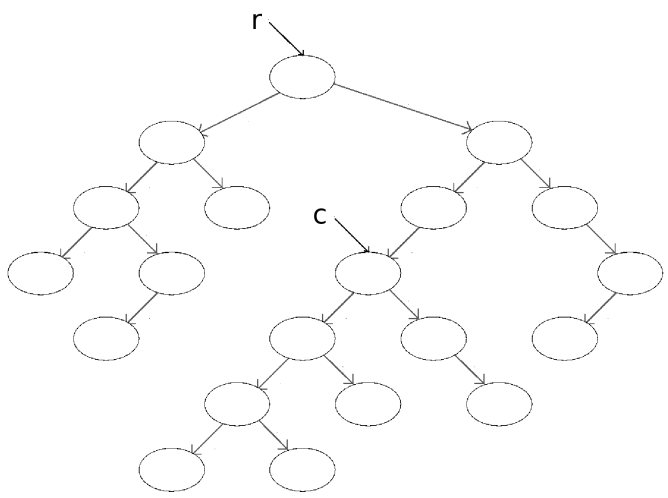


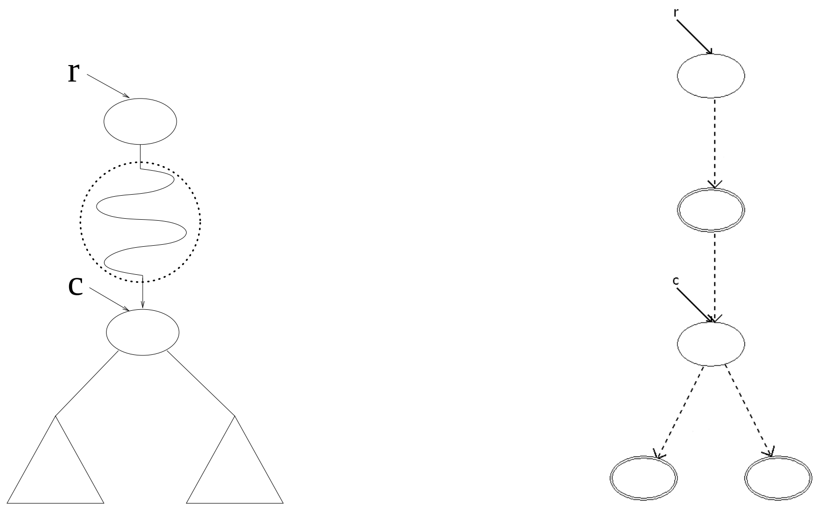
Figure 41: A concrete representation of a binary tree. All nodes are concrete, i.e., stand for exactly one node and all predicates are definite. Thus it looks just like any other binary tree example that you might drawn on a blackboard.

on their positions in the tree. In short Figure 41 depicts an example of what a teacher might paint on the blackboard to show an example of a binary tree.

Figure 42(a) again shows a binary tree, but this one is a lot more abstract than in Figure 41. When lecturing about insertion in binary trees then the teacher may paint a similar tree on the blackboard. The point of the abstraction here is of course, that the emphasis is on the current focus of the algorithm, i.e., the pointer `c` and whether or not to go to the left or right subtree. The choice is independent of the path from root to `c`, so that is abstracted. It is also independent from the subtrees, so they too are abstracted.

The Figures 41 and 42(a) show, that both tree representations have their use and their place in teaching. It is dependent on what the goal is, whether to choose one or the other. Figure 42(b) shows the same abstract tree as in Figure 42(a), only this time in TVLA or shape graph notation. This goes to show that abstract shape graphs are indeed canonical abstract representations of data structures and we can easily model any such abstract representations with TVLA. What we could not do so far is model trees like in Figure 41, the reason for this is the abstraction function, that would immediately abstract Figure 41 into a shape graph like in Figure 42(b).

The advantage of the concrete representation is, that every detail of the data structure is visible at all times and as such every change to the data structure is also clearly visible. The abstract shape graph does not have this luxury, if we go down one tree level in Figure 42(b) then we will come to the exact same shape graph again. This means instead of focusing on what is changing, abstraction is good to point out what is staying the same, i.e., to recognize invariants. Everyone in a teaching profession will surely agree that invariants are crucial in understanding algorithms. This is another reason why we want



(a) A much more abstract representation of a binary tree.

(b) The TVLA version of the tree in (a).

Figure 42: Equivalent abstract representations of binary trees. The one on the left side is something that may be painted on a blackboard, while the right on is an abstract shape graph from our Visualizer.

to have both views available.

This raises the question of how to provide concrete shape graphs, that actually look like Figure 41. Before answering this, we first must define what a concrete shape graph is. We already mentioned two properties that must hold:

- No summary nodes. Summary nodes are the very epitome of abstraction as they can stand for an arbitrary positive number of nodes, but in a concrete shape graph we want every node to only represent itself.
- All logical predicates are definite. Since we now only have concrete nodes, all relations between those nodes should either hold or not hold. Remember that the point of using 3-valued logic was solely to express uncertainty, but in a concrete graph all information should be readily available and thus definite.

This might seem sufficient, but there is one important thing still missing: There must not be any abstraction function. The obvious consequence of this is of course, that nodes will not be abstracted into summary nodes. The not so obvious consequence is, that there is no longer the usual invariant for any shape graph in place. With an abstraction function, there can only ever be at most one node for any given canonical name, without that function, we now can have arbitrarily many nodes that share the same canonical name. This in turn could have disastrous effects on the boundedness of the analysis, since we no longer have an upper bound on the size of concrete shape graphs. We will see however, that this is in fact a non-issue, as the analysis will still finish, as long as the algorithm terminates on the specified input graph.

Creating these concrete shape graphs is actually very easy and straightforward: All we have to do is give the analysis a concrete shape graph as input and then disable the blur operation of TVLA. Why is this logically sound?

- We use the exact same analysis specification to compute both the abstract and concrete shape graphs. Every concrete shape graph must be subsumed by at least one abstract shape graph. This follows directly from the fact that shape analysis computes invariants for every program point, i.e., if the concrete shape graph is a valid description of a possible heap state for that algorithm, then there must be an abstract shape graph that describes this case.
- The embedding theorem ensures us, that if an operation changes the logical properties of an abstract shape graph, then the resulting changes must also hold true for all its concretizations. In other words we can certainly use the same update formulas on a concrete shape graph and expect that the resulting concrete shape graph will again be subsumed by another abstract shape graph.
- The boundedness of the analysis is the biggest issue. We no longer have the assurance that the analysis will eventually come to an end, because the number of nodes and thus the maximum number of different shape graphs is no longer bounded. If we look a bit closer though, we see that this is actually not a big problem: The analysis can only diverge if there is a loop and that loop either produces a new shape graph every time that still passes the loop exit condition or if the shape graph at the end of the loop is identical to the shape graph at the beginning again, i.e., it was effectively not changed by the loop. This means that the analysis will diverge if and only if the actual algorithm would diverge too on that input. That should of course never happen in a properly programmed algorithm.

Since the goal was to provide a dualistic view of both concrete and abstract, we still have to find a way to match concrete shape graphs to abstract ones. As already mentioned, for every valid concrete shape graph (valid here means, that it is actually a possible heap situation for that algorithm), there must exist at least one abstract shape graph that subsumes it. This matching, however, is not necessarily well defined.

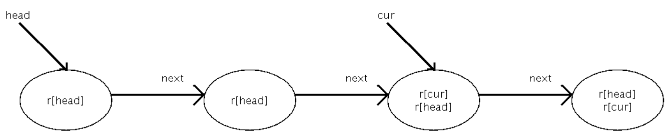


Figure 43: A concrete representation of a singly linked list.

Consider the example shown in Figure 43. It is a simple concrete shape graph that shows a singly linked list with exactly four concrete nodes. There is a `head` pointer to the first element of the list and an iteration pointer called `cur` as usual.

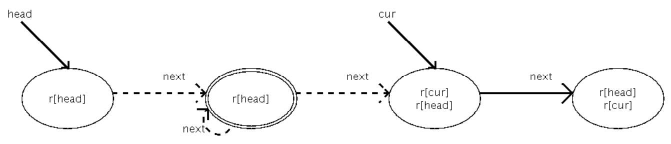


Figure 44: An abstract representation of a singly linked list, that subsumes the concrete shape graph in Figure 43.

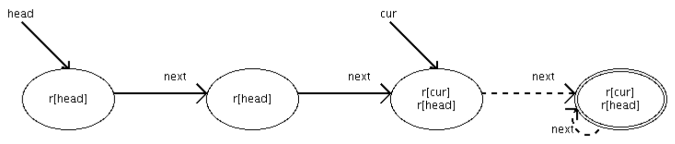


Figure 45: Another abstract representation of a singly linked list, that also subsumes the concrete shape graph in Figure 43. This time the end of the list is represented by a summary node.

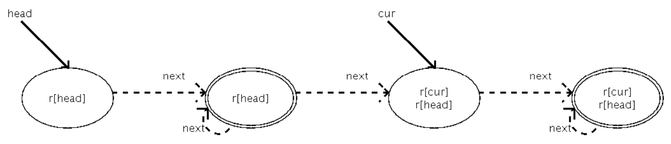


Figure 46: This abstract shape graphs not only subsumes the concrete shape graph in Figure 43, it also subsumes both of the abstract shape graphs in Figures 44 and 45.

Now if we consider Figure 44, it should be obvious, that this abstract shape graph subsumes the concrete one. The only difference is in the second node, which is a summary node. This abstract shape graphs represents all those concrete cases where there are arbitrary many, but at least one, nodes between the `head` and `cur` pointers.

Looking at Figure 45, we can see that this one too does subsume the concrete shape graph. The abstract shape graph represents all those cases, where `cur` is indeed on exactly the third element of the list and there are arbitrarily many, but at least one, node left in the rest of the list.

So now we have the seemingly strange case, that we have two abstract shape graphs that can represent the same concrete case. Indeed, Figure 43 depicts the only concrete case that is represented by both abstract shape graphs. This is, however, not as strange as it might seem. Remember that the precision of the shape analysis can vary, depending on what we want to show and it is thus not unusual, that there are cases where multiple abstract shape graphs represent the same concrete cases. In fact, if we look at Figure 46, then we see another abstract shape graph, that not only subsumes the concrete one from Figure 43, but also both of the abstract ones from the Figures 44 and 45.

The reason why this can in fact happen in a shape analysis is precision. The fewer summary nodes in a shape graph, the higher the definite information values usually.

Thus it might make sense to have to same case multiple times, because the subcases may be more precise than the general case. It is however highly dependent on the program, the specification and even the options given the TVLA engine whether or not certain cases can even occur. Generally speaking though, we do not know what will be there.

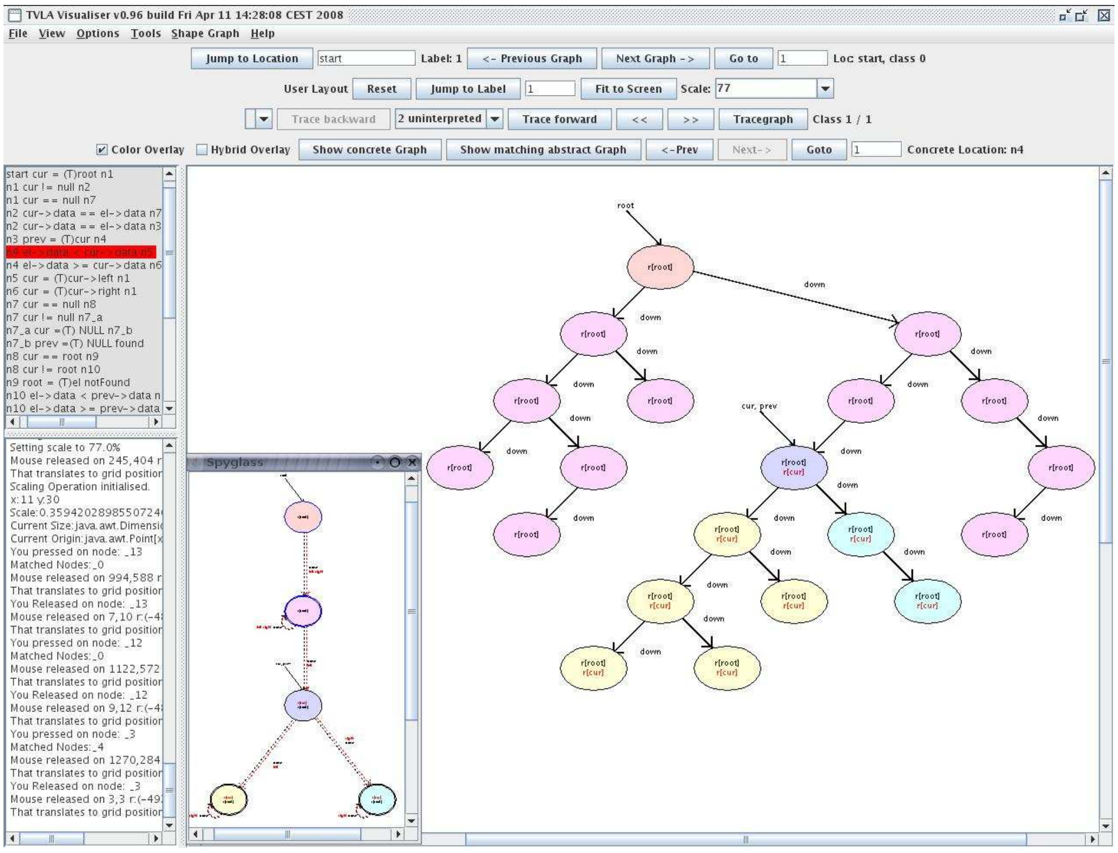
This leaves us with a slight decision problem: The above figures already show, that there is no well defined abstract shape graph that subsumes a given concrete one, there may be arbitrarily (but finitely) many. Furthermore Figures 44 and 45 show, that the least general abstract shape graph that subsumes the concrete one is not well defined. While in theory, there is always a well defined most abstract shape graph, there is no guarantee, that this theoretical shape graph is actually part of the current shape analysis. So consider for a moment, that the abstract shape graph in Figure 46 would not occur in the analysis and we only had the two in Figures 44 and 45. In that case the most general shape graph is also not well defined.

As there is no way to automatically generate a metric for which shape graph makes more sense, we just pick one randomly in case of a draw. There still remains the question however, whether we try to find the most general abstract shape graph or the least general one. There are arguments for both sides: E.g., the least general one being sharper than the most general one. In the end we decided to go for the most general one. The reasoning is, that we want to provide a dual view of concrete and abstract, that means we already have a perfectly definite concrete shape graph with full information value, thus it makes sense to try and match it with the most general case, to show the most abstract one in contrast.

So how do we actually compute whether or not a concrete shape graph can be *matched* to a given abstract shape graph? To do this we basically emulate what the blur operation in TVLA does: We start by matching nodes from the concrete shape graph to the abstract one. While this is hard to do in general graphs it is actually much easier with shape graphs. All we have to do is to compute the canonical name[7] of each node and check whether or not there exists a node with the same canonical name in the abstract shape graph. After a successful node matching, we still have to make sure that the edges also match, i.e., if there is a concrete edge between two nodes in the concrete graph, then there must at least exist an indefinite edge between the corresponding nodes in the abstract shape graph (or a self loop if both concrete node are matched to the same abstract node). Lastly we also have to check the nullary predicates. Just to emphasize once more: We do not look for a perfect matching, we only make sure that the concrete shape graph is actually a concretization of the abstract one.

Figure 47 shows a screen shot of our Visualizer. In the big window we can see a concrete shape graph, that is matched to the abstract shape graph in the smaller window. Both graphs are color coded, to more easily see which nodes are matched to which. We can at any time switch between concrete and matching abstract shape graph and can either navigate through the concrete shape graphs to follow the program execution, or

[7]Reminder: The canonical name of a node is the vector of truth values of the abstraction predicates, that are clearly labeled as such in the specification.



```
start cur = (T)root n1
n1 cur != null n2
n1 cur == null n7
n2 cur->data == el->data n7
n2 cur->data == el->data n3
n3 prev = (T)cur n4
n4 el->data < cur->data n5
n4 el->data >= cur->data n6
n5 cur = (T)cur->left n1
n6 cur = (T)cur->right n1
n7 cur == null n8
n7 cur != null n7_a
n7_a cur =(T) NULL n7_b
n7_b prev =(T) NULL found
n8 cur == root n9
n8 cur != root n10
n9 root = (T)el notFound
n10 el->data < prev->data n
n10 el->data >= prev->data
```

```
Setting scale to 77.0%
Mouse released on 245,404 r
That translates to grid positior
Scaling Operation initialised.
x:11 y:30
Scale:0.359420289855072
Current Size:java.awt.Dimensi
Current Origin:java.awt.Point[x
You pressed on node: _13
Matched Nodes:_0
Mouse released on 994,588 r
That translates to grid positior
You Released on node: _13
Mouse released on 7,10 r:(-4
That translates to grid positior
You pressed on node: _12
Matched Nodes:_0
Mouse released on 1122,572
That translates to grid positior
You Released on node: _12
Mouse released on 9,12 r:(-4
That translates to grid positior
You pressed on node: _3
Matched Nodes:_4
Mouse released on 1270,284
That translates to grid positior
You Released on node: _3
Mouse released on 3,3 r:(-49
That translates to grid positior
```

Figure 47: This is a screen shot from our Visualizer, that shows a concrete shape graph in the main window, being matched to an abstract shape graph in the smaller window. Matching nodes are color coded for easier recognition.

go through the abstract shape graphs, to see if they have matching concrete cases.

The highlight of this feature is the built in interactivity provided by the shape graph creation tool. We already described the functionality of this tool in Section 2.4 and as we hinted there, it is very useful for providing a bit of interactive content. Other tree insertion visualizations often allow the user to choose the next element to be inserted and then watch how the algorithm works on that input. We allow the same! Using the "modify current shape graph" option allows the user to make modifications to the currently viewed concrete shape graph and then click on the "start analysis" button to start the analysis with this shape graph as input. The output is automatically imported and matched to the abstract shape graphs. This allows to start with an empty tree for example, then insert a single element. We can then go to the last shape graph (where the new tree is described) and add a new element to be inserted and restart the analysis. This can be repeated and allows to build your own tree in a similar fashion to other visualizations.

In closing I can summarize, that the possibility to view both concrete and abstract shape graphs at the same time is a great help for understanding the program being analyzed, but also is a good way to get some intuition of how abstraction works. In addition it also allows some interactivity by modifying the concrete shape graph and then see what happens if the modified shape graph is used as input for the next program execution.

3.4 Pseudo Code

When thinking about the subject of demonstrating or explaining algorithms, it is hard to imagine doing it without the use of some form of pseudo code. There are a lot of different program languages with different syntax, yet quite many are close enough in notation, that certain constructs are well known and easily recognized. Pseudo code also helps with those hopelessly complicated statements, that can be much easier put into a single descriptive sentence, rather then several cryptic lines of code. In short, pseudo code is an invaluable help for teachers and students alike.

So, what has pseudo code got to do with shape analysis? The answer is simple: The analyzed program has to be specified somehow. The way this is done is by TVLA actions, as described in Section 1.3. Out of necessity, TVLA action names are often a bit technical in nature and thus not like standard pseudo code. Thus it is not very intuitive or easy to read and only really good if we want to debug an analysis. The reason for this becomes more apparent if we remind ourselves, that basically for every slight difference in a data structure we need a new action. So as a simple example, if we consider lists and write code like `cur=cur->next` then it is obvious for anyone and even for the compiler, that we have a pointer here, that gets assigned a new value and that this value is stored in the `next` component of the object or structure it is pointing to now. A human will conclude at once, that obviously, here a pointer is set to the next element in the list.

The real interesting thing about that example is, that it does not even matter if the list is singly linked, double linked, a skip list or whatever else. A human mind will understand it in the right way anyway and even the compiler can usually do it without further help (unless there is a type conflict). TVLA, however, can not. Materializing a node out of a singly linked list or a doubly linked list is not the same thing logically. There may be different predicates involved and certainly the focus formula will change. If there are more predicates in one of the data structures than the other then the update formulas likely will also change.

The result is, that we need a TVLA action for every data structure and every program statement, even if they seemingly do the exact same thing. The avid reader may now remember that TVLA actions themselves already had a tag that allowed the creator to supply a pseudo code string. So why not just use those strings? This is best explained by showing an example. The following control flow specification was taken directly from the example folder of TVLA, it is an insert algorithm for a standard binary tree.

```
// cur = root;
start Copy_Var_T(cur, root) n1
```

```
// while (cur != NULL && cur->data != el->data) {
   n1 Is_Not_Null_Var(cur) n2
   n1 Is_Null_Var(cur) n7
   // For now not interpreting non-equality (while not using DataIsNequal).
   n2 Equal_Data_T(cur, el) n7
   n2 Not_Equal_Data_T(cur, el) n3
   // prev = cur;
   n3 Copy_Var_T(prev, cur) n4

   // if (el->data < cur->data)
      n4 Less_Data_T(el, cur) n5
      n4 Greater_Equal_Data_T(el, cur) n6

      // cur = cur->left;
      n5 Get_Sel_T(cur, cur, left) n1

   // else cur = cur->right;
      n6 Get_Sel_T(cur, cur, right) n1
// }
// Don't insert duplicates
// if (cur == NULL) {
   n7 Is_Null_Var(cur) n8
   n7 Is_Not_Null_Var(cur) n7_a
   // Null out cur and prev, as an optimizer could.
   n7_a Set_Null_T(cur) n7_b
   n7_b Set_Null_T(prev) found

   // if (cur == root)
   n8 Is_Eq_Var(cur, root) n9
   n8 Is_Not_Eq_Var(cur, root) n10
   // root = el;
   n9 Copy_Var_T(root, el) notFound

   // if (el->data < prev->data)
      n10 Less_Data_T(el, prev) n11
      n10 Greater_Equal_Data_T(el, prev) n13

   // prev->left = el;
      n11 Set_Sel_Null_T(prev, left) n12
      n12 Set_Sel_T(prev, left, el) n12_a
      // Null out el and prev, as an optimizer could.
      n12_a Set_Null_T(el) n12_b
      n12_b Set_Null_T(prev) notFound
```

```
    // else prev->right = el;
       n13 Set_Sel_Null_T(prev, right) n14
       n14 Set_Sel_T(prev, right, el) n14_a
       // Null out el and prev, as an optimizer could.
       n14_a Set_Null_T(el) n14_b
       n14_b Set_Null_T(prev) notFound
// }

found uninterpreted() test
notFound uninterpreted() test

// Now test the structures
test Is_Sorted_Data_T(root) exit
test Is_Not_Sorted_Data_T(root) error
```

The reasons why the build in tag of the TVLA actions is inadequate are threefold:

1. Some standard statements cannot be formulated if we just translate a single action to a single pseudo code line. A simple example of this is the program point `n1`, that consists of two actions actually. Both together form the header of a `while`-loop, but individually they would be translated into two lines of pseudo code:
   `cur->data != 0`
   `cur->data == 0`
   It is not at all obvious, that these form the header of a `while`-loop, or maybe a conditional for an `if-then-else` statement. So there has to be either an automatic detection of loops and headers or some form of user editing that post-processes the pseudo code. It should be noted here, that automatic loop detection is in fact possible, but difficult, since the TVLA control flow specification is basically an extended `goto`-program. This does not only mean that finding the right headers in densely nestled loops can be difficult, but that there may actually exist loops, that are not well formed! Well formed here means, that loops are either completely disjunct or completely nestled in the others. With `goto`-statements it is however possible to build loops, that share parts of their body with other loops, thus making it impossible to directly translate them into `while` programs without an extensive detangling of the loops.

2. Some actions (mostly those that concern data values) may not actually do something. An example of this can be seen in the program point `n2`. The data values are not explicitly modeled by TVLA, all we have is a relation that tells us whether two data values are less or equal to each other. Thus program point `n2` does not actually do anything to the data structure and serves just as an abstract `if-then-else` statement, that models the case that the to-be-inserted element is already in the tree. An even more extreme case of this can be seen near the end at the `found` and `notFound` program points, where two different branches of program execution

are spliced together again by an empty action.

3. The last point is a bit similar to number 1. Sometimes logic needs some extra steps to properly formulate actions, that would normally be expressed by just a single line of code. An example of this are the program points `n11` to `n12_b` and `n13` to `n14_b`. Both of those blocks are actually defined by a single line of pseudo code, that is written as a comment above the respective block. Obviously the creator of the analysis also knew, that normally, you would only write the assignment and not further think about what happens on compiler side. When specifying TVLA analyses however, sometimes it might be necessary to take things one step at a time to provide maximum precision.

That means when all is said and done the strings to describe the actions in TVLA are nice to give the reader a short description of what the single action will simulate, but they are not enough to provide sensible pseudo code. Thus we have implemented our own pseudo code, that can be written as comments in the original tvp file, as shown below:

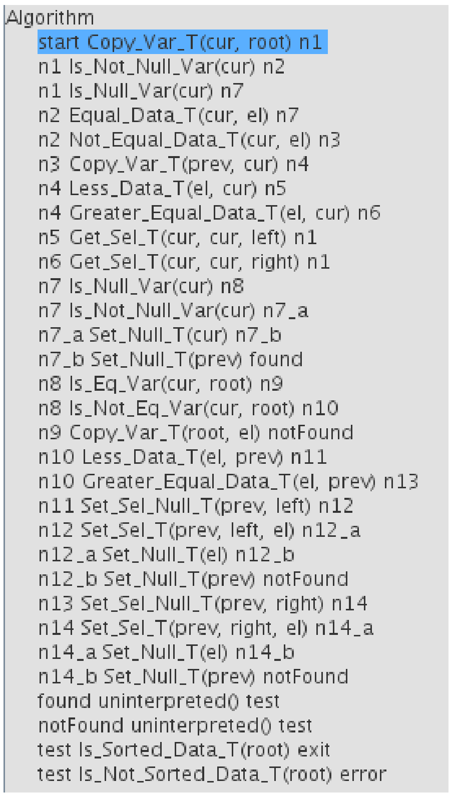
```
//START_PSEUDO
//cur = root #start#
//while (cur != NULL && cur->data != el->data) { #n1,n2#
//c: main loop of the algorithm
//  prev = cur; #n3#
//c: Move the prev pointer to the cur pointers position.
//  if (el->data < cur->data) { #n4#
//    cur = cur->left; #n5#
//  }
//  else { #n4#
//    cur = cur->right; #n6#
//  }
//}
//if (cur == NULL) { #n7#
//  if (cur == root) { #n8#
//    root = el; #n9#
//  }
//  else { #n8#
//    if (el->data < prev->data) { #n10#
//      prev->left = el; #n11,n12,n12_a,n12_b,notFound#
//    }
//    else { #n10#
//      prev->right = el; #n13,n14,n14_a,n14_b,notFound#
//    }
//  }
//}
//else {} #n7_a,n7_b,found#
```

```
//c: The element was already present in the tree and we don’t want to
     enter duplicate data values.
//*EXIT* #test,exit#
//c: This is not a real program location. This is just a pseudo
     location to see the final program state if everything went well
//*ERROR* #error#
//c: This is a pseudo location to see if there was unexpected trouble.
     This location should hopefully be empty.
//END_PSEUDO
```

The features of our pseudo code are:

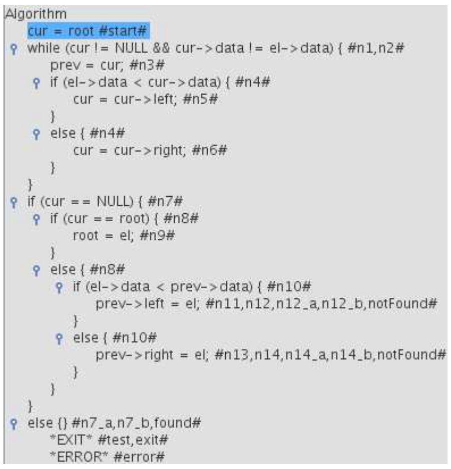
- The ability to assign arbitrarily many program points to a line of pseudo code (the labels between the #). This addresses the issues from points 1 and 3. In the Visualizer, clicking on one of these pseudo code lines will always show shape graphs from the first program location in the respective list and if going forward will skip over the other program locations where nothing (interesting) is happening anyway.
- The ability to insert comments, that will be shown as tool tips in the Visualizer if the user mouses over the pseudo code line. A comment always starts with the `c:` keyword and will be matched to the statement directly above it.
- `While`-loops and `if-then-else` statements are considered to be blocks framed by parentheses, as is usual in most programming languages. In the Visualizer, these blocks can be folded, to blend code out that is no longer, or not yet, interesting and thus increases readability.
- Our Visualizer allows complete freedom over the style of the users pseudo code. Instead of having to make do with what an automatic generator would produce, now the user can fit the listing exactly to their liking, thus giving them the ability to fit the presentation to their teaching style and not vice versa.

To give an impression of the difference between the TVLA code and our pseudo code we present the Figures 48(a) and 48(b). The pictures are screen shots from our Visualizer and should not require any more explanations of why pseudo code is helpful to have for the visualization of algorithms.



```
Algorithm
    start Copy_Var_T(cur, root) n1
    n1 Is_Not_Null_Var(cur) n2
    n1 Is_Null_Var(cur) n7
    n2 Equal_Data_T(cur, el) n7
    n2 Not_Equal_Data_T(cur, el) n3
    n3 Copy_Var_T(prev, cur) n4
    n4 Less_Data_T(el, cur) n5
    n4 Greater_Equal_Data_T(el, cur) n6
    n5 Get_Sel_T(cur, cur, left) n1
    n6 Get_Sel_T(cur, cur, right) n1
    n7 Is_Null_Var(cur) n8
    n7 Is_Not_Null_Var(cur) n7_a
    n7_a Set_Null_T(cur) n7_b
    n7_b Set_Null_T(prev) found
    n8 Is_Eq_Var(cur, root) n9
    n8 Is_Not_Eq_Var(cur, root) n10
    n9 Copy_Var_T(root, el) notFound
    n10 Less_Data_T(el, prev) n11
    n10 Greater_Equal_Data_T(el, prev) n13
    n11 Set_Sel_Null_T(prev, left) n12
    n12 Set_Sel_T(prev, left, el) n12_a
    n12_a Set_Null_T(el) n12_b
    n12_b Set_Null_T(prev) notFound
    n13 Set_Sel_Null_T(prev, right) n14
    n14 Set_Sel_T(prev, right, el) n14_a
    n14_a Set_Null_T(el) n14_b
    n14_b Set_Null_T(prev) notFound
    found uninterpreted() test
    notFound uninterpreted() test
    test Is_Sorted_Data_T(root) exit
    test Is_Not_Sorted_Data_T(root) error
```

(a) The normal listing, using just the tvla action names.



```
Algorithm
    cur = root #start#
    while (cur != NULL && cur->data != el->data) { #n1,n2#
        prev = cur; #n3#
        if (el->data < cur->data) { #n4#
            cur = cur->left; #n5#
        }
        else { #n4#
            cur = cur->right; #n6#
        }
    }
    if (cur == NULL) { #n7#
        if (cur == root) { #n8#
            root = el; #n9#
        }
        else { #n8#
            if (el->data < prev->data) { #n10#
                prev->left = el; #n11,n12,n12_a,n12_b,notFound#
            }
            else { #n10#
                prev->right = el; #n13,n14,n14_a,n14_b,notFound#
            }
        }
    }
    else {} #n7_a,n7_b,found#
        *EXIT* #test,exit#
        *ERROR* #error#
```

(b) Our pseudo code.

Figure 48: A direct comparison of how pseudo code improves the readability of the program text.

4 Pairing Heap Analysis

This Section will give an in-depth view of the TVLA specification for pairing heaps, or, to be more precise, the specification for the insert, delete-min and decrease-key operations on pairing heaps. The analysis was done with the express purpose of testing the theories presented in Section 3.1, while also creating something not done before with TVLA. The reason for choosing pairing heaps were threefold:

- The two data structures used in the analysis (lists and trees) were already quite familiar due to the many examples for both, that are included with the TVLA installation. Thus the hope was that there would be less time needed to combine the different specifications into one expanded whole.

- While there are quite a few examples of simple list and tree algorithms, so far there was no heap like data structure specified in TVLA. Maybe even more importantly, to the best of our knowledge, there exists no other analysis specification that included two different data structures at once! Usually it is either a list example or a tree example. In our case we have the usual tree structure for the proper heap, but we also use a list structure to manage the child lists.

- Pairing heaps are easy to implement. This also means there is not much code needed and not many special cases that would bloat the number of shape graphs.

I already described pairing heaps and the pseudo code for the three operations `insert`, `delete-min` and `decrease-key` in Section 3.1.3, as well as the reasons why I chose to represent the data structure in the way I did. In this section I will only look at how the concepts were represented in TVLA and not why they were defined like that in the first place. So without any further ado, we assume the following invariants for our data structure:

```
struct list_node {
   list_node *next;
   list_node *prev;
   tree_node *child;
};

struct tree_node {
   list_node *clist;
   int data_value;
};
```

- Our heap data structures consist solely of objects of type `list_node` and `tree_node`. The actual type of the data value in the tree does not matter as long as there exists an order relation on the data type. The type `int` has been chosen exemplary for convenience and has no effects on the working of the analysis at all.

- Every `tree_node` object has always a handle in the form of a `list_node`. In other words, at any time there has to exists exactly one `list_node` object for every `tree_node` object, that has a reference to said `tree_node` as its `child` pointer. This is again mostly for convenience, since it allows us to skip a few conditionals in the actual TVLA code.
- Stack pointers into the heap, that are meant as an object reference to a tree element will always point to the handle (i.e., the `list_node`), instead of the corresponding `tree_node`. This means that for any heap operations, that require a reference to a heap object, a handle reference will be supplied.
- All our children lists will always contain a sentinel at the end. The sentinel never points to any tree node.
- If we start an operation with a given heap, then we assume, that it is a valid pairing heap. We also assume, that object and data types are matching. The analysis will not type check at all and just assume, that all data comparisons will work normally.

I will explain the specification with the help of the `decrease-key` operation. It may not be the most interesting from the viewpoint of the algorithm, but it is the most demanding from the viewpoint of the specification, because here, we actually have to go forward and backward in a child list in the middle of a heap.

4.1 Predicates

Since we are using the already known lists and trees, some of the predicates will be familiar, both in name and in function. But due to the novel nature of this specification, many of them will have changed definitions and of course some are completely new. I will start with the first part of the tvp-file, where all the predicates are defined:

```
/*********************************************/
/*************** Core Predicates *************/
/********************************************/
%p dle(v1, v2) transitive {}

%p linked(v1,v2)

// the predicate clist points to child list of a heap node
%p clist(v1,v2) function invfunction

// the pointer from a list element to a tree node
%p child(v1, v2) function invfunction

//does the subtree with v1 as root satisfy the heap property?
%p isHeap(v1)
```

```
%p tree(v1) abs
%p list(v1) abs

// For every program variable z there is a unary predicate that
// holds for list elements pointed by z.
// The unique property is used to convey the fact that the predicate
// can hold for at most one individual.
// The pointer property is a visualization hint for graphical renderers.
foreach (z in PVar) {
  %p z(v1) unique abs pointer
}

// The predicate n represents the next field of the list data type.
%p n(v1, v2) function acyclic

// The predicate p represents the prev field (previous pointer) of
// the list data type
%p p(v1, v2) function acyclic

/*************************************************/
/**** Instrumentation (i.e., derived) predicates ****/
/*************************************************/
%i reachable(v1, v2) = linked*(v1,v2) transitive reflexive antisymmetric

// The is[n] predicate holds for list elements pointed by two different
// list elements.
%i is[n](v) = E(v_1, v_2) (v_1 != v_2 & n(v_1, v) & n(v_2, v))
%i is[p](v) = E(v_1, v_2) (v_1 != v_2 & p(v_1, v) & p(v_2, v))

// The t[n] predicate records transitive reflexive reachability between
// list elements along the n field.
// removed reflexive so we can use more than one type of data structure
%i t[n](v1, v2) = (n*(v1, v2) & !tree(v1))
            | (E(v3) clist(v1,v3) & n*(v3,v2)) transitive antisymmetric
%i t[p](v1, v2) = p*(v1, v2) & !tree(v1) transitive antisymmetric

foreach (x in PVar) {
  %i br[x](v) = E(v1) x(v1) & t[p](v1,v) abs
  %i r[x](v) = E(v1) (x(v1) & reachable(v1, v)) abs
  %i r[n, x](v) = (E(v1) (x(v1) & (clist(v1,v) | t[n](v1,v))))
                | (E(v1,v2) x(v1) & clist(v1,v2) & t[n](v2,v)) abs
}
```

The first predicate `dle` is already a familiar one. As a reminder: It is short for data less or equal and used to model sortation of data values. This is unchanged and since it cannot be derived from anything else, it is also a core predicate.

Predicate `linked` is new and with good reason: If we look at our data structure then we see a plethora of different pointer variables that can "link" two objects together. We have the `next`, `clist` and `child` pointers that all will make two object linked together. The first question is, why not define the predicate as instrumentation predicate, since it obviously is `true` if any of the three aforementioned predicate holds between two objects? The answer to that is because of precision. The `linked` predicate is very important, because we will define reachability with it and we will need it to be as sharp as possible, thus we have to focus on it and sharpen it in actions, as we shall see later. The second question is why did I not mention the `prev` pointer? After all it also links two objects together. The reason for this is a bit more intricate, but basically I want reachability to be one-way only. The backwards reachability is restricted to lists only, since there is no need in our specification to ever go up a heap tree. Thus the backwards reachability has separate predicates, making the constraints a lot easier to formulate.

The `clist` and `child` predicates are pretty straightforward as they only model the respective pointers in the data structures. As usual for pointers, they are defined as `function` and `invfunction`, indicating, that every pointer can only have one target and every object should only be pointed to be one pointer.

The predicate `isHeap` is the equivalent of `inOrder` in binary trees. It means that the heap invariant is obeyed in the respective subtree. This is important in order to properly set and sharpen the `dle` predicate when materializing nodes. If a new node is materialized out of some summary node, it may well be that the `dle` value is initially unknown, but if all `isHeap` predicates are set in the predecessors, then we can sharpen the `dle` to `true`.

The predicates `tree` and `list` are new. Their only reason for existence is the fact that we have two different data structures in the same analysis and that we need to keep track of which nodes belong to which data structure. This means, either `tree` or `list` must be `true` for any given node at any time and they must never both hold true for the same node. The fact that we defined them as abstraction predicates further ensures, that this invariant holds even for summary nodes. This allows us to make different constraint rules, depending on whether a node is a tree node or a list node.

The next three predicates are standard predicates, that do not need explaining. It gets interesting again when we look at the instrumentation predicates. We define a meta-reachability, pragmatically named `reachable`, that expresses reachability over three different pointers and over two different data structures. This is simply expressed as the reflexive, transitive hull of the `linked` predicate. Note, that it is also defined as anti-symmetric, which makes sharpening much easier when we materialize nodes and this is a major reason why we strictly separate between forward reachability and the backward reachability of lists.

The next two predicates, `is[n]` and `is[p]`, seem to be rather uninteresting. They express whether or not a node is pointed to by two list elements. This can (or should) never happen in our algorithms, so it is just a nice fail safe feature, right? Not quite.

As it turns out, these are rather needed predicates for sharpening the children lists. Consider the situation shown in Figure 49. In order to keep the example simple, we look at a singly linked list, that is being iterated over by the `cur` pointer. Every time the `cur` pointer goes to the next element in the list, a new node is materialized. The figure shows, what would happen without the `is[n]` predicate, namely TVLA is unable to determine, that the indefinite self loop for the `n` pointer at the `cur` node can actually be sharpened to a definite `false`. The exact same thing happens if we go backwards using the `prev` pointer in a doubly linked list.

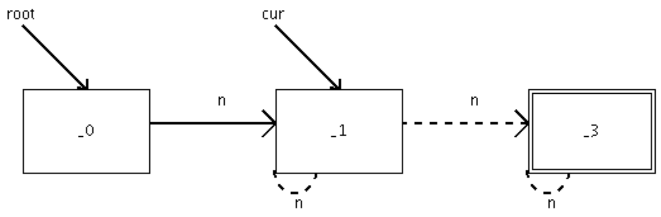


Figure 49: An example of a single linked list, where the `cur` pointer just was moved forward from the `root`, thus materializing a new node.

The `t[n]` and `t[p]` predicates are the standard reachability predicates for doubly linked lists. However, we have modified the definitions a little bit. The first modification is the `!tree` condition, that makes sure, that list reachability is indeed only possible between list nodes of the same list. Looking at the definition of `t[n]`, we see that there is an exception to this however! The exception is, that there is indeed also list reachability from a tree node to its corresponding child list and vice versa. This is necessary, to distinguish between general list nodes (that may be summarized in a summary node) and the specific list nodes that constitute the child list of a specific tree node. This is illustrated in Figure 50. Without the `r[n,arg]` predicate drawn in red, the list summary nodes would be abstracted together into a single summary node. The `r[n,arg]` predicate is directly defined through the `t[n]` predicate. In the backwards reachability case, it is needed to be able to set the `clist` pointer to the rest of the list if we have to remove the first element in the child list. Note that another reason why I wrote the condition `!tree` in these definitions is, that I did not want the tree nodes to be able to list-reach themselves. Thus the reachabilities are not defined as `reflexive` as usual, because that would mean every single node in the data structure would have a self loop of those two predicates. Instead the reflexivity is defined manually later on through constraints, to only affect list nodes.

The last three predicate definitions are pretty straightforward. For each pointer variable we define three reachabilities:

- Predicate `br[x]` defines reachability over the `prev` pointer. This only works on the list level, with the addition of the one tree node that the children list belongs to.

- Predicate `r[n,x]` defines reachability over the `next` and `clist` pointers. This one is important for what is shown in Figure 50.

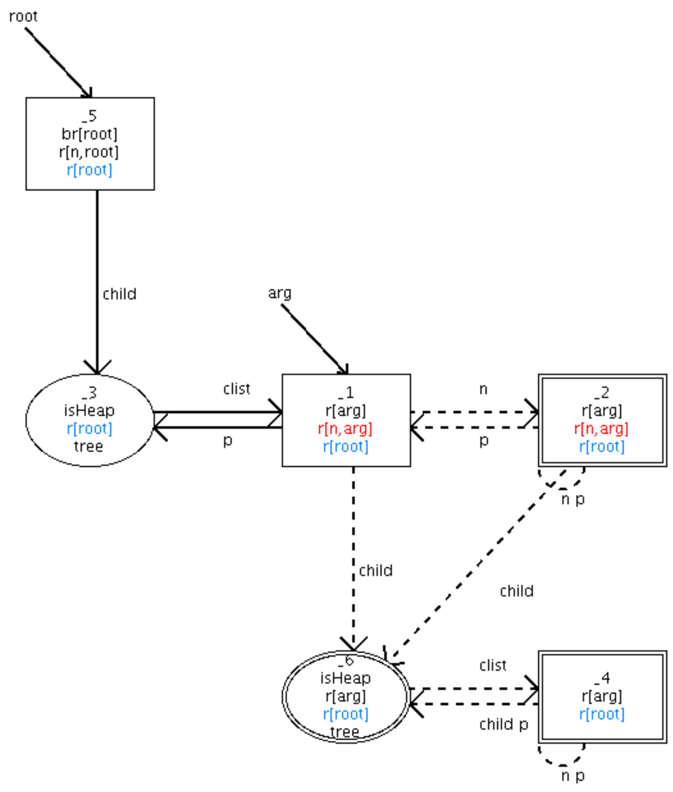


Figure 50: A standard example for a heap data structure. Here we look at the children list of the root node that is being iterated over by the `arg` pointer. As can be seen, the list nodes (drawn as rectangles) are all reachable by the same pointers (`root` and `arg`). The only way to keep the two summary list nodes separate is the list reachability drawn in red.

- Predicate `r[x]` defines the meta-(forward)reachability over all kinds of pointers and object types with the notable exception of the `prev` pointer.

Aside from the above mentioned special cases, reachabilities are needed as a means to express connections. That means, we can only deduce that a list is connected and thus, that the pointer variables are indeed definite, because we know that there must be a way to reach later nodes. The fact that we have multiple types of data structures in this specification means, that we also must have multiple kinds of reachabilities. We have the meta-reachability, that tells us that the whole heap is connected, the forward list reachability, that lets us iterate over lists and keep them apart from each other and lastly the backward list reachability that allows us to iterate backward over the list, so we can delete a list element and reconnect the list afterwards.

4.2 Constraints

In a specification this complex, there are a lot of constraints needed, for lists, for trees and for the combination of both. I will list and briefly explain those that changed and that are most interesting, as a comprehensive explanation would exceed the scope of this section. I will start with the constraints concerning the reachability predicates:

```
foreach (x in PVar) {
  %r !br[x](v) & x(v1) ==> !t[p](v1,v)
  %r br[x](v1) & !br[x](v2) ==> !p(v1,v2)

  %r tree(v1) & list(v2) & x(v2) & A(v4)!n(v2,v4) & reachable(v2,v3)
     & !reachable(v1,v3) & v2!=v3 ==> !child(v2,v1)

  %r x(v1) & list(v2) & !r[x](v2) & reachable(v1,v3) & v1!=v3
     ==> !child(v2,v3)
  %r x(v1) & list(v2) & !r[x](v2) & reachable(v1,v3) & v1!=v3
     ==> !n(v2,v3)
  %r x(v1) & tree(v2) & !r[x](v2) & reachable(v1,v3) & v1!=v3
     ==> !clist(v2,v3)
  %r x(v1) & list(v2) & !r[x](v2) & reachable(v1,v3) & v1!=v3
     ==> !p(v3,v2)

  %r r[n,x](v) & x(v1) ==> reachable(v1,v)
  %r r[n,x](v) & x(v1) & list(v1) ==> t[n](v1,v)
  %r r[x](v) & x(v1) ==> reachable(v1,v)
  %r r[x](v1) & !r[x](v2) ==> !reachable(v1,v2)

  %r r[x](v1) & !r[x](v2) & r[x](v3) & !reachable(v2,v3)
     ==> !reachable(v2,v1)
  %r x(v1) & linked(v1,v2) & reachable(v2,v3) & v1!=v2 & v1!=v3 & v2!=v3
     ==> !child(v1,v3)
  %r r[x](v1) & r[x](v2) & r[x](v3) & v1!=v2 & v2!=v3 & v1!=v3
     & reachable(v1,v3) & !reachable(v1,v2) & !reachable(v2,v1)
     ==> !reachable(v2,v3)
}
```

The first two constraints are very basic, but help sharpen the reachability when materializing nodes. The third one is more interesting, what it says is, that if we have some tree node and a list node with a pointer on it and that list node is a sentinel (i.e., has no successors) and that list node can reach some other node, that is not itself, but that cannot be reached by the tree node, then the tree node cannot be the child of the list node. This rather complicated sounding formula is only needed to make sure that there are no indefinite child pointers going out of sentinels and to keep subtrees separate if we cut a list element out of the list.

The next four constraints just help to sharpen to various connection pointers that exist in our data structure. They basically all just say, that if pointer x cannot reach a list node v_2, but can reach some other node v_3 then v_3 must not be reachable from v_2 by any means. The only exception is the last of the rules, that concerns the p predicate. Note that here v_2 and v_3 are switched, which means, that it must never happen, that you can only go back to an element, but there is no way to go forward again.

The next four are simple again and just sharpen the reachable predicate. Despite the fact that these look very simple, TVLA is not able to deduce them by itself.

The next three are a little more intricate. The first says, that if I have a node v_2 that is not reachable by a pointer x and two nodes that are reachable by that pointer, then v_2 either can reach both of them or none. This follows from the general tree properties: v_2 either sits higher up in the tree in the same subtree and can then reach both, or it is in a different subtree and cannot reach either.

The second one says if x points to node v_1 and has a direct link to another node v_2 then none of the reachable nodes from v_2 can have a child pointer from v_1 (except v_2 itself of course). This is needed to eliminate indefinite child pointers when iterating over lists, i.e., materializing list nodes.

The third and last one enforced the tree invariant, that nodes in different subtrees may not reach each other. In this case v_1 and v_3 are in the same subtree, but v_2 is in a different sub tree, thus v_2 cannot reach v_3. This too was necessary to sharpen some unwanted indefinite reachable values to false if we materialize different subtrees.

```
//add the reflexivity manually to dle for only tree nodes
%r v1==v2 & tree(v1) ==> dle(v1,v2)

//only tree nodes must be comparable
%r !dle(v1, v2) & tree(v1) & tree(v2) ==> dle(v2, v1)

//add reflexivity to list nodes
%r v1==v2 & list(v1) ==> t[n](v1,v2)
```

As I already mentioned earlier, reflexivity can pose a problem when there are multiple data structures present. This is due to the fact, that if a predicate is defined as reflexive, then TVLA will automatically create a constraint, that makes sure, that every node in the shape graph must have that predicate. But if we look at our heap, then it does not make sense to have a *data*-less-than predicate for list elements, simply because these do not contain any data values besides the pointers. Thus we exempt the list nodes from the dle rules, by stating that only tree nodes must be comparable with the dle relation. The first constraint forces reflexivity on tree nodes and the second makes sure all tree nodes are comparable, i.e., there must hold at least one dle relation between any two tree nodes.

The third constraint adds reflexivity to the t[n] predicate, but since this predicate is explicitly used as list reachability it again makes no sense, that it applies to tree nodes as well. So this is just the respective case of the first constraint, only for list nodes this time.

```
//heapness
%r isHeap(v1) & reachable(v1,v2) & tree(v2) ==> isHeap(v2)
%r isHeap(v1) & reachable(v1,v2) & tree(v2) ==> dle(v1,v2)
```

These two constraints are sanity checks that make sure that the heap properties are obeyed. The first constraint says, that if a tree with root v_1 satisfies the heap property, then all its subtrees also have to satisfy the heap property. The second constraint gives meaning to the heap property: If a tree with root v_1 is a heap, then it must have a smaller data value than all its descendants, thus we have defined a min-heap.

There are a lot more constraints to consider, over a full page actually, but these constraints mostly follow the form of:

```
%r clist(v1,v2) ==> linked(v1,v2)
%r n(v1,v2) ==> linked(v1,v2)
%r child(v1,v2) ==> linked(v1,v2)
```

This is the definition of the linked predicate for example. While these constraints are necessary, they are not very interesting in terms of understanding the specification. There are a lot of those short constraints, most of which are only there to sharpen predicates where possible.

4.3 Actions

Arguably the most interesting part of any specification is specifying the actions. This is due to the fact, that in actions we have to describe how a program statement changes the heap. This in turn can prove very hard, because when we focus on something, there may be literally dozens of shape graphs created and only a couple that actually make sense. Of course if we did a good job on specifying the constraints, then a lot of these “wrong” shape graphs will be discarded right away, due to violation of some of those constraints. However, due to the indefinite nature of summary nodes, it is often not quite so easy to discard them right away and it might become necessary to change the focus formula slightly. Even worse, sometimes there might be shape graphs, that satisfy all constraints, but still are not possible in that special operation. In that case precondition formulas must be used to sieve out the unwanted shape graphs.

In our analysis we managed to describe the heaps in a very compact way, thus reducing the number of shape graphs to single digits for every program point. But this efficiency comes at the price of higher difficulty when specifying the actions. We shall demonstrate this on some exemplary actions.

```
%action Get_Next_L(lhs, rhs) {
  %t lhs + " = " + rhs + "->" + n
  %f { E(v_1, v_2) rhs(v_1) & n(v_1, v_2) & p(v_2,v_1) & reachable(v_2,v)}
  %p   (A(v1) (!isHeap(v1) | E(v2) clist(v1,v2)))
     & (E(v1,v2) rhs(v1) & n(v1,v2)
        & (E(v3)child(v2,v3) | (A(v4)!t[n](v2,v4) | v2==v4)))
     & (E(v1)rhs(v1) & E(v2)child(v1,v2))
```

```
      & !(E(v1,v2,v4) rhs(v1) & n(v1,v2) & A(v3)!n(v2,v3)
          & isHeap(v4) & reachable(v2,v4))
  %message (!E(v) rhs(v)) ->
            "Illegal dereference to\n" + n + " component of " + rhs
  {
    lhs(v) = E(v_1) rhs(v_1) & n(v_1, v)
    is[p](v) = is[p](v)
  }
}
```

This is the action that models the program statement `lhs = rhs->next`, i.e., we set a pointer to the next element of the list element being pointed at by another pointer (or itself). The focus formula is very straightforward and is the same as in other standard examples. The only thing of note is, that the free variable v is a hint for TVLA that this variable can only be instantiated by a concrete node, or the statement must be false for all instantiations. The interesting and complex part of this action is the precondi-

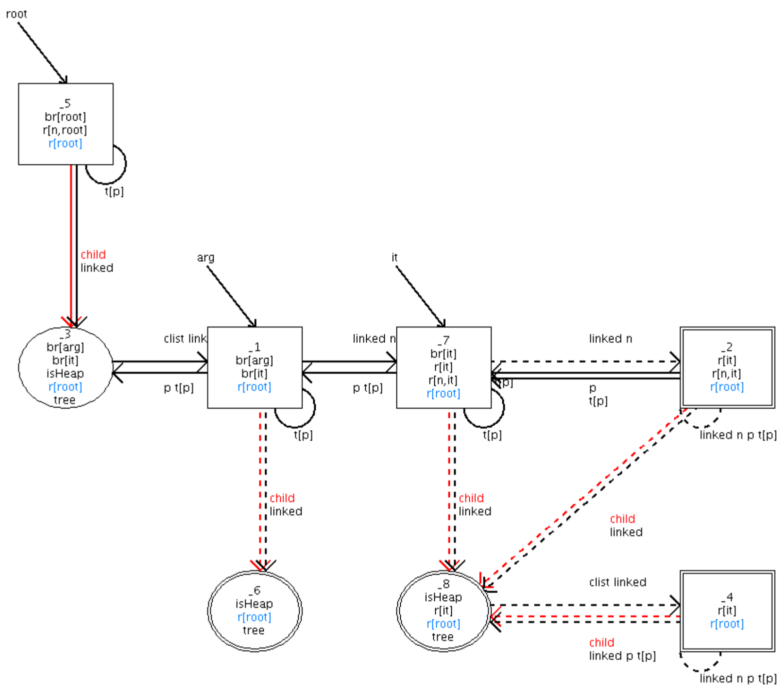


Figure 51: One of the problem cases that arise if the first formula was removed. The problem is that the child of the `arg` pointer has no connection to any list element. This is, however, not possible due to our invariant, that every list at the very least contains a sentinel.

tion, that can be seen as a conjunction of four major formulas. Each of these filters out very specific cases of “bad” shape graphs. They are bad in the sense, that while our integrity constraints cannot invalidate[8] them, they nevertheless describe cases that cannot happen.

Instead of tediously describing the meaning of each major formula, I will show which shape graphs are blocked by each of them, i.e., which shape graphs would appear additionally, if they respective formula was removed.

The first problem case is shown in Figure 51. As a reminder: The rectangle nodes are list nodes, the round nodes are tree nodes. What we can see here is the situation where we are iterating over the children list of the root node and the algorithm just set the `it` pointer to the next child. The tree nodes that are reachable from `arg` and `it` are separated as should be, but there is one significant mistake: The summary tree node below `arg` has no connection to any list node. Why is this a problem? Even if the summary node stood only for one single tree node, i.e., a leaf, then it still must have at least one connected list node, namely the sentinel. In our data structure model there can be no empty lists. Why not solve this particular problem by way of constraints? Because then we could never create or destroy (free) such nodes and this is something that is actually needed when we want to do a `delete-min` operation. Figure 52 shows the correct shape graph for this case.

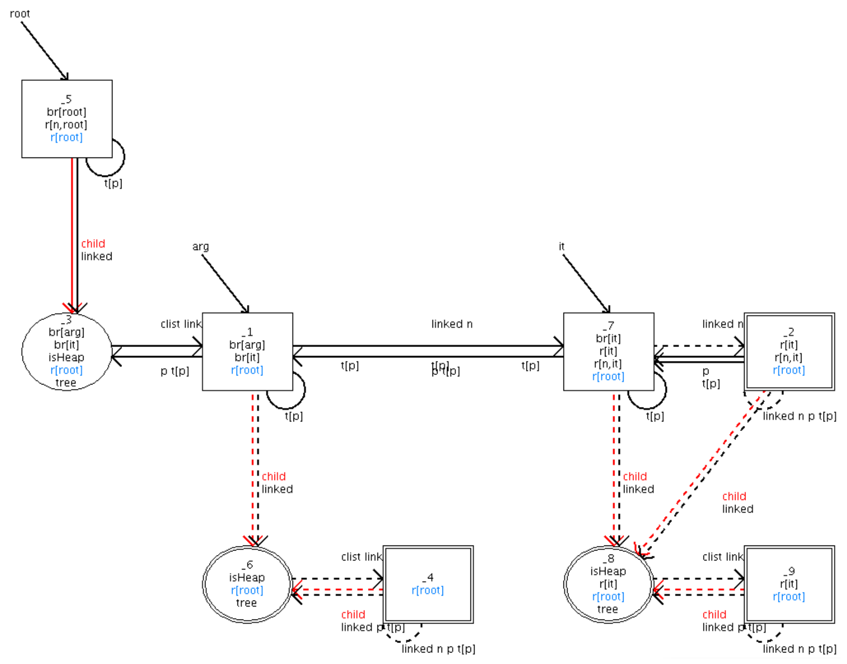


Figure 52: The correct version of the same case that is portrayed in Figure 51

As a last note for this case, the shape graph shown in Figure 51 is actually only one of

[8]It is not entirely correct, that they could not be invalidated through constraints. However, to do so would cause more effort than it is worth. It usually involves nullary predicates that tell the consistency rule that it should not apply in certain phases of the algorithm. That is neither good style, nor is it good for analysis efficiency.

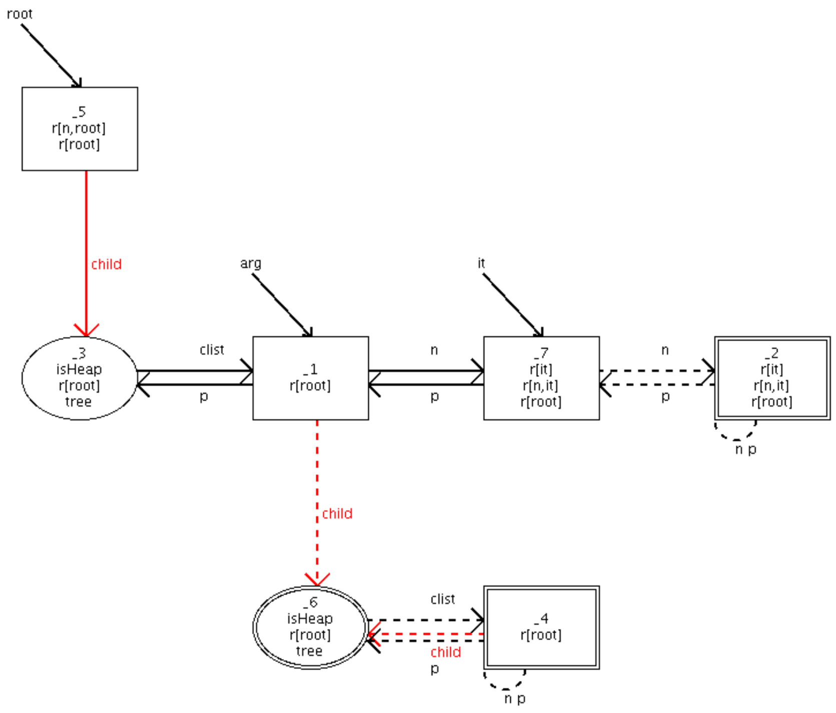


Figure 53: The problematic case if the second formula is missing. The problem here is that we have a non-sentinel list element, that does not point to a subtree. In our data structure we never want to have such list elements.

six problematic shape graphs that arise, because the missing list connection can occur at any sub tree and thus effectively doubles the number of shape graphs at this program point.

Figure 53 shows one of the problematic shape graphs that would appear if the second formula were missing. This time we have list elements that are not sentinels, which do not have a subtree attached. This is another thing that may never happen in a working data structure. The reason why we cannot avoid this with constraints is the same as in the previous case: If we delete elements, then there might exist a list node for a short time that has no attached tree element, because it just got deleted.

Leaving out the third formula has similar results to the second one, as can be seen in Figure 54. Again we have list elements, that are not sentinels, that have no subtree attached. The only difference is the place where they occur, namely at the `it` pointer instead of the `arg` pointer. The reasons of why we cannot easily fix this issue with constraints is the same as in the previous case. As with the other problematic cases before, there are actually multiple instances of this problem.

Figure 55 illustrates the problem that the last formula solves. What we see in the figure is a normal heap shape graph, where all the list elements in the currently iterated list have sub trees attached. To understand why this is a problem, we have to remember our invariants. One of those state, that every child list has to have a sentinel at the end

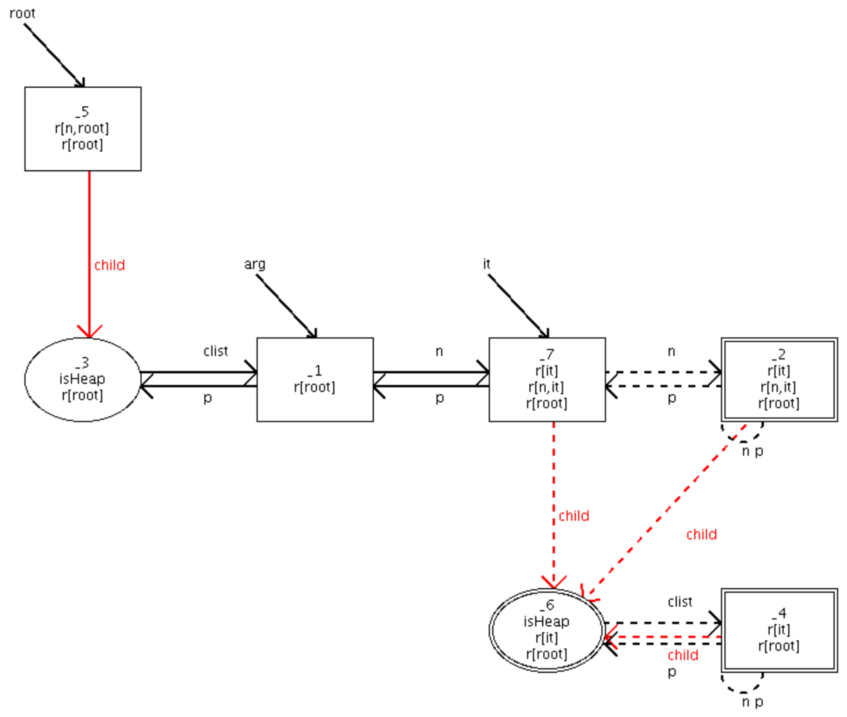


Figure 54: The problem, that formula three solves is quite similar to Figure 53, only this time the missing subtree is at the `it` pointer, instead of the `arg` pointer.

and that this sentinel should never point to any tree element. Thus this shape graph depicts either a list without a sentinel, or a list where the sentinel points to a tree node, either way, this must not happen.

What prevents us from doing a simple solution like the following constraint?

```
list(v1) & A(v2)!n(v1,v2) & tree(v3) ==> !child(v1,v3)
```

Translated the formula says, if there is a list node with no successor, then that list node should not have a child pointer to a tree node. At first thought, this might seem fine and certainly would do the trick, however, there is a slight problem. Constraints are invariants themselves: They have to hold for any program point and any shape graph and any valid instantiation of nodes. Thus let us look again at Figure 55: There are actually two list nodes where the constraint would apply.

One is the problem list node, that we actually want to be gone. The other is the handle for the tree root, i.e., the list node where the `root` pointer is on. This one we do not want to be gone. We actually need it. Because due to how we defined our data structure, whenever we move a tree node (or a whole subtree) around, then we really just move their handle. In other words, keeping a constraint like that would break our analysis, there would not be any shape graphs left!

The rest of the action is pretty much standard and I will not explain it any further. Instead we can look at the one action that really has a big impact on sortation predicates

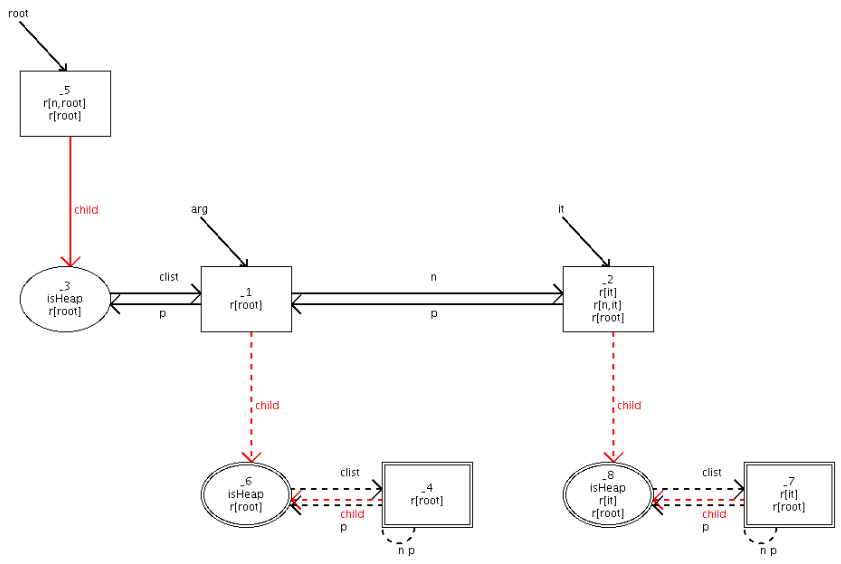


Figure 55: This shows the problematic case, if the last formula is left out. As can be seen, all the list elements have a subtree attached and there are no further list elements left. Thus either the sentinel is missing, or it too has a subtree attached, which is invalid.

in the heap data structure:

```
%action DecreaseKey(lhs) {
  %t "decrease key value of " + lhs + "->child"
  %p E(v) lhs(v)
  {
    dle(v1,v2) = (E(v)lhs(v) & child(v,v1) & !reachable(v,v2)
                             & !list(v1) & !list(v2)
                 ? 1/2
                 : (E(v)lhs(v) & child(v,v2) & !reachable(v,v1)
                               & !list(v1) & !list(v2)
                    ? 1/2
                    : dle(v1,v2)))
  }
}
```

I already said in Section 4.1, that I had to change the way how the sortation predicates worked. This was simply due to the fact that the heap sortation is unlike the standard tree sortation. Most operations will not change the fundamental heap property, else it would not be a heap any longer. The notable exception to this is of course, if the key value of an element is changed, in this case decreased.

To understand the above formula two things are important to remember:

- Due to our invariants, when we call a decrease_key on a tree node, then we give a pointer to its handle, not the tree node itself. I.e., `lhs` will point to an object of type `list_node`.

- TVLA offers a construct that works much like an if-then-else statement, that can be used in formulas. The syntax is (φ_1 ? φ_2 : φ_3), where the φ_i are the usual 3-valued logical formulas. The semantic of the statement is as follows: If $varphi_1$ evaluates to `true`, then the value of the formula is φ_2. If φ_1 evaluates to false, then the value of the formula is φ_3. If φ_1 evaluates to `unknown` then the result is the join of φ_2 and φ_3, i.e., the value of φ_2 if it is equal to the value of φ_3 and `unknown` else.

This means the update formula for the `dle` predicate contains two nested conditionals. The main condition just asks if v_1 is the child of the handle, i.e., the node which data value is to be decreased and if there is another node that cannot be reached from the handle. This means the node is either further up the tree, or it is not in the subtree spanned by the parent of v_1. If this is the case, then the `dle` predicate is set to `unknown`. Note that this only really changes the `dle` values of the ancestors of v_1, because different subtrees in heaps can contain arbitrary values anyway and thus `dle` between subtrees should always be `unknown` anyway. Note that the above case assumes that v_1 is the node which data value decreases, this does not change the reverse direction yet, i.e., the `dle` predicate that goes to v_1 rather than coming from it.

Otherwise the evaluation goes into the next conditional, here it is now assumed, that v_2 is the node that gets decreased. The rest remains exactly the same as in the first case, changing only the `dle` predicates of the ancestors of v_2. In all other cases the `dle` predicate remains unchanged. In particular this is important, so that `dle` values between nodes that are not being decreased remains the same.

4.4 Conclusion and Summary

There are of course a lot more actions and also more constraints than we showed here. The point of this section, however, was not to give a comprehensive listing of the specification, but to show the general concepts and problems encountered. To summarize, there were the following major points to consider:

- We wanted to apply the concepts and intuition of Section 3.1, to create an analysis specification that produced easy to understand output, with emphasis on minimizing the number of shape graphs. To achieve this, we defined our data structure in certain ways, to ensure there would be no unnecessary case distinctions.

- We had to incorporate two different data structures into a new data structure. The difficulty with this was to define consistency rules that either apply to both or that have a way to only refer to one of the two data structures. This was done by using predicate tags for nodes, that determined which data structure they belonged to. (`tree` and `list`)

- Some predicates had to be redesigned, to accommodate the new property of heaps. Most notably the the sortation predicates. All but the basic `dle` predicate were removed and the update of the sortation was relegated solely to actions.
- Other predicates had to be added to model the new data structure and out of necessity to distinguish between different cases. A good example of this are the reachability predicates. While there still are the classic list reachabilities, both forward and backward, we defined a new meta reachability, incorporating both list and tree nodes.
- Narrowing down the number of shape graphs and streamlining the analysis had the side effect, that we could not easily define constraints that took care of all impossible cases while also leaving all the possible cases in. Thus we had to rely more on the precondition option in actions and built more complex preconditions than in other specifications, that took care of blocking impossible shape graphs.

Finally, I would like to point out how specifying an analysis works in practice. While we presented the analysis here in a sequential way, this is of course not how it was designed. It is true, that first and foremost, there comes the phase where the decision is made to analyze a certain algorithm and then the search starts for a suitable program that implements this algorithm. Here the criteria mentioned in Section 3.1 come into play, i.e., what kind of data structure is the best or most efficient to teach the algorithm and is that data structure suitable for good visualization using TVLA. It can and will happen, that the most efficient data structure that might be best for teaching is not the best to use in visualization, so the teacher has to make a decision that will include a trade off between what is practical and what is good to visualize.

Once the program is decided, the real specifying work can begin. While we presented the analysis here in terms of predicates, constraints and actions in sequential order, in reality they all depend on each other. That means it has more a kind of iterative nature, i.e., starting with the basic structure predicates, defining the first basic constraints, specifying the first action. Then look at the output, see what does not look right and start adjusting constraints and actions. Once things look fine, add the next predicates and iterate. If there are problems in shape graphs after an action was performed, then those might be corrected either by adding constraints, or by editing the action. Sometimes constraints added later make some preconditions or even update formulas unnecessary. Such was the case for example in the action `DecreaseKey(lhs)` that we presented earlier: Instead of four major formulas in the precondition, there were five, but one turned out to be unnecessary after all.

This is to illustrate how difficult and inherently suboptimal the specifications for TVLA often can be. There is no tool that can tell us how good an analysis is, this is solely up to the user and ultimately depends on what the analysis is used for.

5 Summary

In this work I presented an unusual approach to algorithm visualization: The traditional method of visualization is to simulate the program on some concrete example and visualize each step on some level of abstraction. Our approach is based on the output of shape analysis, the so called shape graphs. Shape analysis computes all possible heap states of a program at any program point and thus allows for a better meta view, as well as the possibility to not only show invariants, but for the learner to easily deduce them on their own.

Our approach solves or at least reduces some of the biggest problems with algorithm visualizations:

- Instead of having to look for a suitable visualization for every new algorithm to be presented in a lecture, our approach uses only one single visualization tool. On the one hand, that means the users only have to get used to one interface. Since all visualizations are derived from shape analysis output, all visualizations will also have similar style. On the other hand, if there is no visualization for a certain program yet, then the teacher can create one by specifying an analysis for the desired program. This is only a partial solution, because as we tried to show in Sections 3.1 and 4, creating an analysis is not trivial and can be time consuming.

- A maybe even bigger problem than finding a visualization in the first place, may be to integrate it into the lecture without confusing the students. Oftentimes there are different ways to characterize a problem, sometimes also different kinds of notation. All teachers have their own favorite style of running their lectures and thus the visualization should match that style. Often visualizations offer little in the way of customization in terms of layout, annotations, method names or variable names. All of this can easily be configured in our Visualizer or the analysis itself.

I have presented some of the problems that occur with displaying and computing a layout for shape graphs. Some of those are related to the usually huge amount of information, most of which may not be relevant at that point. That means there have to be tools to filter out unwanted data or ways to make the important data stand out. The more fundamental and harder to solve problems are all related to the abstraction and 3-valued logic, namely with what we called “summary nodes”. In Section 3.1 I have shown that these problems are inherent in shape analysis and that it is generally not possible to define a complete order on the nodes, making sensible layout a challenge.

Huge amount of data and thus the resulting confusion, is not only a problem within a given shape graph, but also in the sheer number of shape graphs. Depending on the program and the chosen abstraction, the number of shape graphs per program point can be anywhere from no more than a handful to potentially hundreds or more. Thus I also mentioned the work of Dierk Johannes and the implementation of some of that work into our Visualizer, which tries to reduce the number of shape graphs by partitioning them into equivalence classes, that are specified by the user.

While the meta view that shape analysis provides is ultimately the goal when trying to understand an algorithm, we recognize that sometimes a concrete example can quickly help with the understanding. Thus we implemented a way to generate concrete shape graphs, which work as concrete examples, while still having the style of the abstract shape graphs. This allows us to follow a concrete execution path through a program, while at the same time observing the corresponding abstract path.

Shape analysis is the basis for our approach, it is a powerful method to statically analyze programs to verify properties. That means, however, that many analyses were specified with verification in mind rather than visualization of the output. An analysis that is efficient for proving the correctness of an algorithm may not be suitable for visualization, as the goals of both are sometimes different: Increasing the detail level of the analysis will provide better logical results, but will almost certainly increase the number of shape graphs and help predicates needed and thus making the output that much harder to understand. That is why I tried to give some intuition as to what is good for visualization and what is not. The bottomline is, that we want to avoid any and all information that is not needed for understanding, thus we want to keep the number of predicates and the number of shape graphs as small as possible. That is not easy and often small changes bring big results. That is also one reason why I presented the pairing heap analysis in Section 4.

Last but not least, there is our Visualizer itself: A visualization tool with about twenty thousand lines of code. In Section 2 I described a lot of the functionality. The tool was meant as a test platform for new ideas and thus has undergone a lot of changes through the years. All the shape graph figures in this work are taken as screenshots from this Visualizer.

The source code for the Visualizer can be found on my homepage www-tcs.cs.uni-saarland.de/parduhn/index.html. It is written completely in Java and not dependent on any external packages, only the standard JDK environment is needed. It is compliant with both tvs- and tr-type output files up to version 3.0 of TVLA. It has only two interfaces that need to be maintained for further TVLA versions, one is of course the parser for the TVLA output files and the other is the command line syntax for invoking TVLA directly from the Shape Graph Creation Tool.

5.1 Further Work

5.1.1 Evaluation

While working on this thesis and debugging analyses, the visualizer was an invaluable help, not only for myself, but also for other colleagues that specified analyses. It saved time and was much easier to configure than the standard TVLA output. It also made specifying input shape graphs a lot easier, thanks to the shape graph creator, that allows to create shape graphs similar to a paint program, graphically directly on screen as opposed to having to do it textually. That means it certainly is a great tool for developing shape analyses, but the scope of the thesis was to provide a different approach to algorithm visualization in teaching. Thus, the most important next step should be an evaluation of our Visualizer and thus also an evaluation whether our approach is indeed an improvement to the traditional visualizations.

I had originally planned on doing the evaluation as part of this work, but I faced a simple problem: There are not enough analyses for a class on algorithms and data structures. There are of course quite a few examples included in TVLA, however, they are mostly single or double linked lists and a few simple binary search tree examples. I felt that a reverse list or insert element in a standard binary tree was too simple an algorithm for evaluation, since the differences from our method to standard methods are too small to make much difference. The only more complicated data structures available are pairing heaps, which are not covered in our lectures, and the insert algorithm for AVL-trees. The latter would have been the only serious candidate for an evaluation, but I decided against it for two reasons:

- The AVL-tree insert analysis was created by myself during my work on my diploma thesis. At that time I did not have the same focus and knowledge on specifying an analysis for visualization. As a result there are over a hundred shape graphs at some program points, which is too much for easy understanding. So either I would have to redo the analysis, which I did not have the time for, or the students would have to be somewhat experienced with reading shape graphs, which is also not going to happen if the evaluation only uses this one algorithm.

- Having only part of a data structure was felt to be incomplete. Using AVL-trees as single data structure to test our Visualizer might work if we at least could simulate all the common operations and not only the insert. Additionally it is quite a bit of work to get introduced to the Visualizer and shape graphs, probably too much effort to apply this skill only once.

In summary, there was just not the time to do the evaluation in a way that I felt comfortable with, ideally alongside the whole basic course on data structures and algorithms. That would have allowed us to observe two groups, one with the Visualizer and the other without and then compare not only their learning experiences, i.e. whether they found it easier or faster to understand certain algorithms, but also to compare their accomplishments in the exams.

5.1.2 Easier Analysis Specification

Easier is always better, but in this case it is rather essential. While this call is nothing new, it is maybe more urgent than ever: There needs to be a faster and more automated way of generating shape analyses. When working with TVLA (and most other shape analysis tools), almost everything has to be done tediously by hand in textual form. Only once you sit down and actually try to do it, it becomes clear how much work it is to specify a single analysis, even a small and trivial one. What I propose is a front end, that should address as many of the following points as possible:

- There should be support for any of the common programming languages. At the very least I recommend Java and C++.

- As much of the operational semantics of the programming language as possible should be automatically generated. I.e., the front end should translate the program code into TVP actions.
- A library for standard data structures like lists, trees, arrays and so forth, that automatically defines the relevant predicates and constraints.
- A library for standard input scenarios as TVS files. I.e., no more specifying your own examples from scratch, potentially missing important predicates or setting them wrong.
- Some way for the user to specify what properties they are interested in and what properties are simply not needed in order to keep the analysis as small as possible.

I do realize how difficult this is to implement, it would certainly require a lot of further research, but it would also be instrumental in making our approach easily accessible to everyone.

5.1.3 Implement a Pathfinder

When thinking about visualization using our approach with shape analysis, it is important to keep in mind that there are differences to the normal method. One direct consequence of using abstraction and computing invariants is, that there is no unique, well defined, deterministic path through program execution. Instead all the possible shape graphs form a directed trace graph, i.e., two shape graphs are connected by an edge if one is the successor of the other. Every path through this graph, that starts at one of the start shape graphs and ends at what can be considered an end shape graph, i.e. a shape graph at an exit location[9]. Since there can be arbitrarily many start shape graphs and exit locations, as well as the fact that every shape graph can have arbitrarily many predecessors and successors, navigating through a program execution may be confusing or at least tedious, depending on the algorithm and user knowledge.

Since the goal is to use the visualization in teaching, it should not be assumed that the student already understands the algorithm. That means they might need some assistance to navigate the shape graphs in a meaningful manner. This is where a pathfinder tool may be useful, it is very easy to get lost in the potentially many loops of the graph, or to run into a dead end that is of no great interest. Thus the pathfinder tool should be able to distinguish between the "average" case of the algorithm, special cases and also mirror cases. While it is probably not feasible to do that fully automatically, the person specifying the analysis could give the tool hints which end locations are special or common cases. This would in effect make it easier to view the execution as a sort

[9]An exit location is no real program point, but rather an additional program point introduced by TVLA. This is done because shape graphs at a given program point always describe the heap state prior to executing the code at that program point, so to see what the "last" program statement does, an end or exit program point is added where those shape graphs can go. Also it is sometimes desirable to implement artificial program points in order to check for certain properties in the end shape graphs.

of slide show instead of having to ponder which successor shape graph to take at every split in the path.

References

[1] U. Assmann and M. Weinhardt. Interprocedural heap analysis for parallelizing imperative programs. In W. Giloi, S. Jähnichen, and B. Shriver, editors, *Programming Models For Massively Parallel Computers*, pages 74–82, Washington DC, XXXX. IEEE Press.

[2] R. Baecker. Sorting out sorting: A case study of software visualization for teaching computer science. In J. Stasko, J. Domingue, M. H. Brown, and B. A. Price, editors, *Software Visualization: Programming As A Multimedia Experience*, pages 369–381. MIT Press, 1998.

[3] U. Banerjee, D. Gelernter, A. Nicolau, and D. A. Padua, editors. *Languages and Compilers for Parallel Computing, 6th International Workshop, Portland, Oregon, USA, August 12-14, 1993, Proceedings*, volume 768 of *Lecture Notes in Computer Science*, 1994. Springer. ISBN 3-540-57659-2.

[4] M. H. Brown. *Algorithm Animation.* MIT Press, 1987.

[5] M. H. Brown. Zeus: A system for algorithm animation and multi-view editing. In *VL*, pages 4–9, 1991.

[6] M. H. Brown. Perspectives on algorithm animation. In *Proceedings of the ACM SIGCHI '88 Conference on Human Factors in Computing Systems*, pages 33–38, May 1998.

[7] B.-Y. E. Chang and X. Rival. Relational inductive shape analysis. In *POPL*, pages 247–260, 2008.

[8] B.-Y. E. Chang, X. Rival, and G. C. Necula. Shape analysis with structural invariant checkers. In *SAS*, pages 384–401, 2007.

[9] D. R. Chase, M. N. Wegman, and F. K. Zadeck. Analysis of pointers and structures. In *PLDI*, pages 296–310, 1990.

[10] M. L. Fredman, R. Sedgewick, D. D. Sleator, and R. E. Tarjan. The pairing heap: A new form of self-adjusting heap. *Algorithmica*, 1(1):111–129, 1986.

[11] C. G. Healey and J. T. Enns. Attention and visual memory in visualization and computer graphics (preliminary version). In *IEEE Transactions on Visualization and Computer Graphics.*

[12] S. Horwitz, P. Pfeiffer, and T. W. Reps. Dependence analysis for pointer variables. In *PLDI*, pages 28–40, 1989.

[13] D. Johannes. *Aufbereitung von Shapeanalyseausgaben zur Visualisierung der abstrakten Programmausführung.* PhD thesis, Saarland University, 2010.

[14] N. D. Jones and S. S. Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *POPL*, pages 66–74, 1982.

[15] R. M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations: Proc. of a Symp. on the Complexity of Computer Computations*, page 85–103. Plenum Press, 1972.

[16] A. Kerren and J. T. Stasko. Chapter 1: Algorithm animation, software visualization. In *LNCS 2269*, pages 1–15, 2002.

[17] S. Kleene. *Introduction to Metamathematics*. North-Holland, 1987.

[18] J. R. Larus and P. N. Hilfinger. Detecting conflicts between structure accesses. In *PLDI*, pages 21–34, 1988.

[19] T. Lev-Ami. *A framework for Kleene based static analysis*. PhD thesis, Tel-Aviv University, 2000.

[20] T. Lev-Ami and S. Sagiv. Tvla: A system for implementing static analyses. In *SAS*, pages 280–301, 2000.

[21] T. Lev-Ami, T. W. Reps, S. Sagiv, and R. Wilhelm. Putting static analysis to work for verification: A case study. In *ISSTA*, pages 26–38, 2000.

[22] S. K. Lodha, J. Beahan, T. Heppe, A. Joseph, and B. Zane-Ulman. Muse: A musical data sonification toolkit. In *Proceedings of International Conference on Auditory Display (ICAD)*, 1997.

[23] A. Mack and I. Rock. *Inattentional Blindness*. MIT Press, 1998.

[24] T. L. Naps, J. R. Eagan, and L. L. Norton. Jhave – an environment to actively engage students in web-based algorithm visualizations. In *Proceedings of the 31st SIGCSE Technical Symposium on Computer Science Education*, page 109–113, March 2000.

[25] R. A. Rensink. To see or not to see: The need for attention to perceive changes in scenes. In *Psychological Science 8, 5*, page 368–373, 1997.

[26] R. A. Rensink. Seeing, sensing, and scrutinizing. In *Vision Research 40, 10-12*, page 1469–1487, 2000.

[27] G.-C. Roman and K. C. Cox. A declarative approach to visualizing concurrent computations. *IEEE Computer*, 22(10):25–36, 1989.

[28] D. Sagi and B. Julész. Detection versus discrimination of visual orientation. In *Perception 14*, page 619–628, 1985.

[29] S. Sagiv, T. W. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Trans. Program. Lang. Syst.*, 20(1):1–50, 1998.

[30] S. Sagiv, T. W. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.*, 24(3):217–298, 2002.

[31] C. A. Shaffer, M. Cooper, and S. H. Edwards. Algorithm visualization: a report on the state of the field. In *SIGCSE*, pages 150–154, 2007.

[32] J. Stransky. A lattice for abstract interpretation of dynamic (lisp-like) structures. *Inf. Comput.*, 101(1):70–102, 1992.

[33] A. Treisman. Preattentive processing in vision. In *Computer Vision, Graphics and Image Processing 31*, page 156–177, 1985.

[34] A. Treisman. Search, similarity, and integration of features between and within dimensions. In *Journal of Experimental Psychology: Human Perception & Performance 17, 3*, page 652–676, 1991.