UNIVERSITY OF GOTHENBURG

Visualization Of A Finite First Order Logic Model

Master of Science Thesis in Computer Science and Engineering

CHRISTIAN SCHLYTER

University of Gothenburg
Chalmers University of Technology
Department of Computer Science and Engineering
Göteborg, Sweden, May 2013

Visualization Of A Finite First Order Logic Model

Christian Schlyter

Examiner: Koen Claessen

University of Gothenburg
Chalmers University of Technology
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden May 2013.

Abstract

Creating visualizations of finite first order logic models can be very beneficial for users studying the models as it provides an additional aid for the user and makes it easier for the user to understand and comprehend the model. This thesis describes methods on how to visualize the different parts of the model, the domain, functions and predicates, and then combining them into a drawing for the user to look at. We will also provide methods on how to reduce a model using function and predicate properties in order to simplify the drawings we make of a model.

Acknowledgements

I wish to thank my family for all the help and support they have given me. Without it I would never have been able to finish my master thesis in computer science.

I would also like to thank my supervisor and examiner Koen Claessen for always being such an inspiring, understanding, patient and kind person.

It has been a long hard road but now it is finally finished.

Contents

Chapter 1

Introduction

Mathematics, in its various forms, is very formal and strict. Being able to visualize mathematics can be very beneficial in getting a better understanding of and acquire new insights into mathematics. It makes it easier to see properties and definitions of mathematics that might otherwise have been missed in its formality. This is because many mathematical definitions and properties have visual counterparts as well.

First order logic is a formal logical system used to define and solve many problems in a wide range of fields, everything from proving geometry properties[4] to solving puzzles[5]. A solution to a defined problem in first order logic is called a model and contains the necessary information that solves the premises and conjectures made by the problem description. There exists many tools to solve these problems using automated reasoning. The program creates a model that solves the problem if it can find one. This model can be very large and cumbersome to interpret. Creating a technique to visualize this model would provide great assistance in understanding the solution.

Bengt.
Mother of Bengt is Anna.
Father of Bengt is Erik.
Mother of Anna is Karin.
Father of Anna is Bo.
Mother of Erik is Siv.
Father of Erik is Gustav.

Table 1.1: Family tree model

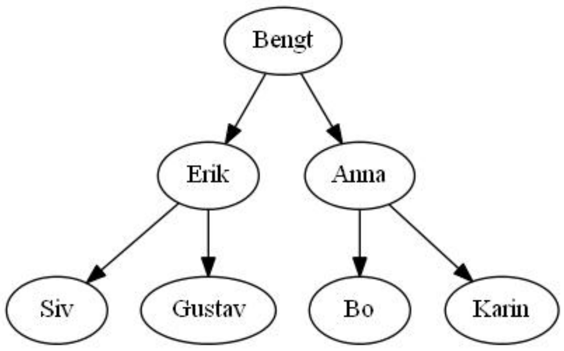

Figure 1.1: Visualized family tree model

Figure 1.1 is a simple visualization of table 1.1, a model of a family tree. The visualization makes it much easier to see how the different persons in the family is related to each other. Getting the same information by just looking through the table can be very tedious and more difficult.

The purpose of this thesis is to visualize a finite first order logic model. We will provide methods on how to visualize the different parts of a first order logic model. We will also show how to connect the different parts together in order to create a simple and comprehensible drawing of the model. The drawing of the model should be equal to its formal definition, i.e. no information should be lost in the transformation from model description to drawing, except in certain cases when it is really wanted.

There exists, as far as I know, no other program or method to visualize a finite first order logic model. The focus of this report is to show how this can be done in a simple and methodical manner. Many of the practical choices then it comes to what color, shape, etc to use has been made with simplicity in mind. If these choices are the best in regards to visibility or usability are questions not raised or answered in this report. The exact details on how to draw a circle, line or text so it does not get tangled up with other circles, lines, text etc is also something that is not considered in this report. It is up to the user, or program of choice, to draw these symbols as he, she or it sees fit. The figures in this thesis has been drawn using the DOT graph drawing program. The instructions sent to DOT has been made in a program I made that transforms logic model descriptions into DOT format using the methods described in this thesis.

Once we have gone through the various methods you can use to visualize a model we will look at some larger examples and how using these visualization methods can help us in acquiring a better understanding of the models we are looking at. Lastly we will show how using known properties of the functions and predicates of a model can be used to decrease the number of drawings we have to make in visualizing the model.

1.1 First Order Logic

First order logic is a formal logical system used to define and sometimes solve many problems, in a wide range of fields. You could see it as an extension of propositional logic, by introducing quantifiers and predicates, to make it a richer and more descriptive language[1]. It exhibits many interesting and beneficial properties such as soundness and completeness[1]. Properties that are crucial to reasoning and solving many of problems that can be expressed in first order logic. It is assumed that the reader already has some familiarity with first order logic as a detailed explanation of first order logic is beyond the scope of this paper. For a more detailed description of first order logic I recommend reading Logic in computer science[1] and Computability and Logic[2].

1.2 Finite First Order Logic Model

A finite first order logic model consists of the domain, the universe of concrete values, function symbols, predicate symbols and their interpretation. The finite in finite first order logic models refer to the domain having a finite number of elements. This makes it easier to visualize the model since there are no infinite domain, functions or predicates to consider.

A description for how the domain elements, functions and predicates will be represented in this report follows.

Every element from the domain can have any name as long as it is different from any other domain element. In this thesis the domain elements are usually labeled with an exclamation mark followed by a natural number, unique for the domain element. It is important to note that just because the model is using natural numbers it is not necessarily possible to deduce any ordering among the domain elements. The numbers are just there to provide each domain element with a unique name. Usually only the number of elements in the domain is written. If it says that the domain size is 5, it is assumed that each element is named and labeled !1,!2,!3,!4 and !5, unless otherwise noted.

Functions and predicates are defined as lists of values. Every possible combination of arguments are presented together with the result for the predicate of function given those arguments. Sometimes the entire table is not listed, only the values that are important for the current visualization.

```
domain size is 2.

constant_function = !1

function(!1,!1) = !1
function(!1,!2) = !2
function(!2,!1) = !1
function(!2,!2) = !2

predicate(!1) = true
predicate(!2) = false
```

Table 1.2: First order logic model

Table 1.2 is an example of how a finite first order logic model is presented in this report. The domain consists of two domain elements, named !1 and !2. There are two functions, a constant function and a function with an arity of two, i.e. it takes two arguments. There is one predicate of arity one.

1.3 Paradox

Paradox is an automated model finder based on the research paper New Techniques that Improve MACE-style Finite Model Finding. [3] The research paper and Paradox have been written and developed by Koen Claessen and Niklas Srensson.

It is the models generated by paradox that has inspired to the creation of this thesis, as it would be beneficial to have a tool to visualize first order logic models in order to easier understand the models generated by paradox. The visualization techniques in this thesis can, of course, be used to visualize all finite first order logic models. The model examples in the larger examples chapter have all been created using paradox. It has also been used for finding and reducing figures using predicate and function properties, which you can read about in the reducing a figure chapter.

1.4 DOT

Graphviz is an open source graph visualization software[6]. Graphviz contains, among other things, dot which is a program used to draw directed graphs. Dot reads an input file, describing how the graph should be constructed, and given this information outputs the graph as an image in a format chosen by the user. Dot uses a hierarchical structure in drawing the nodes and edges. While it is possible to control everything in detail when creating a graph using dot, dot is created to do this automatically in the best way possible. This allows the user to focus on how the graph should look and be connected instead of actually bothering about the fine details of actually drawing the graph.

Chapter 2

Visualizing a finite first order logic model

In this chapter we will show how to visualize a finite first order logic model. We will start with how to visualize a finite first order logic domain, as this domain is used by all other functions and predicates, and then move on to visualizing the functions and predicates. Functions and predicates are drawn using different methods based on the arity of the function or predicate. We will also show how functions can be drawn in a different way as constructor functions. The chapter ends with an explanation on how to draw only parts of a model by erasing or ignoring parts of the domain.

It is up to the user to decide which predicates or functions of a model he or she wants to visualize. It is usually not recommended to visualize all functions or predicates of a model at the same time as this would make the drawing extremely cluttered. Instead it is recommended to only draw the functions or predicates that are essential and important to what one is trying to study about the model. It is also recommended to create several drawings of a model instead of just one, with different combinations of visualized functions and predicates, to avoid cluttering and increasing the comprehensibility of how the model is constructed.

2.1 Visualizing the domain

Every first order logic model has a domain, which consists of a nonempty set of concrete values. Using the same measure as paradox, the domain is simply given as a size, see table 2.1. Each element of the domain is labeled with an exclamation mark followed by a number from one to the size of the domain. Table 2.1 is an example of a domain with three elements, labeled !1, !2 and !3.

Domain size is 3.
(each element is labeled !1, !2 and !3).

Table 2.1: Paradox Domain description

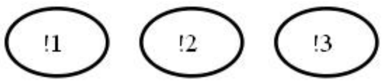

Figure 2.1: First order logic model domain

Figure 2.1 is a visualization of the domain described in table 2.1. Just by looking at the drawing without any prior knowledge of the domain description, one can quickly deduce that the domain has three elements, labeled !1, !2 and !3. One number for each node in the figure.

In order to visualize a finite first order logic model we have to start with the domain since the values contained in the domain are the most basic and elementary parts of a model. All functions and predicates in the model uses them as parameters, and in the case of the functions also as results, to give meaning to the model interpretation. The elements of the domain will be drawn first and functions and predicates will use these drawings, combined with drawings for the functions and predicates, to show how they relate to each other according to the model interpretation.

Visualizing the domain is done by creating a node for each element of the domain. These domain-nodes are given a unique color and shape to distinguish themselves from the functions and predicates that will be drawn later. The shape and color chosen here is a solid round black circle. It makes it simple, pleasing to the eye, distinct and very easy to see. Later on, in section 2.9, when drawing constructor functions a box form will be used. This is because the constructor functions will be drawn inside of the domain node. The box form is used because it makes it easier to fit the constructor function drawings inside the box, especially if there are a lot of them.

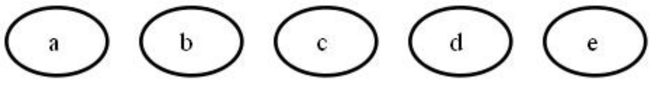

Figure 2.2: Larger first order logic model domain

Figure 2.2 is a visualization of a domain with five elements, all labeled from a to e. It does not matter how many elements there are in the domain. Each domain is visualized in the same way.

Automated tools generally generate elements of the domain that can be very artificially labeled, using letters or numbers to distinguish the nodes from one another. It is very easy to rename the labels of the domain of a visualization to more closely match the original problem description. It is up to the user to rename the domain nodes in a correct and appropriate way.

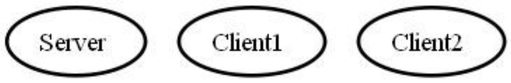

Figure 2.3: Network Domain

Figure 2.3 is a visualization of table 2.1 as a network domain. The !1 element of table 2.1 might represent the server while the other domain elements represent the clients. The nodes have been renamed to more closely match the problem description. This will make it easier to understand the drawing, especially later on then functions and predicates will be drawn.

2.2 Visualizing constant functions

A function of arity zero, also known as a constant function, have no arguments and corresponds directly to one concrete value of the domain. You could say that the constant function and the domain-element are interchangeable. The same view should be apparent in the visualization of this constant function.

```
Domain size is 3.

starting = !1
final = !3
```

Table 2.2: Computer states model

Figure 2.4 is an example of a visualization of the model described in table 2.2 with a domain and two constant functions. This model describes a set of states in a computer program with one final state and one starting state.

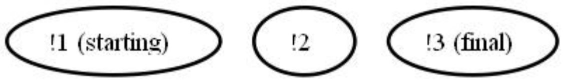

Figure 2.4: Computer states model visualization

Looking at the figure it is easy to see what domain nodes the final and starting states refer to.

Visualizing constant functions is done by inserting the constant function name into the corresponding domain element. This makes it very easy to comprehend which constant functions belongs to which domain element. The constant function name is encompassed in round brackets. This makes it easy to distinguish it from the domain name and predicates with arity one, which will be described later in section 2.3. If several constant functions refer to the same domain elements, the names of the constant functions are added together within the round brackets.

Domain size is 3.
client1 = !1
client2 = !2
server1 = !2
server2 = !3

Table 2.3: Client and server model

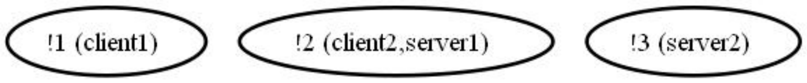

Figure 2.5: Client and server model visualization

Figure 2.5 is a visualization of table 2.3, a model describing a set of client and server nodes. Domain element !2 acts as both a server and client in this example.

2.3 Visualizing predicates with an arity of one

Predicates of arity one, a predicate which takes one argument and depending on the argument gives it result as either true or false, are the most simple predicates to visualize. Due to the fact that it only has one argument it is easy to link the predicate to the element of the domain that it takes as an argument.

Domain size is 3.

professor(!1) = true
professor(!2) = false
professor(!3) = true

Table 2.4: Professor model

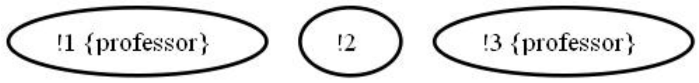

Figure 2.6: Visualization of professor model

Figure 2.6 is a visualization of the domain and predicate listed in table 2.4. The domain represents different human beings. The predicate professor is a predicate with arity one, which given an argument returns true if that person is a professor. Looking at the visualization one can quickly deduce that domain element !1 and !3 are representing persons that are professors.

Visualizing a predicate with arity one is done by visualizing each predicate argument and corresponding result in turn for the predicate. Is is only necessary to visualize the predicate listings that are true since if a node does not contain the predicate name it is assumed that the predicate which takes that domain as an argument has false as its result. This saves space and makes the drawing easier and more comprehensible since there are less items to visualize. If the result for a predicate of arity one listing is true then the name of the predicate is inserted into the domain node that corresponds to that argument. The predicate name is encompassed in curly brackets. This is to easily distinguish it from the domain name and constant functions, described in section 2.2. If several predicates with arity one has an argument which evaluates to true for the domain-element, the predicate names are added together within the curly brackets.

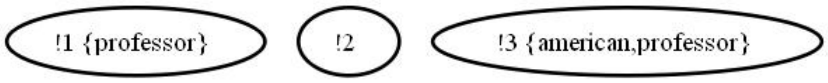

Figure 2.7: Visualization of Professor and American model

Figure 2.7 is a visualization of table 2.5, a model of persons who might be professors or Americans. Table 2.5 and its visualization in figure 2.7 is an

Domain size is 3.
professor(!1) = true
professor(!2) = false
professor(!3) = true

american(!1) = false
american(!2) = false
american(!3) = true

Table 2.5: Professor and American model

example of several predicates of arity one that evaluates to true for an argument representing a person. The element !3 is representing a person who is both an American and a professor.

Domain size is 3.

Tom = !1
Brad = !2
Mary = !3

professor(!1) = true
professor(!2) = false
professor(!3) = true

american(!1) = false
american(!2) = false
american(!3) = true

Table 2.6: Expanded professor and American model

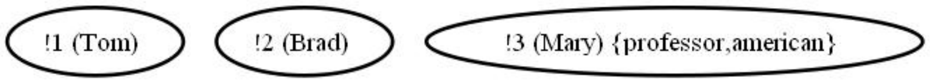

Figure 2.8: Expanded professor and American model visualization

Table 2.6 has been created by expanding the example of table 2.5 by adding constant functions representing the names of the different persons. Figure 2.8 is a visualization of table 2.6. The visualization makes it very easy to see what name is linked to which element and if that element, representing a person, is an American or a professor. Looking at the drawing it is easy to deduce, for example, that only Mary is both an American and professor.

2.4 Differentiating functions and predicates

Visualizing constant functions and predicates with an arity of one has been simple as they are directly linked to a certain domain node. In the coming sections and chapters we will start to visualize more complex functions and predicates with more arguments which will make it harder to easily distinguish different functions and predicates from each other. Using different shapes, colors and labels for edges and nodes can be used to solve this problem and make the visualization more comprehensible and easy to understand.

```
Domain size is 4

net1(!1) = !2
net1(!2) = !3
net1(!3) = !4
net1(!4) = !1

net2(!1) = !3
net2(!2) = !1
net2(!3) = !4
net2(!4) = !2
```

Table 2.7: Network model

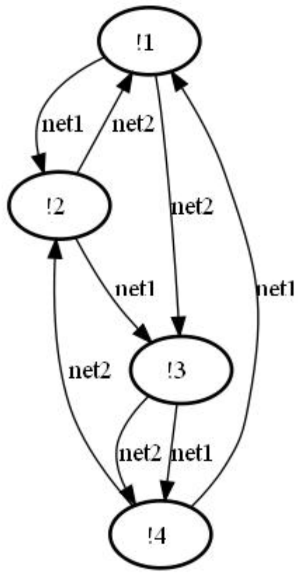

Figure 2.9: Colorless network model visualization

Table 2.7 is a model of a network. The domain represents different computers. Each function of the model, net1 and net2, represents two different network connections. If there is a network connection between two nodes x and y for some network z, then there is a function listing z(x) = y, in the table. Without going into too much detail about the visualization process for the two different functions, that will come later in section 2.5, we can look at a visualization of the table in figure 2.9. Each arrow between nodes represent a network connection between the nodes. Looking at the visualization it is possible to figure out which computers are reachable from which network but it is easy to get the two network connections mixed up.

Using a separate color to draw each function and predicate will help separate the different functions and predicates from each other. Constant functions and predicates with an arity of one is ignored since just changing the color of the name of the node it belongs to will not add any more meaning to the visualization. Every drawn arrow in the visualization will also be labeled with the name of the function or predicate so we know which predicate or function an arrow is supposed to visually represent. Visualizing functions as constructor functions, described in section 2.9, is a different method to view and visualize functions. These functions are drawn using a different shape to distinguish themselves from other functions.

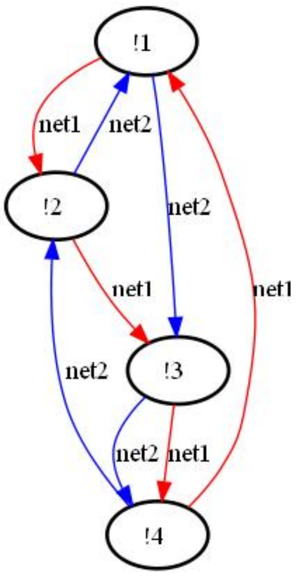

Figure 2.10: Network domain visualization with colors

Figure 2.10 is another visualization of table 2.7. In this visualization we have used different colors when drawing the functions net1 and net2. It is now much easier to see which computers are reachable from which network.

2.5 Visualizing Functions with an arity of one

A function of arity one is a function that takes one argument and returns a result. You could say that with the function there is a a link between the argument and result and that the function transforms the argument into the result. The same view can be applied then visualizing the function. Since the result of the function usually varies depending on the argument of the function, you have to create a drawing for each of the corresponding argument and result of the function, so the function visualization still corresponds to the model description of the function.

```
Domain size is 3

Tom = !1
Jerry = !2
Lisa = !3

loves(!1) = !2
loves(!2) = !3
loves(!3) = !1
```

Table 2.8: People relationship model

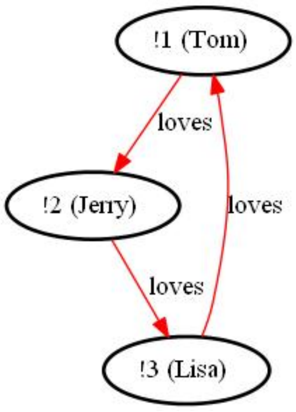

Figure 2.11: Visualization of people relationship model

Table 2.8 is a model representing different persons and how they feel about each other. Figure 2.11 is a visualization of table 2.8. Looking at the visualization it is easy to see that Tom loves Jerry, Jerry loves Lisa and that Lisa loves Tom.

Visualizing a function of arity one is done by creating an arrow that points from the domain-node that represents the argument to the domain-node that represents the result of the function. The direction of the arrow is important since an arrow labeled function of arity one going from node A to node B is interpreted as function(A)=B and not function(B)=A. Each arrow is colored with a separate color unique to the drawing of the function and labeled with the name of the function to easily distinguish it from other functions and predicates. This has to be done for each argument and corresponding result for the function. Looking at the table of 2.8 this means that each table listing of the function loves will be drawn individually to create the whole visualization of the function.

```
Domain size is 2

Tom = !1
Lisa = !2

loves(!1) = !2
loves(!2) = !1
```

Table 2.9: People relationship model

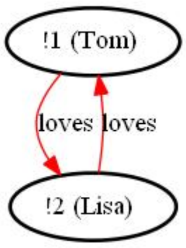

Figure 2.12: Visualization of people relationship model using two arrows

If a function of arity one, function(a)=b also has a listing as function(b)=a, then both these listings can be combined into a single drawing using a single double edged arrow. Then looking at visualization of such a function you will find that there are two arrows going back and forth from the same domain-nodes, as in figure 2.12 which is a visualization of table 2.9. Replacing both these arrows with a double edged arrow as in figure 2.13 will make the visualization simpler as there are less drawings to make without making the visualization harder to comprehend.

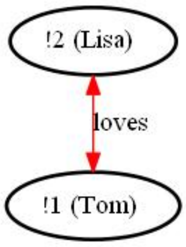

Figure 2.13: Visualization of people relationship model using a single double edged arrow

2.6 Visualizing predicates with an arity of two

A predicate with arity two is a predicate that takes two arguments and depending on the arguments return either true or false. A predicate of arity two is more complicated to visualize than a predicate of arity one as it is not directly linked to one domain-element but two. Since the predicate result varies depending on the what arguments the predicate gets, you have to create a drawing for every combination of arguments and result the predicate has.

domain size is 3

Tom	!1
Jerry	!2
Lisa	!3

father(!1,!1)	false
father(!1,!2)	true
father(!1,!3)	true
father(!2,!1)	false
father(!2,!2)	false
father(!2,!3)	false
father(!3,!1)	false
father(!3,!2)	false
father(!3,!3)	false

Table 2.10: People relationship model

Figure 2.14 is a visualization of the model described in table 2.10. Father is

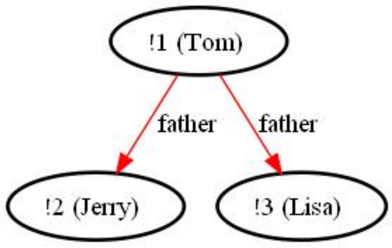

Figure 2.14: People relationship model visualization

a predicate of arity 2. father(x,y) is true if x is the father of y. Looking at the figure it is easy to determine that Tom is the father of both Jerry and Lisa.

Visualizing a predicate of arity two is done by creating an arrow that points from the domain-node that represents the first argument to the domain node that represents the second argument. The direction of the arrow is important since an arrow representing a predicate of arity two labeled predicate going from node A to node B is interpreted as predicate(A,B)=true and not predicate(B,A)=true. As in the case of visualizing predicates of arity one only the predicate listings, for a given predicate of arity two, that have true as a result are visualized. If a predicate of arity two are visualized and a certain argument pair is missing in the drawing it is assumed that the result of the argument pair is false. Each arrow is colored with a separate color unique to the drawing of the predicate and labeled with the name of the predicate to easily distinguish it from other functions and predicates. This has to be done for each combination of the arguments for the predicate that results in true. This means that each table listing for the predicate that has true as a result will be drawn individually to create the whole visualization of the predicate.

```
domain size is 2

Jerry = !1
Lisa = !2

sibling(!1,!1) = false
sibling(!1,!2) = true
sibling(!2,!1) = true
sibling(!2,!2) = false
```

Table 2.11: People relationship model

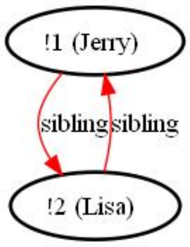

Figure 2.15: Visualization of people relationship model using two arrows

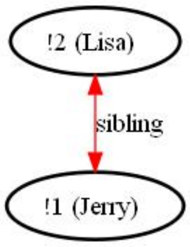

Figure 2.16: Visualization of people relationship model using a single double edged arrow

If a predicate of arity two, predicate(a,b)=true also has a listing as predicate(b,a)=true, then both these listings can be combined into a single drawing using a single double edged arrow. Then looking at visualization of such a predicate you will find that there are two arrows going back and forth from the same domain-nodes, as in figure 2.15 which is a visualization of table 2.11. Replacing both these arrows with a double edged arrow as in figure 2.16 will make the visualization simpler as there are less drawings to make without making the visualization harder to comprehend.

2.7 Visualizing functions with an arity greater than one

A function of arity two or greater is a function that takes two or more arguments and returns a result. Visualizing such a function is more complex compared to previous function visualizations as you have an arbitrary number of arguments that you have to visually link from the domain nodes representing the arguments to the domain node representing the result of the function.

Figure 2.17 is a visualization of table 2.12. Assuming there is an ordering

domain size is 2

max(!1,!1) = !1
max(!1,!2) = !2
max(!2,!1) = !2
max(!2,!2) = !2

Table 2.12: Max function model

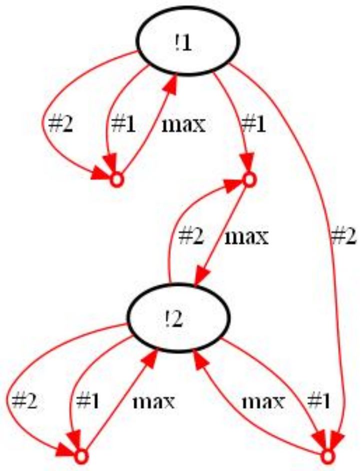

Figure 2.17: Visualization of the max function model

between the elements of the domain, !2 is larger than !1, max is the function returning the largest element as the result. Looking at the drawing one can see how each pair of arguments go together and how the max function arrow always point toward the element representing the largest of the argument elements.

A function listing of arity greater than one is drawn by first creating a small node that will function as a rendezvous point between the arguments and the result. An arrow is then created for each argument going from the domain-element representing the argument to the rendezvous point. The arrow is labeled with the argument ordering so it is possible to separate it from the other arguments going to the rendezvous point. For example in a visualization for a table listing, func(!1,!2,!3) = !4, an arrow labeled #2 represents the second argument !2. A final arrow is then drawn, labeled with the function name, going from the rendezvous point to the domain element representing the result of the function listing. The drawing of the function listing makes it clear what arguments and result it has by just looking at the picture. Drawing the entire function is done by drawing each function listing in turn.

```
domain size is 3

max(!1,!1) = !1
max(!1,!2) = !2
max(!1,!3) = !3
max(!2,!1) = !2
max(!2,!2) = !2
max(!2,!3) = !3
max(!3,!1) = !3
max(!3,!2) = !3
max(!3,!3) = !3
```

Table 2.13: Max function model with a domain size of three

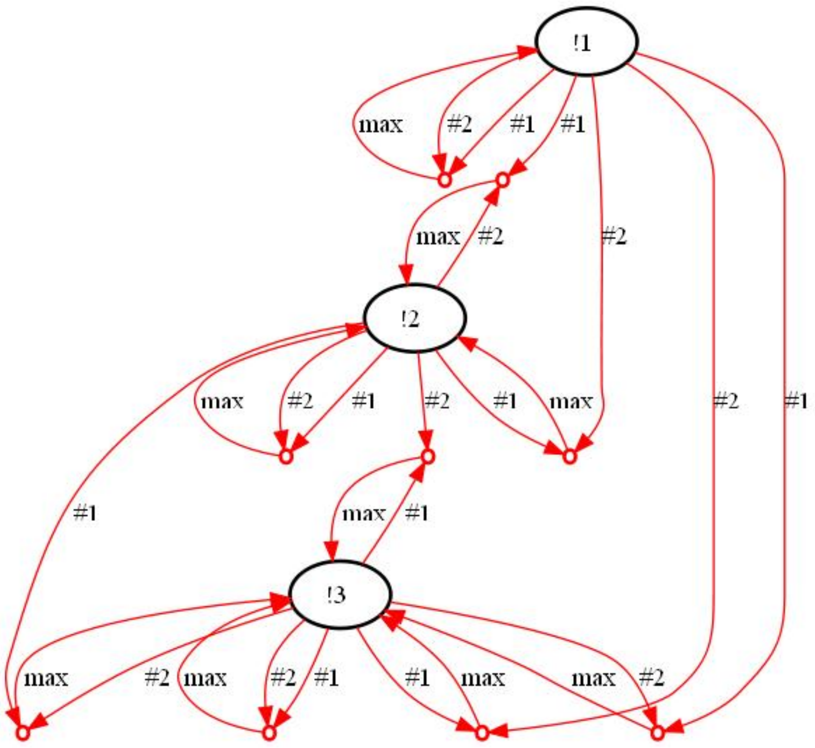

Figure 2.18: Visualization of the max function model with a domain size of three

Figure 2.18 is a visualization of table 2.13. Table 2.13 is similar to table 2.12, describing the max function, but with a larger domain. It is easy to see that the visualization quickly becomes more complex as the domain increases. It is quite hard to distinguish the visualization of individual function listings in the figure, but it is easy to see that most max arrows points toward element !3, which is the largest element in the domain, followed by element !2, and then !1, which is as it should be.

2.8 Visualizing predicates with an arity greater than two

A predicate with arity greater than two is a predicate that takes three or more arguments and returns either true or false. Visualizing such a predicate is more complex than previous predicate visualizations as you have an arbitrary number of arguments that you have to visually link together. As in the case of previous visualization techniques it is only necessary to visualize predicates that are true. If a predicate is visualized and a certain combination of arguments is not visualized it is assumed that the predicate is false for those given arguments.

```
domain size is 3

between(!1,!1,!1) = false
between(!1,!1,!2) = false
...
between(!1,!2,!3) = true
...
between(!3,!3,!3) = false
```

Table 2.14: Model describing the between predicate

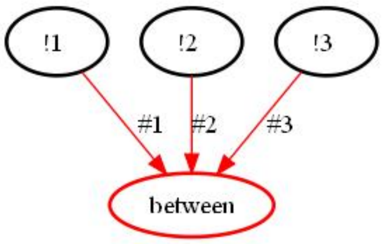

Figure 2.19: Visualization of the model describing the between predicate

Figure 2.19 is a visualization of the model described in table 2.14. Assuming there is an ordering of the elements in the domain, there !1<!2<!3, between(x,y,z) is a predicate that returns true if x<y<z. This is true only

for between(!1,!2,!3) so this table listing is the only one that has to be visualized. Looking at the visualization verifies this claim. You can also see that between is linked with each of the domain-elements representing !1,!2 and !3 which is the arguments for between(!1,!2,!3).

A predicate of arity greater than two is drawn by first creating a node labeled with the name of the predicate that is going to be visualized. An arrow is then created for each argument going from the domain-element representing the argument to the node representing the predicate. The arrow is labeled with the argument number in order to separate it from the other arguments going to the predicate node. For example in a visualization for a table listing, pred(!1,!2,!3) = true, an arrow labeled #2 represents the second argument !2. Only predicate listings that are true have to be visualized. If a predicate listing is not represented in the visualization it is assumed that the predicate has false as a result for that listing. By looking at a visualization of a predicate with arity greater than two it is possible to see what domain-elements makes up a predicate listing and that the predicate is true for such a listing.

```
domain size is 4

between(!1,!1,!1) = false
between(!1,!1,!2) = false
...
between(!1,!2,!3) = true
...
between(!1,!2,!4) = true
...
between(!1,!3,!4) = true
...
between(!2,!3,!4) = true
...
between(!3,!3,!3) = false
```

Table 2.15: Model describing the between predicate with a domain size of four

Figure 2.20 is a visualization of model described in table 2.15, which is similar to table 2.14 except for a larger domain size. Assuming there is an ordering of the elements in the domain, there !1<!2<!3<!4, between(x,y,z) is a predicate that returns true if x<y<z. Looking at the visualization there are four table listings that are true for the predicate between. By inspecting the drawing it is possible to figure out which argument of the predicate makes it true for each table listing. The visualization is, however, not as

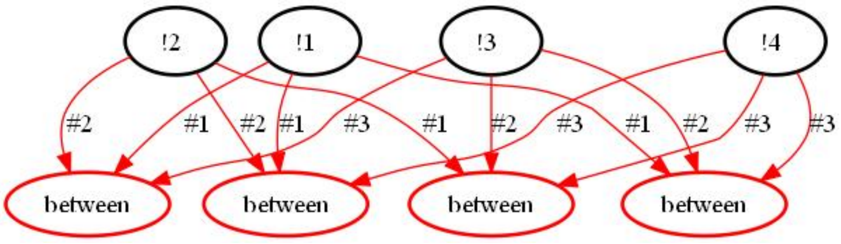

Figure 2.20: Visualization of model describing the between predicate with a domain size of four

easy to interpret as the previous visualization techniques of predicates with lesser arity.

2.9 Constructor Functions

In the previous sections we have looked at functions as something that takes a certain number of arguments and given these arguments you create a result. The same view has been applied then visualizing these functions using arrows to link the domain-nodes representing the arguments to the domain node representing the result. It is also possible to think in reverse. That in order to create a result you have to apply the function with a certain number of arguments. Visually the result is equal to or belongs to a certain domain node and in order to create a value matching the domain element you link the result to the domain nodes representing the arguments. We call these functions constructor functions since they construct a certain value depending on the arguments of the function.

domain size is 4

next(!1) = !2
next(!2) = !3
next(!3) = !4
next(!4) = !1

Table 2.16: Model of a linked list

Figure 2.21 is a visualization of the model described in table 2.16, a model of a linked list. The domain elements represents different list elements and the function next is a function that takes one argument, a list element, and returns the pointer for the next list element in the linked list. Looking at the

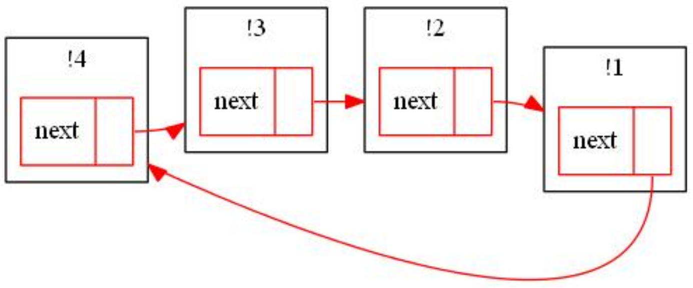

Figure 2.21: Visualization of a linked list

drawing you can see, for example, that to get to the element !2 you need to apply next to the element !1. It is also apparent observing the visualization that the list is a circular linked list.

Since a constructor function is said to belong to a certain domain-element, a constructor function drawing will be made inside the domain-element visualization. The domain elements will be drawn using a square shape instead of the previous round shape. This is to better accommodate the constructor function drawings, as more constructor drawings will easier fit inside a square shaped domain-element visualization compared to the round shape.

As in the case of the previous function drawing techniques the function to be visualized as a constructor function will be drawn in different ways depending on the arity of the function.

A function of arity zero is visualized by encompassing the name of the function in round brackets within a square box. The box is then inserted into the domain-node that corresponds to the result of the function. This is done because a constant function always corresponds to the same domain element, as it has no function arguments. If several constant functions refer to the same domain elements, the names of the constant functions are added together within the round brackets to save place.

Figure 2.22 is a visualization of the model described in table 2.17, a model describing a linked list similar to the one described in table 2.16. The linked list in table 2.17 is, however, not circular. The domain elements represents different list elements and the function next is a function that takes one argument, a list element, and returns the next list element in the linked list. start and end are constant functions detonating the start and end of the list respectively. The drawing makes it very clear which element start and end

domain size is 4

start = !1
end = !4

next(!1) = !2
next(!2) = !3
next(!3) = !4
next(!4) = !4

Table 2.17: Model of a linked list

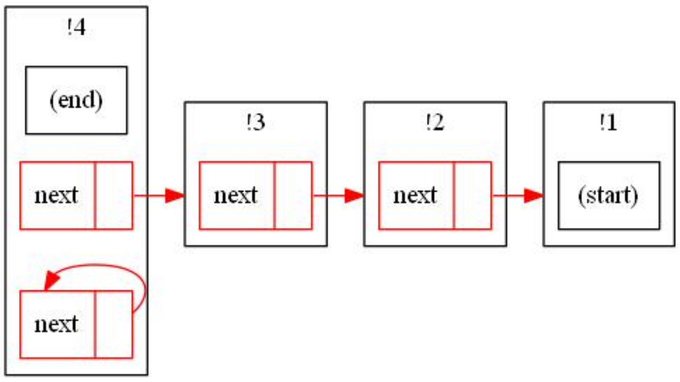

Figure 2.22: Visualization of a model of a linked list

belongs to and how the list is connected.

Visualizing a function as a constructor function with arity greater than zero is done using the same method regardless of arity. For each function listing a series of boxes are created, one plus the arity of the function to be precise. The first box is labeled with the name of the function listing. The other boxes represent the arguments of the function listing and each argument box is attached to the previous box drawing. The boxes will be drawn in the domain-element that corresponds to the result of the function. An arrow will be drawn for each argument of the function listing going from the box representing the argument to the domain node corresponding to the function listing argument. Each function usually consist of several function listings. To draw the entire function each function listing has to be drawn separately.

Figure 2.23 is a visualization of table 2.18 using the constructor drawing technique. Table 2.18 is the same as table 2.13, a model of the max function

domain size is 3

max(!1,!1) = !1
max(!1,!2) = !2
max(!1,!3) = !3
max(!2,!1) = !2
max(!2,!2) = !2
max(!2,!3) = !3
max(!3,!1) = !3
max(!3,!2) = !3
max(!3,!3) = !3

Table 2.18: Max function model

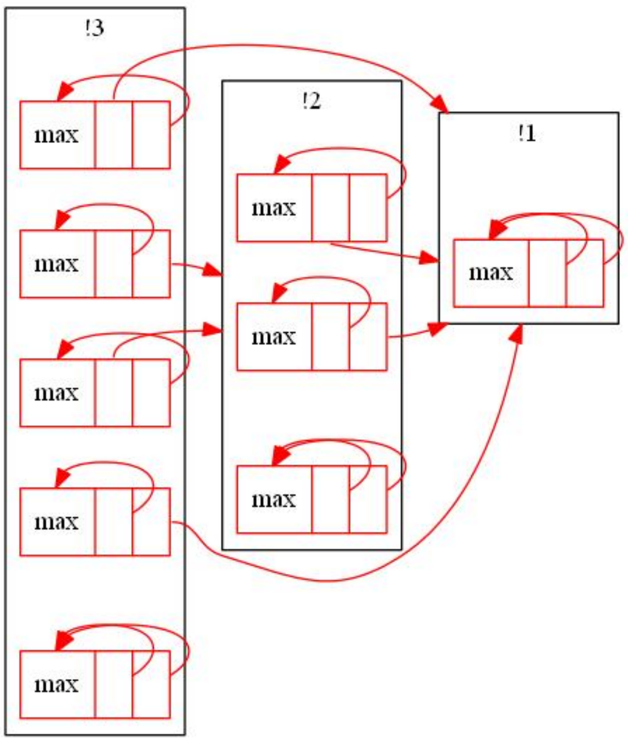

Figure 2.23: Visualization of the max function model

with a domain size of three. Using the constructor function visualization technique the figure of 2.23 is much clearer and comprehensible than the previous drawing technique of figure 2.18. You can easily see that if a max function listing belongs to a domain element, at least one of the function arguments points toward the same domain and the other to a domain representing a lesser value. The domain element three has the most function listings followed by two and lastly the first element. This is as it should be.

2.10 Mixing drawing techniques

We have already seen some small examples in previous chapters how various drawing techniques can be applied at the same time. Figure 2.22 is an example of this, visualizing two functions of arity zero and one of arity one. This is possible because no drawing technique interferes with any other. Once the nodes have been created they are filled with constant functions, predicates of arity one and constructor functions. After that you draw all the other functions and predicates.

domain size is 4

Harry = !1
Harriet = !2
Jake = !3
Mary = !4

sibling(!3,!4) = true
sibling(!4,!3) = true

father_of(!1,!3) = true
father_of(!1,!4) = true

mother_of(!2,!3) = true
mother_of(!2,!4) = true

Table 2.19: Family relationship model

Figure 2.24 is a visualization of table 2.19, a model describing four persons and their relationships. Only the predicate listings that are true for all predicates are listed in order to save space. If an argument combination is not in the table of 2.19 for a predicate, it is assumed to have false as a result for that given predicate. Figure 2.24 is an example of multiple functions and predicates drawn in the same picture. Combined they generate a picture of how the different persons are related to one another. It is easy to see that harry is the father of Mary and Jake, Harriet is the mother of Mary and Jake and that Mary and Jake are siblings.

It is possible to visualize as many predicates, functions and constructor

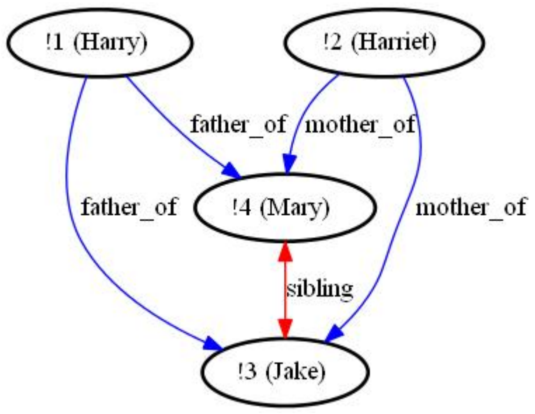

Figure 2.24: Visualization of the family relationship model

functions as you wish. The visualization might however become totally incomprehensible due to all the arrows, circles and boxes cluttering up the drawing. It is up to the user to choose wisely which functions and predicates to visualize and how they should be visualized in order to create drawing that is easy to understand. The visualizations only purpose is to make it easier for the user to understand how a model is composed. This is usually a trial and error process.

In the next chapter, chapter 3, we will look at more complex model examples and how visualizing them can generate a greater understanding for the models, how they are built and how the different functions and predicates interact with the domain elements.

2.11 Removing domain elements to simplify drawing

Removing domain elements from the visualization can sometimes be beneficial in order to more accurately understand different parts of a model or to remove redundant information not needed in order to better understand the visualization. Removing a domain-element is quite simple. Visually it is just to remove the domain element and all function and predicate drawings that has the domain element as a target. All constant functions and predicates of arity one that has the domain element as an argument is also removed with the domain element.

```
domain size is 5

client(!1) = true
...
client(!4) = true
client(!5) = false

server(!1) = false
...
server(!4) = false
server(!5) = true

connect(!1,!2) = true
connect(!2,!1) = true
connect(!1,!3) = true
connect(!3,!1) = true
connect(!3,!4) = true
connect(!4,!3) = true
connect(!1,!5) = true
connect(!5,!1) = true
connect(!2,!5) = true
connect(!5,!2) = true
connect(!3,!5) = true
connect(!5,!3) = true
connect(!4,!5) = true
connect(!5,!4) = true
all other listings have
false as a result, (connect(X,Y) = false)
```

Table 2.20: Network model

Figure 2.24 is a visualization of table 2.20, a model of a network with three predicates, connect, client and server. The domain depicts different computers. Client and server are predicates that returns true if the given computer given as argument is a client or server respectively. Connect(X,Y) returns true if there is a connection between computers X and Y. Looking at the picture it is possible to see how the network is connected, for example the server is connected to all other client computers. It is also possible to see on the double sided arrows that each connection is a two-way connection. Lets assume that we are not interested in how the servers are connected, we are only interested in the clients. removing the servers, in this case element !5, will yield the figure 2.25. Looking at the picture it is much easier to see how the clients are connected without the distraction of the server and its connections.

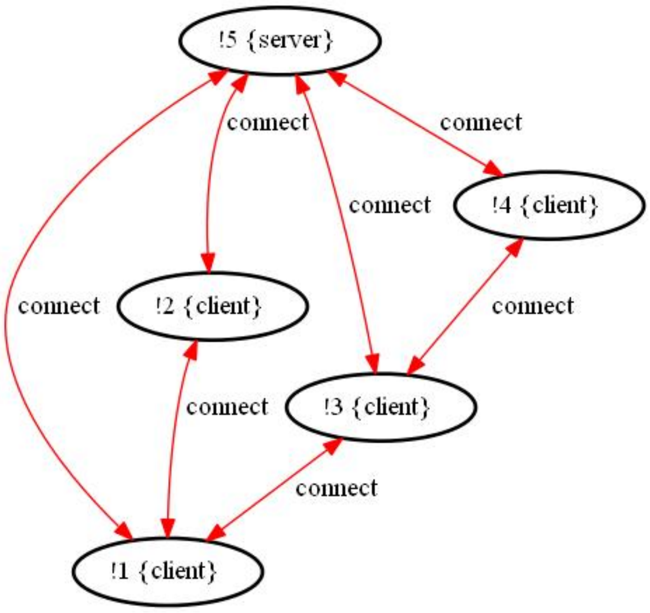

Figure 2.25: Visualization of network model

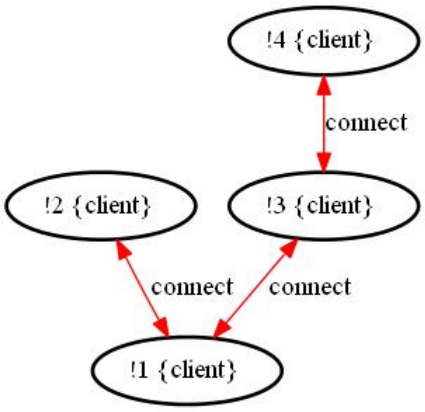

Figure 2.26: Visualization of network model without servers

A simpler and more practical method is to not visually draw the element in the first place. If you know that a certain domain-element should not be drawn then every function or predicate listing that has that domain element as an argument or result is ignored with the domain element. It is important to know that the image created is now not completely equal to the model description. Important information might be lost if you remove domain-elements. It is up to the user to carefully remove domain-elements in order to create a better understand of the visualization.

Chapter 3

Reducing a figure

In this chapter we will talk about function and predicate properties and how we can use them to reduce the number of table listings for a predicate or function in order to create simpler visualizations.

3.1 Function and predicate properties

In logic, and mathematics in general, properties are used to classify and order functions and predicates. Knowing that a function or predicate satisfies a certain property or properties gives useful information of the structure of the function or predicate and how it behaves. We will call properties that deal with functions for operations and properties that deal with predicates for relations. Examples of predicate relations are symmetric, transitive and reflexive and examples of function operations are associative, commutative and idempotence.

```
domain size is 4

connect(!1,!2) = true
connect(!1,!3) = true
connect(!1,!4) = true
connect(!2,!3) = true
connect(!2,!4) = true
all other listings have
false as a result, (connect(X,Y) = false)
```

Table 3.1: Network model

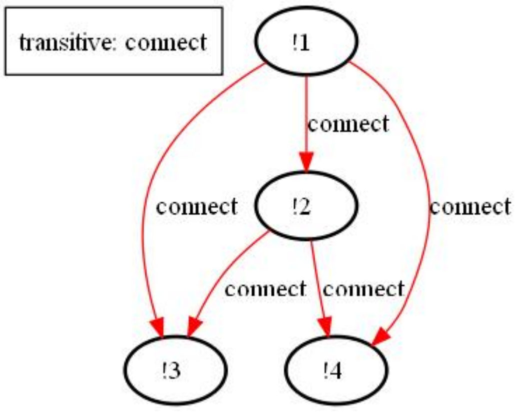

Figure 3.1: Visualization of a network model

Figure 3.1 is a visualization of table 3.1, a model of a network. connect(X,Y) is a predicate of arity two that returns true if there is a connection between computers X and Y. Connect is transitive, meaning that if connect(X,Y) and connect(Y,Z) are true then connect(X,Z) is true as well. The practical meaning of connect being transitive for the network is that if there is a way to get from node X to node Z via another node, then there is always a direct connection between X and Z also.

A function operation or predicate relation is a well defined property. Since the visualization techniques described in previous chapters are based on how the functions and predicates are constructed, formal properties of functions and predicates have visual counterparts as well. The meaning of this is that by just looking at a visualization you might figure out that a function or predicate has certain properties.

An example of this is the transitive relation for predicates, described in the example of figure 3.1. A predicate, lets call it pred, is transitive if pred(X,Y) and pred(Y,Z) are true then pred(X,Z) is true as well. Visually, based on how we draw predicates described in chapter two, this translates to, a predicate is transitive if there is an arrow going from node A to node B and from node B to node C, there is an arrow going from node A to node C as well.

What properties to look for in a model is up to the user to decide and is often related to the field the models are representing. Creating a list of well known general properties, such as symmetric, transitive etc, and using a model checker to see if a function or predicate satisfies these properties is a method that has been used in this thesis.

domain size is 2

max(!1,!1) = !1
max(!1,!2) = !2
max(!2,!1) = !2
max(!2,!2) = !2

Table 3.2: Max function model

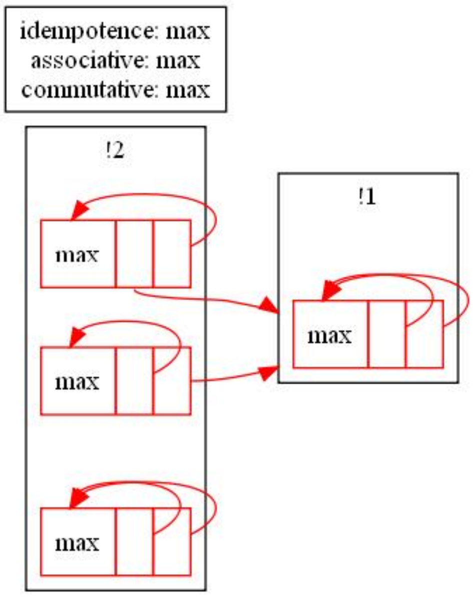

Figure 3.2: Visualization of max function model

Figure 3.2 is a visualization of table 3.2, a model of the max function. The model has two domain elements and assuming there is an ordering between the elements of the domain there !2 >!1, max is the a function returning the largest element of its arguments. The max function has been visualized as a constructor function. The max function has many properties such as idempotence, associative and commutative. This is also displayed in the figure. It might have many more properties, these are just the properties I have searched for and found myself.

3.2 Reducing a figure using function or predicate properties

When a predicate of function has a certain property it is possible to use this property to reduce the number of table listings for the function or predicate. This enables us to draw the figure using less drawings making the figure less cluttered. It is still important that no information is lost when removing table listings for the predicate or function in the sense that it has to be possible to recreate the original listings for the function or predicate with the new reduced listings and the property.

The method is to go through the function or predicate you want to try to reduce with a property and essentially remove any listing that can be considered redundant because it can be inferred knowing the property of the function or predicate. It must be possible to recreate the complete original table listings with the property and the new reduced listings and, this is important, there should only be possible to recreate one and only one complete table listings and that is the original table listings. Otherwise it is possible to deduce several different models of the function or predicate from the new reduced listings and property which makes it impossible to know which one the original is.

function or predicate
name(args) = result1
name(args) = result2
...
name(args) = resultn
property

Table 3.3: Function or Predicate

Table 3.3 describes a function or a predicate. Each row contains the name, arguments and result of the function or predicate. It also contains the property, as a formula, related to the function or predicate we wish to reduce using this property. In order to determine if a function or predicate listing, a row in table 3.3, can be removed using the property we have to know if the table listing is essential or redundant in defining the function or predicate with the property. To do this we need to know if it is possible to generate any other model of the function or predicate when the listing has been removed that is different from the original model of the function or predicate. This is done by inverting the result, setting result to not(result), for the table listing and using a model checker to search for models. If the model of the function with the property and inverted result is satisfiable, i.e. there exits a model, then we cannot remove the listing since if we did it would be possible to generate more than one model using the new reduced table listings with the property. If the model of the property and inverted result is unsatisfiable, i.e. no model exists, then we can remove the table listing. After one row has been investigated we continue with the next row and so on until all rows have been investigated or potentially removed.

domain size is 3

connect(!1,!1) = true
connect(!1,!2) = true
connect(!1,!3) = true
connect(!2,!1) = true
connect(!2,!2) = true
connect(!2,!3) = true
connect(!3,!1) = true
connect(!3,!2) = true
connect(!3,!3) = true
connect is reflexive, meaning that connect(X,X) is true for all X in the domain.

Table 3.4: Network model

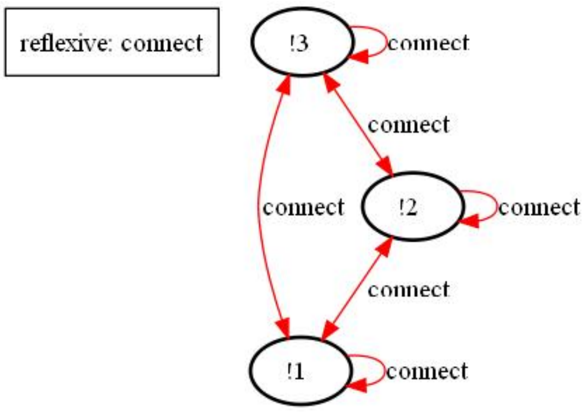

Figure 3.3: Network model visualization

Figure 3.3 is a visualization of table 3.4, a model of a network. connect(X,Y) is a predicate of arity two that returns true if there is a connection between computers X and Y. Connect is reflexive meaning that connect(X,X) is true for all X in the domain. Practically this means that every computer is able to connect to itself. Lets assume that we are not interested if a computer is able to connect to itself, only if it can connect to other computers. Using the reflexive property and reducing the table using this property will result in the table 3.5. Figure 3.4 is a visualization of table 3.5. All the reflexive table listings has been removed. This is because with the new reduced tale of 3.5 and the reflexive property it is only possible to recreate one set of table listings, the table of 3.4. Removing, or reducing, these reflexive table listings makes it much easier to see what we are looking for, namely which computer is connected to which other computer.

```
reduced predicate connect
domain size is 3

connect(!1,!1) = true
connect(!1,!3) = true
connect(!2,!1) = true
connect(!2,!3) = true
connect(!3,!1) = true
connect(!3,!2) = true
connect is reflexive, meaning that connect(X,X) is true for all X in the domain.
```

Table 3.5: Reduced network model

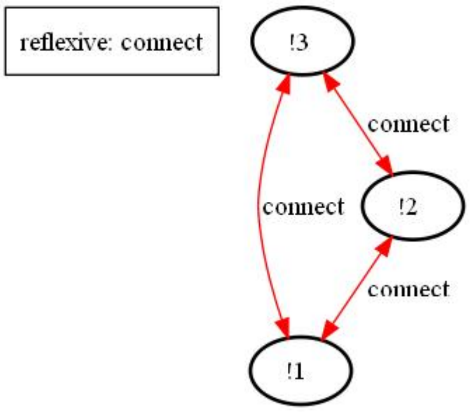

Figure 3.4: Visualization of reduced network model

It is possible to use more than one property when reducing a figure as long as all properties are present when reducing a figure. The connect predicate of table 3.4, a model of a network, in addition to being transitive is also reflexive and symmetric. Reducing the table of 3.4 with all these properties will result in the reduced table 3.6, which has only two predicate listings. Figure 3.5 is a visualization of table 3.6. The problem with the reduced table 3.6, and its visualization in figure 3.5, is that it is so reduced it does not make any sense anymore.

It is up to the user to decide what properties to look for and reduce with. To decide this it helps in knowing how the different properties operates and works, both formally and visually. The case of figure 3.4, a visualization of table 3.5, is an example of this, there it made sense trying to reduce the figure with the reflexive property because reflexiveness in this case meant connecting to itself, something we were not interested in.

```
reduced predicate connect
domain size is 3

connect(!1,!2) = true
connect(!1,!3) = true
connect is reflexive, meaning that connect(X,X) is true for all X in the domain.
connect is transitive, meaning that if connect(X,Y) and connect(Y,Z) are true
then connect(X,Z) is true as well.
connect is symmetric, meaning that if connect(X,Y) is true then connect(Y,X)
is true as well
```

Table 3.6: Heavily reduced network model

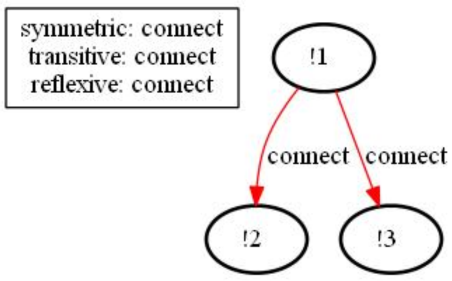

Figure 3.5: Visualization of heavily reduced network model

Reducing a function or predicate by going through the list from top to bottom does not necessary provide the optimal solution if you define an optimal solution as having the least amount of table listings after a reduction.

Table 3.7 is a model of a network. connect(X,Y) is a predicate of arity two that returns true if there is a connection between computers X and Y. Reducing connect using the transitive property, reducing from top to bottom, will generate the reduced table of 3.8, with four table listings. It is, however, possible to generate table 3.9 by choosing a different order of reduction, instead of going from top to bottom, with only three listings. It is

```
domain size is 3

connect(!1,!1) = true
connect(!1,!2) = true
connect(!1,!3) = true
connect(!2,!1) = true
connect(!2,!2) = true
connect(!2,!3) = true
connect(!3,!1) = true
connect(!3,!2) = true
connect(!3,!3) = true
connect is transitive, meaning that if connect(X,Y) and connect(Y,Z) are true
then connect(X,Z) is true as well.
```

Table 3.7: Network model

therefor impossible to know how much we can reduce a function beforehand. We just have to be happy with what we get. Figure 3.6 is a visualization of table 3.8. Figure 3.7 is a visualization of table 3.9.

```
reduced predicate connect
domain size is 3

connect(!1,!2) = true
connect(!1,!3) = true
connect(!2,!1) = true
connect(!3,!1) = true
connect is transitive, meaning that if connect(X,Y) and connect(Y,Z) are true then
connect(X,Z) is true as well.
```

Table 3.8: Reduced network model

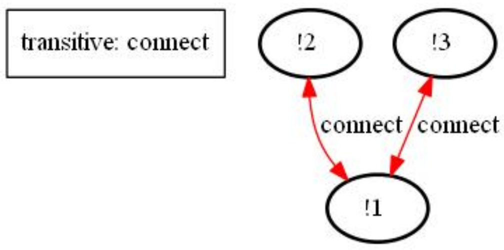

Figure 3.6: Visualization of reduced network model

reduced predicate connect
domain size is 3

connect(!1,!2) = true
connect(!2,!3) = true
connect(!3,!1) = true
connect is transitive, meaning that if connect(X,Y) and connect(Y,Z) are true then connect(X,Z) is true as well.

Table 3.9: Optimally better reduced network model

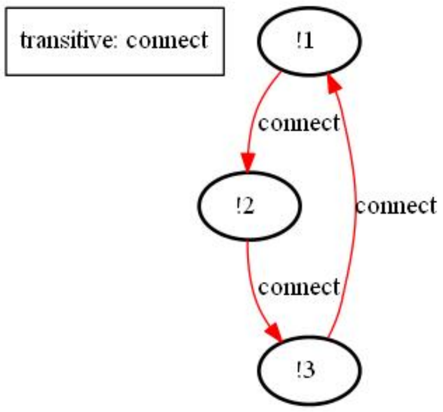

Figure 3.7: Visualization of reduced network model

Chapter 4

Larger examples

In this chapter we will look at some larger examples of problems and their model solutions. Using the visualization techniques in previous chapters we will create visualizations that will make it much easier for us to understand how the model is composed without having to resort to looking through large lists of numbers and values, which might not prove helpful anyway.

All the details of the problems and their model solutions can be located in the appendix but the point of this chapter is not to look at all the details of a problem or a solution, but instead use the visualization techniques we have in order to create drawings we can look at instead. A quick overview of the necessary functions and predicates used in the model description is usually all we need.

4.1 Haskell list with a domain size of three

The table of A.1.1 in appendix describes a list problem in Haskell, defined in first order logic using the tptp.org syntax. Using paradox so solve the problem will generate the model described in table A.1.2, also located in appendix.

We don't have to look at all the details of how lists in Haskell have been described in the specification of the problem, we are only interested in the problem and its solution. A short description of the functions and predicates we need to know about follows so we can understand the problem and the solution to the problem.

Each domain element in the model represents a list.

list is a predicate specifying if a domain element is a list or not.

nil is a constant function equivalent to the Haskell empty list.

tail is a function returning the tail of a list, i.e. the list without its first element.

app is short for append and is a function used to append two different lists together.

Now we have enough information to look at the problem and understand the solution. The specific problem, defined in the back of the table A.1.2 in appendix, is as follows

list(ps) & ps = app(ps,qs) & ps != nil

We are looking for a model there we have two defined lists, ps and qs. ps have to be a list. ps is constructed by appending ps with itself and a list qs. ps can not be the empty list. A model of a solution with a domain size of three is described in table A.1.2. We need to visualize and look at the following functions and predicates list,nil,ps,qs,app and tail in order to see how the solution is composed.

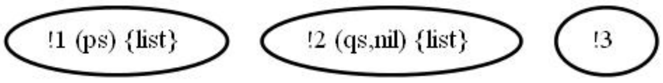

Figure 4.1: Haskell list model

Figure 4.1 is a visualization of the predicate list and the functions ps, qs and nil. We can see that the elements !1 and !2 are lists and that element !3 is not. Element !2 is same as qs and the empty list nil and ps is the same as element !1. Looking at the figure one can see that the formulas list(ps) and ps != nil from the problem description is satisfied. In order to see how ps = app(ps,qs) has been solved we need to also visualize the function app.

Figure 4.2 is a visualization of the functions ps,qs,nil,app and tail. In the previous visualization we saw that element !3 is not a list element and it has therefor been removed from this visualization as it is not relevant to our understanding of the solution to the problem. We can see that ps = app(ps,qs) has been solved by setting qs as the empty list nil. Appending an empty list to any list will always yield the original list. We can also see that the tail element of ps is qs, the empty list.

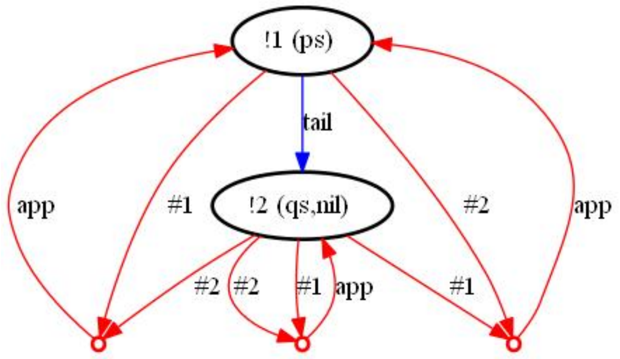

Figure 4.2: Haskell list model

4.2 Haskell list with a domain size of five

The table A.2.1 of appendix describes another list problem in Haskell, defined in first order logic using the tptp.org syntax. Using paradox so solve the problem will generate the model described in table A.2.2, also located in appendix.

The problems have been defined using the same functions and predicates as in the previous example with the addition of the rev, short for reverse, function. Since this problem has a larger domain compared to the previous problem it could also be interesting to look at the cons, short for constructor, function.

reverse is a function that returns the list that is the reversed list of the list given as an argument.

cons is short for constructor and is a function used to construct lists in Haskell.

The specific problem, defined in the back of the table A.2.1 in appendix, is as follows

list(ps) & list(qs) & ps = app(as,bs) & qs = app(rev(as),bs)) & ps != qs

We are looking for a model with four specific lists ps,qs,as and bs. Ps and qs has to be lists. Ps is constructed by appending the list as to bs. Qs is constructed by appending the reversed list of as to the list bs. the list ps can not be the same as the list qs. We need to look at the following functions and predicates ps,qs,as,bs,nil,tail,rev and list in order to see how the solution is composed.

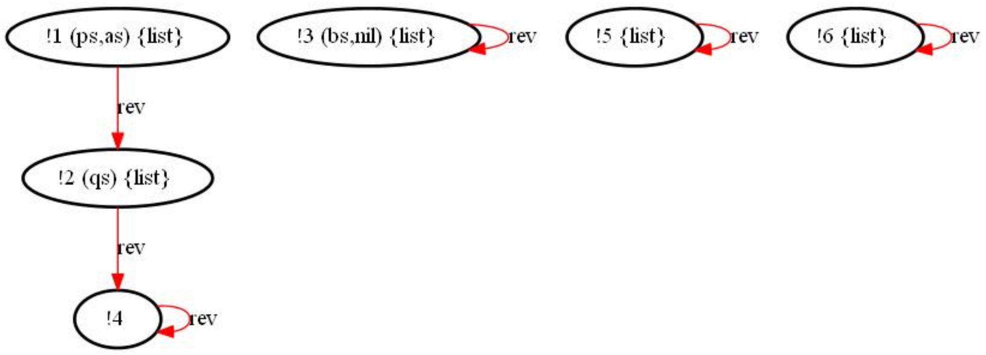

Figure 4.3: Haskell list model

Figure 4.3 is a visualization of the predicate list and the functions ps,qs,as,bs,nil and rev. We can see that all elements are lists except the element !4. Element !1 is the same as ps and as. Element !2 is the same as qs, it is also the reversed list of ps and as. Element !3 is the same as bs and the empty list, nil. Elements !5 and !6 are ordinary lists. Looking at the figure one can see that the formulas list(ps), list(qs) and ps != qs from the problem description have been satisfied. In order to see how qs = app(rev(as),bs)) and ps = app(as,bs) has been solved we need to also visualize the function app.

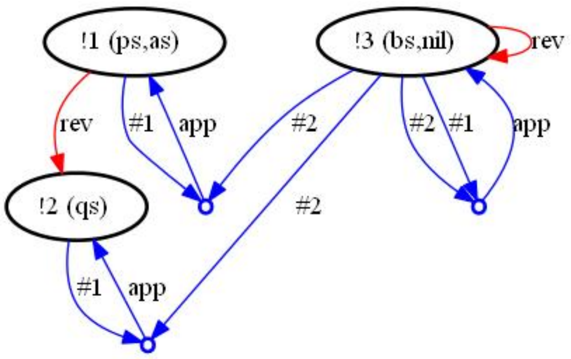

Figure 4.4: Haskell list model

Figure 4.4 is a visualization of the functions ps,qs,as,bs,nil,rev, and app. I have removed the list elements !3,!4 and !5 in order to make it easier to see how the solution to the problem is composed. We can see how ps = app(as,bs) has been solved by setting bs to nil, or the empty list. Appending an empty list to any list will always return the original list. qs = app(rev(as),bs)) has been solved by setting qs as the reversed list of as. Appending the reversed list of as, that is qs, to bs, an empty list, will return the original list qs. Studying the function app we can see that appending any list of as,ps and qs with bs will result in the original list. Since bs, or nil, represents the empty list, this is as it should be when it comes to list behaviour.

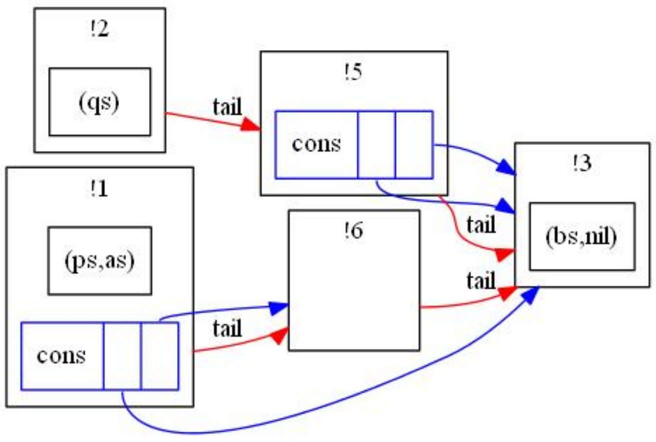

Figure 4.5: Haskell list model

Figure 4.5 is a visualization of the functions ps,qs,as,bs,nil,tail and cons. The cons function has been drawn as a constructor function. The element !4 has not been drawn because it is not a list element, see figure 5.3. Studying this visualization we can get even more information on how the different lists of the domain is related to each other. For example can see that we can create a list element ps or as by using the cons function on element !6 and !3, the empty list.

4.3 Linked c list with a domain size of five

The table A.3.1 of appendix describes a list problem in c, defined using first order logic with the tptp.org syntax. Using paradox so solve the problem will generate the model described in table A.3.2, also located in appendix.

We don't have to look at all the details of how lists in c have been described in the specification of the problem, we are only interested in the problem and its solution. A short description of the functions and predicates we need to know about in order to understand the problem and solution follows.

Each domain element represents a list-node in c, containing data and a pointer to the next element of the list.

Next is a function that takes one argument. Its result is the node-element the pointer of the argument refers to, i.e. the next element of the list.

end is a constant function representing the end-node of the list.

start is a constant function representing the start-node of the list

null is a constant function and represent the null pointer in c.

next_tc is a predicate of arity two and represents the transitive closure of the function next, if it had been written as a predicate of arity two instead of a function of arity one. Visually this makes no difference. Visually the transitive closure represents reachability in a graph.

Now we have enough information to look at the problem and understand the solution. The specific problem, defined in the back of the table A.3.1 in appendix, is as follows

next(start)!=start & next(next(start))!=next(start) &
next(next(next(start)))!=next(next(start)) &
next(next(next(next(start))))!=next(next(next(start))).

The problem says that we are looking for a model with at least five elements and that the elements of the list are unique and separate from each other. A model of a solution with a domain size of five is described in table A.3.2 in appendix. We need to visualize and look at the following functions and predicates end,start,null,next and next_tc in order to see how the solution has been created.

Figure 4.6 is a visualization of the functions end,start,null and next. Looking at the figure we can clearly see how the list is composed. The first element of the list is !1, followed by !4, !5, !2 and finally !3. The list element !1 is the start of the list and element !2 is the end of the list. Element !3 acts as the null pointer. It can be interesting to visualize the predicate next_tc to see the reachability of next.

Figure 4.7 might initially look very complex but the explanation for how next_tc works is quite simple. Reachability in this example for next can be described as if there is a path from a start node to an end node, possibly via other nodes, using next, then there is also a direct path from the start node to the end node. We can see, for example, that next_tc(!1,!3) is true, i.e there is an arrow going from element !1 to element !3 for predicate next_tc. This means that it is possible to travel from element !1 to element !3 using the next function. We also see that next_tc is transitive and reflexive. Figure 5.8 has been made by reducing the predicate next_tc using these two

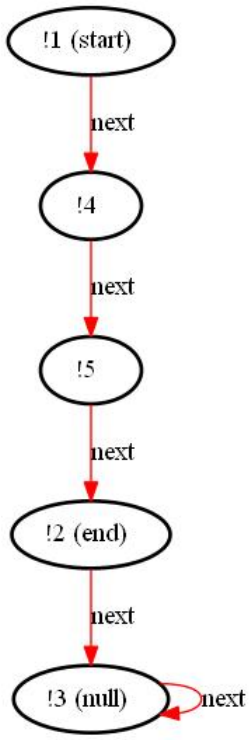

Figure 4.6: C list model

properties.

Figure 4.8 has been created by reducing the predicate next_tc from figure 5.7 by using the two properties of transitivity and reflexivity. Interesting enough the next_tc predicate and next function is almost the same. This is not strange if one considers how the two properties of transitivity and reflexivity affects a visualization. Visually a predicate is reflexive if there is an arrow from the domain element to itself for all domain elements. Visually a predicate is transitive if there is an arrow from element A to Element B and from element B to element C, there is an arrow going directly from element A to element C as well.

4.4 Collinearity

The table A.4.1 of appendix describes a collinearity problem defined using first order logic with the tptp.org syntax. A set of points are said to be collinear if they exist on the same line. The problem description is an attempt to describe collinearity among a set of points using first order logic.

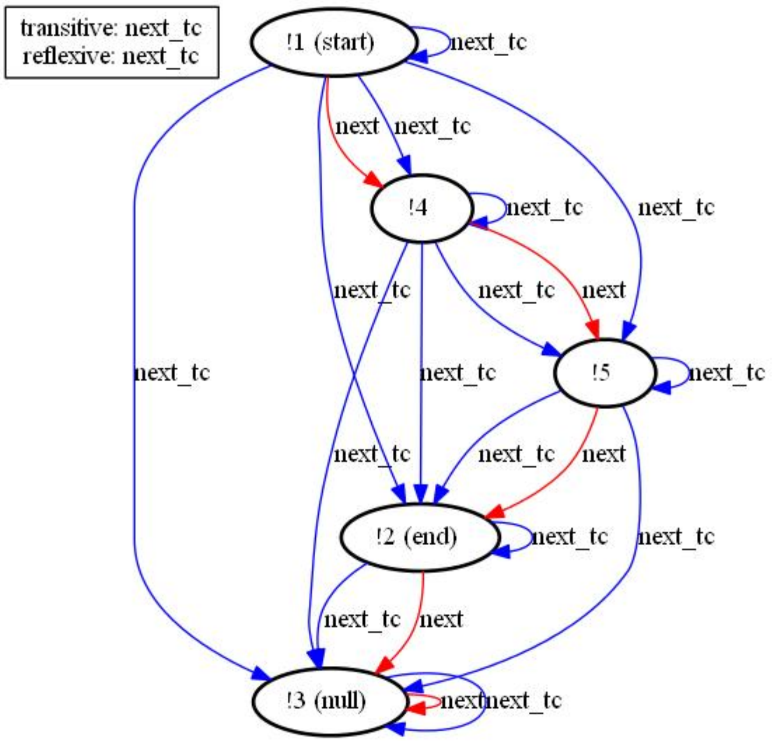

Figure 4.7: C list model

At first it might seem as the axioms and hypothesis of the problem is descriptive enough to satisfactory define that a set of points are collinear but as we will see, this is not the case.

The problem description makes the following claims about collinearity of a set of points.

Two points X and Y are always collinear.

If the points X,Y and Z are collinear, then the points Y,X and Z are collinear as well.

If the points X,Y and Z are collinear, then the points Z,X and Y are collinear as well.

If the points X,Y1,Y2 are collinear and the points Y1,Y2,Z are collinear and Y2 is a different point compared to X,Y1 and Z then X,Y1,Z are also collinear.

The problem description makes the following assumptions or hypothesis

For each pair of points X and Y, there exists a third point Z, collinear with

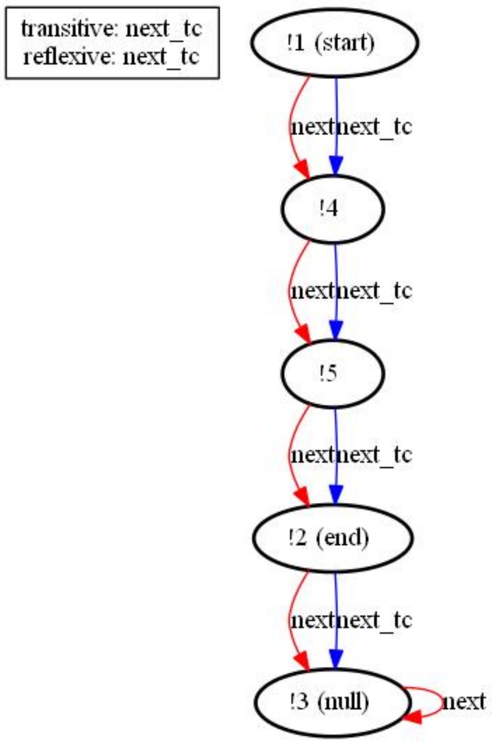

Figure 4.8: Reduced C list model

X and Y. The third point Z is different from the points X and Y.

These axioms and hypothesis should intuitively be enough to describe that all points in a set satisfying these formulas are collinear with each other. In order to test this we add the assumption that there exists three points p1,p2 and p3 that are not collinear and try to find such a model with a model finder, in this case paradox. We should not find such a model if our assumption that all points of the set are collinear but surprisingly we do. The model with a domain size of seven can be found in table A.4.2 in appendix. Lets look closer at the model by visualizing the predicate collinear.

Collinear is a predicate of arity three. Collinear is true if the points it takes as arguments are on the same line, i.e are collinear. Visualizing collinear directly is not a good idea as it has an arity of three and more than forty table-listings that are true which we have to visualize. The visualization would become very cluttered and it would be next to impossible to gain anything from the visualization. Using properties we can reduce the predicate collinear so it will be easier to grasp and understand. Lets look at a couple of properties we can use to reduce the predicate collinear with and why we choose these properties.

The three-distinctive property, formula ![X,Y] : collinear(X,X,Y), i.e there exists points X and Y so that collinear(X,X,Y) is true. This property says that two points X and Y is always collinear. Not having to display collinearity for every two points will save space.

The rotatable property, formula pred(X,Y,Z) =>pred(Y,X,Z), i.e. if the points X,Y and Z are collinear, then the points Y,X and Z are also collinear.

The swappable property, formula pred(X,Y,Z) =>pred(Z,X,Y), i.e. if the points X,Y and Z are collinear, then the points Z,X and Y are also collinear.

The rotatable and swappable property are very alike and together they essentially say that it does not matter in what order three points who are collinear come in. Reducing the figure using these two properties will hopefully remove many of the permutations of the predicate collinear.

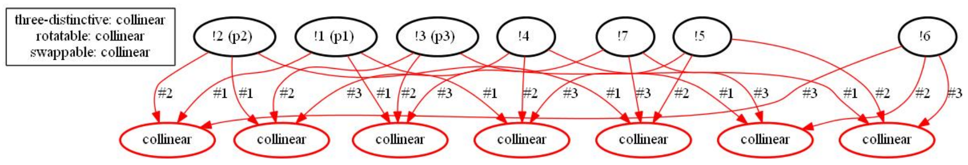

Figure 4.9: Collinear model

Figure 4.9 is a visualization of the functions p1,p2 and p3 and the predicate collinear which has been reduced using the three properties of three-distinctive, rotateable and swappable. Looking at this visualization we can see that !1 is equal to the point p1, !2 is equal to the point p2 and finally !3 is equal to the point p3. We can also see that !1, !2 and !3 are nor collinear as no combination of !1,!2 and !3 are visualized as collinear. This is because first order logic, or induction proof, is not detailed enough to properly describe collinearity of a set of points. Our axioms and hypothesis are therefor not enough.

It is possible to describe collinearity of a set of points but the proof is more advanced and you cannot use first order logic or induction in this simple manner to prove it.

Chapter 5

Conclusion

This thesis describes basic techniques and methods in visualizing first order logic models. Depending on the conditions of the model and what the user want to visulize, some methods and techniques works better than others. The purpose of creating a visualization is to make it easier to understand the model. The biggest problem in visualizing a model is to keep the drawings from becoming to cluttered, which makes it harder to understand what the visualization is supposed to show.

A model with a large domain will always be more difficult to visualize compared to a model with a small domain. Not only is there more domain elements to draw but a larger domain will automatically mean that functions and table listings will be larger since the arguments and results of the functions and predicates have to cover a larger domain. More table listings mean that we have to draw more things in order to properly visualize them.

A function or predicate with a large arity will be harder to visualize than a function or predicate with a small arity. This is because the number of things to visualize, arrows, dots, circles, boxes, etc is directly related to the composition of the functions and predicates themselves. A predicate listing of arity four will, for example, always require at least four arrows to draw compared to a predicate listing of arity three which requires only three arrows to draw. A function or predicate with a larger arity also usually has more listings to visualize compared to one with less arity. This means you usuallay have to create more drawings to fully visualize a function or predicate with a large arity compared to one with less arity.

The visualization techniques of the functions and predicates can be very similar. This can be a problem as it it not possible to separate a function of arity one or a predicate of arity two by just looking at a visualization, as they are drawn in the same manner. You have to know beforehand which of them is which. This is, however, quite a small problem as one usually knows all the names of the functions or predicates of a model and therefor knows if a particular name belongs to a function or a predicate.

There exists two methods to visualize functions, as constructor functions or ordinary functions. Ordinary functions will be drawn outside the domain elements, while constructor functions will be drawn inside of the domain elements. Only drawing functions as constructor functions can make the domain elements very cluttered while only drawing functions as ordinary functions can make the drawing outside of the domain elements cluttered as it has to share the space with other functions and predicate drawings. A balance, if possible, between these drawing methods is preferable.

With the above written considerations, an ideal model to visualize is a model with a small domain with functions and predicates with low arity. If this is the case then it is usually straightforward to visualize the entire model, with the entire domain and all of the functions and predicates, in the same visualization.

If the model has a large domain with many functions and predicates with a large arity then everything becomes more difficult. One has to carefully consider which functions or predicates to visualize in one drawing. It is usually preferable to make many visualizations and visualize different combinations of functions and predicates. Which functions and predicates to visualize in this manner is up to the user to decide and it usually depends on what parts of the model the user is interested in. We have also shown how it is possible to erase parts of the domain to make the visualization smaller. This is good then you want to concentrate on a smaller part of the model. Another method to reduce the number of things to draw or visualize is to use the techniques described in the reducing a figure chapter, which is aimed at reducing the functions and predicates depending of the properties they have. Using this method demands a deeper understanding of the model as it is up to the user to find and use properties that are relevant to reduce with.

Chapter 6

Future work

I believe that the techniques that has been presented in this thesis provides a good foundation in visualizing first order logic models. It is also a good foundation into further research in order to improve or expand on the ideas presented in this thesis. In this chapter I will provide som suggestions of what I personally think would be interesting to analyze and research.

6.1 Expanding the visualization techniques

I would like to see more methods and techniques then it comes to visulizing functions, predicates and the domain in order to provide the user with more variety and options. Now there is only one method to visualize the domain, one method to visualize predicates and two methods to visualize functions. Good techniques to view and visualize very large models would be benificial, as these are usually the hardest models to visualize. I especially would like to see better methods in visualizing functions and predicates with large arity, as these visualizations have a tendency to become very cluttered very fast.

6.2 Perception

In this thesis I have made decisions on what colors, shapes, etc to use based solely on making the different figures as distinct as possible. These decisions are probably not optimal when it comes to what colors and shapes are best from a human computer interaction point of view. For example a combination of certain colors and shapes might make the visualization easier for the user to interpret compared to other combinations of colors and shapes.

It can be difficult to seperate certain functions and predicates from each other as the visualization techniques are very similar. Using colors, shapes or other patterns might be used to make functions and predicates more distinct.

Bibliography

[1] Mikael Huth and Mark Ryan, Logic in computer science, snd edition, 2004.

[2] George S. Boolos, John P. Burgess and Richard Jeffrey, Computability and Logic, 2005.

[3] Koen Claessen and Niklas Srensson, New Techniques that Improve MACE-style Finite Model Finding, 2003.

[4] Geoff Sutcliffe, http://www.cs.miami.edu/∼tptp/cgi-bin/SeeTPTP?Category =Problems&Domain=GRA, 2013.

[5] Geoff Sutcliffe, http://www.cs.miami.edu/∼tptp/cgi-bin/SeeTPTP?Category =Problems&Domain=PUZ, 2013.

[6] http://www.graphviz.org/, 2013

Appendix A

Appendix

A.1 Haskell list with a domain size of three

A.1.1 Problem description

fof(axiom, axiom, list(nil)).

fof(axiom, axiom, list(tail(nil))).

fof(axiom, axiom, ![Xs] : (list(tail(Xs)) =>list(Xs))).

fof(axiom, axiom, ![Xs] : (list(Xs) =>(Xs=nil |Xs=cons(head(Xs),tail(Xs))))).

fof(axiom, axiom, ![X,Xs] : (list(cons(X,Xs)) =>(head(cons(X,Xs))=X & tail(cons(X,Xs))=Xs))).

fof(axiom, axiom, ![Xs,Ys] : ((Xs!=nil & Ys!=nil & list(Xs) & list(Ys) & head(Xs)=head(Ys) & tail(Xs)=tail(Ys)) =>Xs=Ys)).

fof(axiom, axiom, ![Xs] : (list(Xs) =>(Xs=nil |(list(tail(Xs)) & sub(tail(Xs),Xs))))).

fof(axiom, axiom, ![Xs,Ys,Zs] : (sub(Xs,Ys) & sub(Ys,Zs) =>sub(Xs,Zs))).

fof(axiom, axiom, ![Xs] : (list(Xs) => sub(Xs,Xs))).

fof(axiom, axiom, ![Xs,Ys] : (list(app(Xs,Ys)) =>((Xs=nil & app(Xs,Ys)=Ys) |(head(app(Xs,Ys)) = head(Xs) & list(app(tail(Xs),Ys)) & tail(app(Xs,Ys)) = app(tail(Xs),Ys))))).

fof(axiom, axiom, ![Xs] : (list(rev(Xs)) =>((Xs=nil & rev(Xs)=nil) |?[UnitX] : (Xs!=nil & head(UnitX)=head(Xs) & tail(UnitX)=nil & rev(Xs) =app(rev(tail(Xs)),UnitX))))).

fof(axiom, axiom, (list(ps) & ps = app(ps,qs) & ps != nil)).

A.1.2 Solution model

```
+++ BEGIN MODEL
% domain size is 3


app(!1,!1) = !3
app(!1,!2) = !1
app(!1,!3) = !3
app(!2,!1) = !1
app(!2,!2) = !2
app(!2,!3) = !3
app(!3,!1) = !3
app(!3,!2) = !3
app(!3,!3) = !3


cons(!1,!1) = !3
cons(!1,!2) = !3
cons(!1,!3) = !3
cons(!2,!1) = !3
cons(!2,!2) = !1
cons(!2,!3) = !3
cons(!3,!1) = !3
cons(!3,!2) = !3
cons(!3,!3) = !3


head(!1) = !2
head(!2) = !2
head(!3) = !2


list(!1) <=>$true
list(!2) <=>$true
list(!3) <=>$false


nil = !2


ps = !1


qs = !2
```

```
rev(!1) = !1
rev(!2) = !2
rev(!3) = !3

sK13_axiom_UnitX(!1) = !1
sK13_axiom_UnitX(!2) = !2
sK13_axiom_UnitX(!3) = !1

sub(!1,!1) <=>$false
sub(!1,!2) <=>$false
sub(!1,!3) <=>$false
sub(!2,!1) <=>$true
sub(!2,!2) <=>$false
sub(!2,!3) <=>$false
sub(!3,!1) <=>$false
sub(!3,!2) <=>$false
sub(!3,!3) <=>$false

tail(!1) = !2
tail(!2) = !3
tail(!3) = !3
+++ END MODEL
```

A.2 Haskell list with a domain size of five

A.2.1 Problem description

```
fof(axiom, axiom, list(nil)).

fof(axiom, axiom, list(tail(nil))).

fof(axiom, axiom, ![Xs] : (list(tail(Xs)) =>list(Xs))).

fof(axiom, axiom, ![Xs] : (list(Xs) =>(Xs=nil |Xs=cons(head(Xs),tail(Xs))))).

fof(axiom, axiom, ![X,Xs] : (list(cons(X,Xs)) =>(head(cons(X,Xs))=X &
tail(cons(X,Xs))=Xs))).

fof(axiom, axiom, ![Xs,Ys] : ((Xs!=nil & Ys!=nil & list(Xs) & list(Ys) &
head(Xs)=head(Ys) & tail(Xs)=tail(Ys)) =>Xs=Ys)).

fof(axiom, axiom, ![Xs] : (list(Xs) =>(Xs=nil |(list(tail(Xs)) & sub(tail(Xs),Xs))))).
```

fof(axiom, axiom, ![Xs,Ys,Zs] : (sub(Xs,Ys) & sub(Ys,Zs) =>sub(Xs,Zs))).

fof(axiom, axiom, ![Xs] : (list(Xs) => sub(Xs,Xs))).

fof(axiom, axiom, ![Xs,Ys] : (list(app(Xs,Ys)) =>((Xs=nil & app(Xs,Ys)=Ys) |(head(app(Xs,Ys)) = head(Xs) & list(app(tail(Xs),Ys)) & tail(app(Xs,Ys)) = app(tail(Xs),Ys))))).

fof(axiom, axiom, ![Xs] : (list(rev(Xs)) =>((Xs=nil & rev(Xs)=nil) |?[UnitX] : (Xs!=nil & head(UnitX)=head(Xs) & tail(UnitX)=nil & rev(Xs) =app(rev(tail(Xs)),UnitX))))).

fof(axiom, axiom, (list(ps) & list(qs) & ps = app(as,bs) & qs = app(rev(as),bs) & ps != qs)).

A.2.2 Solution model

```
+++ BEGIN MODEL
% domain size is 6


app(!1,!1) = !4
app(!1,!2) = !4
app(!1,!3) = !1
app(!1,!4) = !4
app(!1,!5) = !4
app(!1,!6) = !4
app(!2,!1) = !4
app(!2,!2) = !4
app(!2,!3) = !2
app(!2,!4) = !4
app(!2,!5) = !4
app(!2,!6) = !4
app(!3,!1) = !4
app(!3,!2) = !4
app(!3,!3) = !3
app(!3,!4) = !4
app(!3,!5) = !5
app(!3,!6) = !6
app(!4,!1) = !4
app(!4,!2) = !4
app(!4,!3) = !4
app(!4,!4) = !4
app(!4,!5) = !4
app(!4,!6) = !4
app(!5,!1) = !4
app(!5,!2) = !4
app(!5,!3) = !5
app(!5,!4) = !4
app(!5,!5) = !4
app(!5,!6) = !1
app(!6,!1) = !4
app(!6,!2) = !4
app(!6,!3) = !6
app(!6,!4) = !4
app(!6,!5) = !2
app(!6,!6) = !4

as = !1

bs = !3

cons(!1,!1) = !4
cons(!1,!2) = !4
cons(!1,!3) = !4
cons(!1,!4) = !4
cons(!1,!5) = !4
cons(!1,!6) = !4
cons(!2,!1) = !4
cons(!2,!2) = !4
cons(!2,!3) = !4
cons(!2,!4) = !4
cons(!2,!5) = !4
cons(!2,!6) = !4
cons(!3,!1) = !4
cons(!3,!2) = !4
cons(!3,!3) = !5
cons(!3,!4) = !4
cons(!3,!5) = !4
cons(!3,!6) = !1
cons(!4,!1) = !4
cons(!4,!2) = !4
cons(!4,!3) = !6
cons(!4,!4) = !4
cons(!4,!5) = !2
cons(!4,!6) = !4
cons(!5,!1) = !4
cons(!5,!2) = !4
cons(!5,!3) = !4
cons(!5,!4) = !4
cons(!5,!5) = !4
cons(!5,!6) = !4
cons(!6,!1) = !4
cons(!6,!2) = !4
cons(!6,!3) = !4
cons(!6,!4) = !4
cons(!6,!5) = !4
cons(!6,!6) = !4

head(!1) = !3
head(!2) = !4
head(!3) = !3
head(!4) = !1
head(!5) = !3
head(!6) = !4

list(!1) <=>true
list(!2)< = >true
list(!3) <=>true
list(!4)< = >false
list(!5) <=>true
list(!6)< = >true

nil = !3

ps = !1

qs = !2

rev(!1) = !2
rev(!2) = !4
rev(!3) = !3
rev(!4) = !4
rev(!5) = !5
rev(!6) = !6

sK13_axiom_UnitX(!1) = !5
```

```
sK13_axiom_UnitX(!2) = !6
sK13_axiom_UnitX(!3) = !5
sK13_axiom_UnitX(!4) = !4
sK13_axiom_UnitX(!5) = !5
sK13_axiom_UnitX(!6) = !6

sub(!1,!1) <=>$false
sub(!1,!2) <=>$false
sub(!1,!3) <=>$false
sub(!1,!4) <=>$false
sub(!1,!5) <=>$false
sub(!1,!6) <=>$false
sub(!2,!1) <=>$false
sub(!2,!2) <=>$false
sub(!2,!3) <=>$false
sub(!2,!4) <=>$false
sub(!2,!5) <=>$false
sub(!2,!6) <=>$false
sub(!3,!1) <=>$true
sub(!3,!2) <=>$true
sub(!3,!3) <=>$false
sub(!3,!4) <=>$false
sub(!3,!5) <=>$true
sub(!3,!6) <=>$true
sub(!4,!1) <=>$false
sub(!4,!2) <=>$false
sub(!4,!3) <=>$false
sub(!4,!4) <=>$false
sub(!4,!5) <=>$false
sub(!4,!6) <=>$false
sub(!5,!1) <=>$false
sub(!5,!2) <=>$true
sub(!5,!3) <=>$false
sub(!5,!4) <=>$false
sub(!5,!5) <=>$false
sub(!5,!6) <=>$false
sub(!6,!1) <=>$true
sub(!6,!2) <=>$false
sub(!6,!3) <=>$false
sub(!6,!4) <=>$false
sub(!6,!5) <=>$false
sub(!6,!6) <=>$false
```

```
tail(!1) = !6
tail(!2) = !5
tail(!3) = !4
tail(!4) = !4
tail(!5) = !3
tail(!6) = !3
+++ END MODEL
```

A.3 C list with a domain size of five

A.3.1 Problem description

```
cnf(tc,axiom, next_tc(X,X)).

cnf(tc,axiom, next(X)!=Y |next_tc(X,Y)).

cnf(tc,axiom,  next_tc(X,Y) | next_tc(Y,Z) |next_tc(X,Z)).

cnf(tc,axiom,  next_tc(X,Y) |X=Y |next(X)=next_st(X,Y)).

cnf(tc,axiom,  next_tc(X,Y) |X=Y |next_tc(next_st(X,Y),Y)).

cnf(tc,axiom,  next_tc(X,Y) |X=Y |next_cl(X,next_st(X,Y),Y)).

cnf(tc,axiom,  next_cl(X,X,V)).

cnf(tc,axiom,  next_cl(X,Y,V) | next_cl(Y,Z,V) |next_cl(X,Z,V)).

cnf(axiom,axiom, next(null)=null).

cnf(axiom,axiom, inv(null,null)).

cnf(axiom,axiom, Start=null |End=null | next_tc(Start,End) |next(End)!=null
|inv(Start,End)).

cnf(axiom,axiom,  inv(Start,End) |Start!=null |End=null).

cnf(axiom,axiom,  inv(Start,End) |Start=null |End!=null).

cnf(axiom,axiom,  inv(Start,End) |Start=null |next_tc(Start,End)).

cnf(axiom,axiom,  inv(Start,End) |Start=null |next(End)=null).

cnf(axiom,axiom, inv(start,end)).

cnf(axiom,axiom, next(start)!=start).

cnf(axiom,axiom, next(next(start))!=next(start)).

cnf(axiom,axiom, next(next(next(start)))!=next(next(start))).
cnf(axiom,axiom, next(next(next(next(start))))!=next(next(next(start)))).
```

A.3.2 Solution model

```
+++ BEGIN MODEL
% domain size is 5


end = !2


inv(!1,!1) <=>$false
inv(!1,!2) <=>$true
inv(!1,!3) <=>$false
inv(!1,!4) <=>$false
inv(!1,!5) <=>$false
inv(!2,!1) <=>$false
inv(!2,!2) <=>$true
inv(!2,!3) <=>$false
inv(!2,!4) <=>$false
inv(!2,!5) <=>$false
inv(!3,!1) <=>$false
inv(!3,!2) <=>$false
inv(!3,!3) <=>$true
inv(!3,!4) <=>$false
inv(!3,!5) <=>$false
inv(!4,!1) <=>$false
inv(!4,!2) <=>$true
inv(!4,!3) <=>$false
inv(!4,!4) <=>$false
inv(!4,!5) <=>$false
inv(!5,!1) <=>$false
inv(!5,!2) <=>$true
inv(!5,!3) <=>$false
inv(!5,!4) <=>$false
inv(!5,!5) <=>$false


next(!1) = !4
next(!2) = !3
next(!3) = !3
next(!4) = !5
next(!5) = !2
```

```
next_cl(!1,!1,!1) <=>$false
next_cl(!1,!1,!2) <=>$false
next_cl(!1,!1,!3) <=>$false
next_cl(!1,!1,!4) <=>$false
next_cl(!1,!1,!5) <=>$false
next_cl(!1,!2,!1) <=>$false
next_cl(!1,!2,!2) <=>$true
next_cl(!1,!2,!3) <=>$true
next_cl(!1,!2,!4) <=>$false
next_cl(!1,!2,!5) <=>$false
next_cl(!1,!3,!1) <=>$false
next_cl(!1,!3,!2) <=>$false
next_cl(!1,!3,!3) <=>$true
next_cl(!1,!3,!4) <=>$false
next_cl(!1,!3,!5) <=>$false
next_cl(!1,!4,!1) <=>$false
next_cl(!1,!4,!2) <=>$true
next_cl(!1,!4,!3) <=>$true
next_cl(!1,!4,!4) <=>$true
next_cl(!1,!4,!5) <=>$true
next_cl(!1,!5,!1) <=>$false
next_cl(!1,!5,!2) <=>$true
next_cl(!1,!5,!3) <=>$true
next_cl(!1,!5,!4) <=>$false
next_cl(!1,!5,!5) <=>$true
next_cl(!2,!1,!1) <=>$false
next_cl(!2,!1,!2) <=>$false
next_cl(!2,!1,!3) <=>$false
next_cl(!2,!1,!4) <=>$false
next_cl(!2,!1,!5) <=>$false
next_cl(!2,!2,!1) <=>$false
next_cl(!2,!2,!2) <=>$false
next_cl(!2,!2,!3) <=>$false
next_cl(!2,!2,!4) <=>$false
next_cl(!2,!2,!5) <=>$false
next_cl(!2,!3,!1) <=>$false
next_cl(!2,!3,!2) <=>$false
next_cl(!2,!3,!3) <=>$true
next_cl(!2,!3,!4) <=>$false
next_cl(!2,!3,!5) <=>$false
next_cl(!2,!4,!1) <=>$false
next_cl(!2,!4,!2) <=>$false
next_cl(!2,!4,!3) <=>$false
next_cl(!2,!4,!4) <=>$false
next_cl(!2,!4,!5) <=>$false
next_cl(!2,!5,!1) <=>$false
next_cl(!2,!5,!2) <=>$false
next_cl(!2,!5,!3) <=>$false
next_cl(!2,!5,!4) <=>$false
next_cl(!2,!5,!5) <=>$false
next_cl(!3,!1,!1) <=>$false
next_cl(!3,!1,!2) <=>$false
next_cl(!3,!1,!3) <=>$false
next_cl(!3,!1,!4) <=>$false
next_cl(!3,!1,!5) <=>$false
next_cl(!3,!2,!1) <=>$false
next_cl(!3,!2,!2) <=>$false
next_cl(!3,!2,!3) <=>$false
next_cl(!3,!2,!4) <=>$false
next_cl(!3,!2,!5) <=>$false
next_cl(!3,!3,!1) <=>$false
next_cl(!3,!3,!2) <=>$false
next_cl(!3,!3,!3) <=>$false
next_cl(!3,!3,!4) <=>$false
next_cl(!3,!3,!5) <=>$false
next_cl(!3,!4,!1) <=>$false
next_cl(!3,!4,!2) <=>$false
next_cl(!3,!4,!3) <=>$false
next_cl(!3,!4,!4) <=>$false
next_cl(!3,!4,!5) <=>$false
next_cl(!3,!5,!1) <=>$false
next_cl(!3,!5,!2) <=>$false
next_cl(!3,!5,!3) <=>$false
next_cl(!3,!5,!4) <=>$false
next_cl(!3,!5,!5) <=>$false
next_cl(!4,!1,!1) <=>$false
next_cl(!4,!1,!2) <=>$false
next_cl(!4,!1,!3) <=>$false
next_cl(!4,!1,!4) <=>$false
next_cl(!4,!1,!5) <=>$false
next_cl(!4,!2,!1) <=>$false
next_cl(!4,!2,!2) <=>$true
next_cl(!4,!2,!3) <=>$true
next_cl(!4,!2,!4) <=>$false
next_cl(!4,!2,!5) <=>$false
next_cl(!4,!3,!1) <=>$false
next_cl(!4,!3,!2) <=>$false
next_cl(!4,!3,!3) <=>$true
next_cl(!4,!3,!4) <=>$false
next_cl(!4,!3,!5) <=>$false
next_cl(!4,!4,!1) <=>$false
next_cl(!4,!4,!2) <=>$false
next_cl(!4,!4,!3) <=>$false
next_cl(!4,!4,!4) <=>$false
next_cl(!4,!4,!5) <=>$false
next_cl(!4,!5,!1) <=>$false
next_cl(!4,!5,!2) <=>$true
next_cl(!4,!5,!3) <=>$true
next_cl(!4,!5,!4) <=>$false
next_cl(!4,!5,!5) <=>$true
next_cl(!5,!1,!1) <=>$false
next_cl(!5,!1,!2) <=>$false
next_cl(!5,!1,!3) <=>$false
next_cl(!5,!1,!4) <=>$false
next_cl(!5,!1,!5) <=>$false
next_cl(!5,!2,!1) <=>$false
next_cl(!5,!2,!2) <=>$true
next_cl(!5,!2,!3) <=>$true
next_cl(!5,!2,!4) <=>$false
next_cl(!5,!2,!5) <=>$false
next_cl(!5,!3,!1) <=>$false
next_cl(!5,!3,!2) <=>$false
next_cl(!5,!3,!3) <=>$true
next_cl(!5,!3,!4) <=>$false
next_cl(!5,!3,!5) <=>$false
next_cl(!5,!4,!1) <=>$false
next_cl(!5,!4,!2) <=>$false
next_cl(!5,!4,!3) <=>$false
next_cl(!5,!4,!4) <=>$false
next_cl(!5,!4,!5) <=>$false
next_cl(!5,!5,!1) <=>$false
next_cl(!5,!5,!2) <=>$false
next_cl(!5,!5,!3) <=>$false
next_cl(!5,!5,!4) <=>$false
next_cl(!5,!5,!5) <=>$false

next_st(!1,!1) = !1
next_st(!1,!2) = !4
next_st(!1,!3) = !4
next_st(!1,!4) = !4
next_st(!1,!5) = !4
next_st(!2,!1) = !1
next_st(!2,!2) = !5
next_st(!2,!3) = !3
next_st(!2,!4) = !5
next_st(!2,!5) = !5
next_st(!3,!1) = !5
next_st(!3,!2) = !5
next_st(!3,!3) = !1
next_st(!3,!4) = !5
next_st(!3,!5) = !5
next_st(!4,!1) = !5
next_st(!4,!2) = !5
next_st(!4,!3) = !5
next_st(!4,!4) = !5
next_st(!4,!5) = !5
next_st(!5,!1) = !5
next_st(!5,!2) = !2
next_st(!5,!3) = !2
next_st(!5,!4) = !5
next_st(!5,!5) = !5

next_tc(!1,!1) <=>$true
next_tc(!1,!2) <=>$true
next_tc(!1,!3) <=>$true
next_tc(!1,!4) <=>$true
next_tc(!1,!5) <=>$true
next_tc(!2,!1) <=>$false
next_tc(!2,!2) <=>$true
next_tc(!2,!3) <=>$true
next_tc(!2,!4) <=>$false
next_tc(!2,!5) <=>$false
next_tc(!3,!1) <=>$false
next_tc(!3,!2) <=>$false
next_tc(!3,!3) <=>$true
next_tc(!3,!4) <=>$false
next_tc(!3,!5) <=>$false
next_tc(!4,!1) <=>$false
next_tc(!4,!2) <=>$true
next_tc(!4,!3) <=>$true
next_tc(!4,!4) <=>$true
next_tc(!4,!5) <=>$true
next_tc(!5,!1) <=>$false
next_tc(!5,!2) <=>$true
next_tc(!5,!3) <=>$true
next_tc(!5,!4) <=>$false
next_tc(!5,!5) <=>$true

null = !3

start = !1
+++ END MODEL
+++ RESULT: Satisfiable
```

A.4 Collinearity Problem

A.4.1 Problem description

```
cnf(two_points_collinear,axiom, ( collinear(X,X,Y) )).

cnf(rotate_collinear,axiom, ( collinear(Y,X,Z) |  collinear(X,Y,Z) )).

cnf(swap_collinear,axiom, ( collinear(Z,X,Y) |  collinear(X,Y,Z) )).

cnf(transitivity_collinear,axiom, ( collinear(X,Y1,Z) |X = Y2 |Y1 = Y2 |Y2
= Z |  collinear(Y1,Y2,Z) |  collinear(X,Y1,Y2) )).

cnf(third_point_collinear,hypothesis, ( collinear(X,Y,third(X,Y)) )).

cnf(third_point_different_1a,hypothesis, ( X != third(X,Y) )).

cnf(third_point_different_1b,hypothesis, ( Y != third(X,Y) )).

cnf(conjecture,negated_conjecture, (   collinear(p1,p2,p3) )).
```

A.4.2 Solution model

```
+++ BEGIN MODEL
% domain size is 7

collinear(!1,!1,!1) <=>$true
collinear(!1,!1,!2) <=>$true
collinear(!1,!1,!3) <=>$true
collinear(!1,!1,!4) <=>$true
collinear(!1,!1,!5) <=>$true
collinear(!1,!1,!6) <=>$true
collinear(!1,!1,!7) <=>$true
collinear(!1,!2,!1) <=>$true
collinear(!1,!2,!2) <=>$true
collinear(!1,!2,!3) <=>$false
collinear(!1,!2,!4) <=>$false
collinear(!1,!2,!5) <=>$false
collinear(!1,!2,!6) <=>$true
collinear(!1,!2,!7) <=>$false
collinear(!1,!3,!1) <=>$true
collinear(!1,!3,!2) <=>$false
collinear(!1,!3,!3) <=>$true
collinear(!1,!3,!4) <=>$false
collinear(!1,!3,!5) <=>$false
collinear(!1,!3,!6) <=>$false
collinear(!1,!3,!7) <=>$true
collinear(!1,!4,!1) <=>$true
collinear(!1,!4,!2) <=>$false
collinear(!1,!4,!3) <=>$false
collinear(!1,!4,!4) <=>$true
collinear(!1,!4,!5) <=>$true
collinear(!1,!4,!6) <=>$false
collinear(!1,!4,!7) <=>$false
collinear(!1,!5,!1) <=>$true
collinear(!1,!5,!2) <=>$false
collinear(!1,!5,!3) <=>$false
collinear(!1,!5,!4) <=>$true
collinear(!1,!5,!5) <=>$true
collinear(!1,!5,!6) <=>$false
collinear(!1,!5,!7) <=>$false
collinear(!1,!6,!1) <=>$true
collinear(!1,!6,!2) <=>$true
collinear(!1,!6,!3) <=>$false
collinear(!1,!6,!4) <=>$false
collinear(!1,!6,!5) <=>$false
collinear(!1,!6,!6) <=>$true
collinear(!1,!6,!7) <=>$false
collinear(!1,!7,!1) <=>$true
collinear(!1,!7,!2) <=>$false
collinear(!1,!7,!3) <=>$true
collinear(!1,!7,!4) <=>$false
collinear(!1,!7,!5) <=>$false
collinear(!1,!7,!6) <=>$false
collinear(!1,!7,!7) <=>$true
collinear(!2,!1,!1) <=>$true
collinear(!2,!1,!2) <=>$true
collinear(!2,!1,!3) <=>$false
collinear(!2,!1,!4) <=>$false
collinear(!2,!1,!5) <=>$false
collinear(!2,!1,!6) <=>$true
collinear(!2,!1,!7) <=>$false
collinear(!2,!2,!1) <=>$true
collinear(!2,!2,!2) <=>$true
collinear(!2,!2,!3) <=>$true
collinear(!2,!2,!4) <=>$true
collinear(!2,!2,!5) <=>$true
collinear(!2,!2,!6) <=>$true
collinear(!2,!2,!7) <=>$true
collinear(!2,!3,!1) <=>$false
collinear(!2,!3,!2) <=>$true
collinear(!2,!3,!3) <=>$true
collinear(!2,!3,!4) <=>$true
collinear(!2,!3,!5) <=>$false
collinear(!2,!3,!6) <=>$false
collinear(!2,!3,!7) <=>$false
collinear(!2,!4,!1) <=>$false
collinear(!2,!4,!2) <=>$true
collinear(!2,!4,!3) <=>$true
collinear(!2,!4,!4) <=>$true
collinear(!2,!4,!5) <=>$false
collinear(!2,!4,!6) <=>$false
collinear(!2,!4,!7) <=>$false
collinear(!2,!5,!1) <=>$false
collinear(!2,!5,!2) <=>$true
collinear(!2,!5,!3) <=>$false
collinear(!2,!5,!4) <=>$false
collinear(!2,!5,!5) <=>$true
collinear(!2,!5,!6) <=>$false
collinear(!2,!5,!7) <=>$true
collinear(!2,!6,!1) <=>$true
collinear(!2,!6,!2) <=>$true
collinear(!2,!6,!3) <=>$false
collinear(!2,!6,!4) <=>$false
collinear(!2,!6,!5) <=>$false
collinear(!2,!6,!6) <=>$true
collinear(!2,!6,!7) <=>$false
collinear(!2,!7,!1) <=>$false
collinear(!2,!7,!2) <=>$true
collinear(!2,!7,!3) <=>$false
collinear(!2,!7,!4) <=>$false
collinear(!2,!7,!5) <=>$true
collinear(!2,!7,!6) <=>$false
collinear(!2,!7,!7) <=>$true
collinear(!3,!1,!1) <=>$true
collinear(!3,!1,!2) <=>$false
collinear(!3,!1,!3) <=>$true
collinear(!3,!1,!4) <=>$false
collinear(!3,!1,!5) <=>$false
collinear(!3,!1,!6) <=>$false
collinear(!3,!1,!7) <=>$true
collinear(!3,!2,!1) <=>$false
collinear(!3,!2,!2) <=>$true
collinear(!3,!2,!3) <=>$true
collinear(!3,!2,!4) <=>$true
collinear(!3,!2,!5) <=>$false
collinear(!3,!2,!6) <=>$false
collinear(!3,!2,!7) <=>$false
collinear(!3,!3,!1) <=>$true
collinear(!3,!3,!2) <=>$true
collinear(!3,!3,!3) <=>$true
collinear(!3,!3,!4) <=>$true
collinear(!3,!3,!5) <=>$true
collinear(!3,!3,!6) <=>$true
collinear(!3,!3,!7) <=>$true
collinear(!3,!4,!1) <=>$false
collinear(!3,!4,!2) <=>$true
collinear(!3,!4,!3) <=>$true
collinear(!3,!4,!4) <=>$true
collinear(!3,!4,!5) <=>$false
collinear(!3,!4,!6) <=>$false
collinear(!3,!4,!7) <=>$false
collinear(!3,!5,!1) <=>$false
collinear(!3,!5,!2) <=>$false
collinear(!3,!5,!3) <=>$true
collinear(!3,!5,!4) <=>$false
collinear(!3,!5,!5) <=>$true
collinear(!3,!5,!6) <=>$true
collinear(!3,!5,!7) <=>$false
collinear(!3,!6,!1) <=>$false
collinear(!3,!6,!2) <=>$false
collinear(!3,!6,!3) <=>$true
collinear(!3,!6,!4) <=>$false
collinear(!3,!6,!5) <=>$true
collinear(!3,!6,!6) <=>$true
collinear(!3,!6,!7) <=>$false
collinear(!3,!7,!1) <=>$true
collinear(!3,!7,!2) <=>$false
collinear(!3,!7,!3) <=>$true
collinear(!3,!7,!4) <=>$false
collinear(!3,!7,!5) <=>$false
collinear(!3,!7,!6) <=>$false
collinear(!3,!7,!7) <=>$true
collinear(!4,!1,!1) <=>$true
collinear(!4,!1,!2) <=>$false
collinear(!4,!1,!3) <=>$false
collinear(!4,!1,!4) <=>$true
collinear(!4,!1,!5) <=>$true
collinear(!4,!1,!6) <=>$false
collinear(!4,!1,!7) <=>$false
collinear(!4,!2,!1) <=>$false
collinear(!4,!2,!2) <=>$true
collinear(!4,!2,!3) <=>$true
collinear(!4,!2,!4) <=>$true
collinear(!4,!2,!5) <=>$false
collinear(!4,!2,!6) <=>$false
collinear(!4,!2,!7) <=>$false
collinear(!4,!3,!1) <=>$false
collinear(!4,!3,!2) <=>$true
collinear(!4,!3,!3) <=>$true
collinear(!4,!3,!4) <=>$true
collinear(!4,!3,!5) <=>$false
collinear(!4,!3,!6) <=>$false
collinear(!4,!3,!7) <=>$false
collinear(!4,!4,!1) <=>$true
collinear(!4,!4,!2) <=>$true
collinear(!4,!4,!3) <=>$true
collinear(!4,!4,!4) <=>$true
collinear(!4,!4,!5) <=>$true
collinear(!4,!4,!6) <=>$true
collinear(!4,!4,!7) <=>$true
collinear(!4,!5,!1) <=>$true
collinear(!4,!5,!2) <=>$false
collinear(!4,!5,!3) <=>$false
collinear(!4,!5,!4) <=>$true
collinear(!4,!5,!5) <=>$true
collinear(!4,!5,!6) <=>$false
collinear(!4,!5,!7) <=>$false
collinear(!4,!6,!1) <=>$false
collinear(!4,!6,!2) <=>$false
collinear(!4,!6,!3) <=>$false
collinear(!4,!6,!4) <=>$true
collinear(!4,!6,!5) <=>$false
collinear(!4,!6,!6) <=>$true
collinear(!4,!6,!7) <=>$true
collinear(!4,!7,!1) <=>$false
collinear(!4,!7,!2) <=>$false
collinear(!4,!7,!3) <=>$false
collinear(!4,!7,!4) <=>$true
collinear(!4,!7,!5) <=>$false
collinear(!4,!7,!6) <=>$true
collinear(!4,!7,!7) <=>$true
collinear(!5,!1,!1) <=>$true
collinear(!5,!1,!2) <=>$false
collinear(!5,!1,!3) <=>$false
collinear(!5,!1,!4) <=>$true
collinear(!5,!1,!5) <=>$true
collinear(!5,!1,!6) <=>$false
collinear(!5,!1,!7) <=>$false
collinear(!5,!2,!1) <=>$false
collinear(!5,!2,!2) <=>$true
collinear(!5,!2,!3) <=>$false
collinear(!5,!2,!4) <=>$false
collinear(!5,!2,!5) <=>$true
collinear(!5,!2,!6) <=>$false
collinear(!5,!2,!7) <=>$true
collinear(!5,!3,!1) <=>$false
collinear(!5,!3,!2) <=>$false
collinear(!5,!3,!3) <=>$true
collinear(!5,!3,!4) <=>$false
collinear(!5,!3,!5) <=>$true
collinear(!5,!3,!6) <=>$true
collinear(!5,!3,!7) <=>$false
collinear(!5,!4,!1) <=>$true
collinear(!5,!4,!2) <=>$false
collinear(!5,!4,!3) <=>$false
collinear(!5,!4,!4) <=>$true
collinear(!5,!4,!5) <=>$true
collinear(!5,!4,!6) <=>$false
collinear(!5,!4,!7) <=>$false
collinear(!5,!5,!1) <=>$true
collinear(!5,!5,!2) <=>$true
collinear(!5,!5,!3) <=>$true
collinear(!5,!5,!4) <=>$true
collinear(!5,!5,!5) <=>$true
collinear(!5,!5,!6) <=>$true
collinear(!5,!5,!7) <=>$true
collinear(!5,!6,!1) <=>$false
collinear(!5,!6,!2) <=>$false
collinear(!5,!6,!3) <=>$true
collinear(!5,!6,!4) <=>$false
collinear(!5,!6,!5) <=>$true
collinear(!5,!6,!6) <=>$true
collinear(!5,!6,!7) <=>$false
collinear(!5,!7,!1) <=>$false
collinear(!5,!7,!2) <=>$true
collinear(!5,!7,!3) <=>$false
collinear(!5,!7,!4) <=>$false
collinear(!5,!7,!5) <=>$true
collinear(!5,!7,!6) <=>$false
collinear(!5,!7,!7) <=>$true
collinear(!6,!1,!1) <=>$true
collinear(!6,!1,!2) <=>$true
collinear(!6,!1,!3) <=>$false
collinear(!6,!1,!4) <=>$false
collinear(!6,!1,!5) <=>$false
collinear(!6,!1,!6) <=>$true
collinear(!6,!1,!7) <=>$false
collinear(!6,!2,!1) <=>$true
collinear(!6,!2,!2) <=>$true
collinear(!6,!2,!3) <=>$false
collinear(!6,!2,!4) <=>$false
collinear(!6,!2,!5) <=>$false
collinear(!6,!2,!6) <=>$true
collinear(!6,!2,!7) <=>$false
collinear(!6,!3,!1) <=>$false
collinear(!6,!3,!2) <=>$false
collinear(!6,!3,!3) <=>$true
collinear(!6,!3,!4) <=>$false
collinear(!6,!3,!5) <=>$true
collinear(!6,!3,!6) <=>$true
collinear(!6,!3,!7) <=>$false
collinear(!6,!4,!1) <=>$false
collinear(!6,!4,!2) <=>$false
collinear(!6,!4,!3) <=>$false
collinear(!6,!4,!4) <=>$true
collinear(!6,!4,!5) <=>$false
collinear(!6,!4,!6) <=>$true
collinear(!6,!4,!7) <=>$true
collinear(!6,!5,!1) <=>$false
collinear(!6,!5,!2) <=>$false
collinear(!6,!5,!3) <=>$true
collinear(!6,!5,!4) <=>$false
collinear(!6,!5,!5) <=>$true
collinear(!6,!5,!6) <=>$true
collinear(!6,!5,!7) <=>$false
collinear(!6,!6,!1) <=>$true
collinear(!6,!6,!2) <=>$true
collinear(!6,!6,!3) <=>$true
collinear(!6,!6,!4) <=>$true
collinear(!6,!6,!5) <=>$true
collinear(!6,!6,!6) <=>$true
collinear(!6,!6,!7) <=>$true
collinear(!6,!7,!1) <=>$false
collinear(!6,!7,!2) <=>$false
collinear(!6,!7,!3) <=>$false
collinear(!6,!7,!4) <=>$true
collinear(!6,!7,!5) <=>$false
collinear(!6,!7,!6) <=>$true
collinear(!6,!7,!7) <=>$true
collinear(!7,!1,!1) <=>$true
collinear(!7,!1,!2) <=>$false
collinear(!7,!1,!3) <=>$true
collinear(!7,!1,!4) <=>$false
collinear(!7,!1,!5) <=>$false
collinear(!7,!1,!6) <=>$false
collinear(!7,!1,!7) <=>$true
collinear(!7,!2,!1) <=>$false
collinear(!7,!2,!2) <=>$true
collinear(!7,!2,!3) <=>$false
collinear(!7,!2,!4) <=>$false
collinear(!7,!2,!5) <=>$true
collinear(!7,!2,!6) <=>$false
collinear(!7,!2,!7) <=>$true
collinear(!7,!3,!1) <=>$true
collinear(!7,!3,!2) <=>$false
collinear(!7,!3,!3) <=>$true
collinear(!7,!3,!4) <=>$false
collinear(!7,!3,!5) <=>$false
collinear(!7,!3,!6) <=>$false
collinear(!7,!3,!7) <=>$true
collinear(!7,!4,!1) <=>$false
collinear(!7,!4,!2) <=>$false
collinear(!7,!4,!3) <=>$false
collinear(!7,!4,!4) <=>$true
collinear(!7,!4,!5) <=>$false
collinear(!7,!4,!6) <=>$true
collinear(!7,!4,!7) <=>$true
collinear(!7,!5,!1) <=>$false
collinear(!7,!5,!2) <=>$true
collinear(!7,!5,!3) <=>$false
collinear(!7,!5,!4) <=>$false
collinear(!7,!5,!5) <=>$true
collinear(!7,!5,!6) <=>$false
collinear(!7,!5,!7) <=>$true
collinear(!7,!6,!1) <=>$false
collinear(!7,!6,!2) <=>$false
collinear(!7,!6,!3) <=>$false
collinear(!7,!6,!4) <=>$true
collinear(!7,!6,!5) <=>$false
collinear(!7,!6,!6) <=>$true
collinear(!7,!6,!7) <=>$true
collinear(!7,!7,!1) <=>$true
collinear(!7,!7,!2) <=>$true
collinear(!7,!7,!3) <=>$true
collinear(!7,!7,!4) <=>$true
collinear(!7,!7,!5) <=>$true
collinear(!7,!7,!6) <=>$true
collinear(!7,!7,!7) <=>$true

p1 = !1

p2 = !2

p3 = !3

third(!1,!1) = !3
third(!1,!2) = !6
third(!1,!3) = !7
third(!1,!4) = !5
third(!1,!5) = !4
third(!1,!6) = !2
third(!1,!7) = !3
third(!2,!1) = !6
third(!2,!2) = !1
third(!2,!3) = !4
third(!2,!4) = !3
third(!2,!5) = !7
third(!2,!6) = !1
third(!2,!7) = !5
third(!3,!1) = !7
third(!3,!2) = !4
third(!3,!3) = !1
third(!3,!4) = !2
third(!3,!5) = !6
third(!3,!6) = !5
third(!3,!7) = !1
third(!4,!1) = !5
third(!4,!2) = !3
third(!4,!3) = !2
third(!4,!4) = !3
third(!4,!5) = !1
third(!4,!6) = !7
third(!4,!7) = !6
third(!5,!1) = !4
third(!5,!2) = !7
third(!5,!3) = !6
third(!5,!4) = !1
third(!5,!5) = !2
third(!5,!6) = !3
third(!5,!7) = !2
third(!6,!1) = !2
third(!6,!2) = !1
third(!6,!3) = !5
third(!6,!4) = !7
third(!6,!5) = !3
third(!6,!6) = !5
third(!6,!7) = !4
third(!7,!1) = !3
third(!7,!2) = !5
third(!7,!3) = !1
third(!7,!4) = !6
third(!7,!5) = !2
third(!7,!6) = !4
third(!7,!7) = !6
+++ END MODEL
+++ RESULT: Satisfiable
```