Towards A Formal Verification of Courcelle’s Theorem

by

Emily Black

A thesis submitted to the
faculty of Wesleyan University
in partial fulfillment of the requirements for the
Degree of Bachelor of Arts
with Departmental Honors in Mathematics and Computer Science

Middletown, Connecticut        April 2017
Acknowledgements

Dan: Thank you for introducing me to computer science, inspiring me to pursue it, and helping me stay afloat in it. Thank you for going along with my request to work on graph algorithms with you, a programming languages theorist. Thank you for long meetings and frequent emails. Thanks for cutting me a lot of slack in my darkest hours. My path would have been quite different without you, and I’m so grateful that it is the way it is.

Mom and Dad: Thanks for always, always being there.

Thienthanh: Thank you for merging into one consciousness with me: eating together, working together, meditating together, moping together, inspiring hope together for the last two weeks. I don’t know if I could have finished without you by my side day in and day out.

Max and Joomy: Thanks for all the venting, commisterating, pep-talking, storytelling, and knowledge-sharing that happened in our office.

Professor Krizanc: Thank you very much for guiding me towards the topic of my thesis, for spending many hours helping me over the summer, and for giving me very helpful advice about graduate school and life in general.

Professors Krizanc and Manfredi: Thank you so much for agreeing to read my thesis!

Leyla: Thank you for feeding me and buying my champagne.
Abstract

The goal of this project is to complete a formal verification of Courcelle’s Theorem in Agda. On a high level, Courcelle’s theorem states that if you ask certain kinds of questions about certain kinds of graphs, you can get a certain kind of linear time answer. This theorem is important because it applies to many NP-hard problems, such as vertex cover and dominating set, and thus gives a quasi-linear algorithm for deciding certain questions (monadic second order logic questions) on a certain class of graphs (bounded treewidth graphs). Verification via proof assistants is a method of ensuring that either a mathematical proof or a program is correct, in a way that removes much of the possibility for human error. At this stage, we have completed our translation of the definitions given in Courcelle’s Theorem: A Game Theoretic Approach by Kneis et al. into Agda, which is the proof that we have chosen to verify. This involved reformulating several complex mathematical concepts into machine-checkable definitions. Specifically, we define symbols, signatures, structures, expansions of structures, restrictions of structures, tree decompositions, MSO formulas, isomorphisms of structures, model checking games, extended model checking games, equivalence between game positions and subgames, and finally a notion of a reduced game. The bulk of this thesis document is a detailed explanation of the fact that the mathematical definitions we formalize correspond to the way we encode them. This is in an effort to convince the reader that, once we do finish the complete proof, it will be correct since our definitions are correct, for the only way a proof assistant can be incorrect is by
the author of the program unintentionally encoding the wrong statement of the theorem. Additionally, we have defined the types of the functions corresponding to the lemmas in the paper that prove Courcelle’s Theorem, however we have yet to implement them. We also have yet to prove the fact that the algorithm is fixed-parameter tractable linear; this is next on our agenda after completing the proof of the algorithm itself. After this work is complete, we hope to extend our verification of Courcelle’s Theorem to some of its applications.
## Contents

Abstract iii  
Table of Contents v  

Chapter 1. Introduction 1  
1. Courcelle’s Theorem 1  
1.1. What Kinds of Questions: Monadic Second Order Logic Questions 2  
1.2. What Kinds of Graphs: Graphs of Bounded Treewidth 4  
1.3. What Kind of Linear: Fixed Parameter Tractable Linear 9  
1.4. Verification and Agda 10  
1.5. Contributions and Future Work 12  
1.6. Road Map 13  

Chapter 2. Helpful Background and History 15  

Chapter 3. Outline of Kneis et al. Algorithm 19  

Chapter 4. Related Work and Applications 22  

Chapter 5. Definitions 23  
1. Structures 23  
1.1. Expansion 25  
2. Restrictions 30  
3. Tree Decompositions 32  
4. MSO Formulas 39
5. Relating Structures: Isomorphisms, Compatibility, Union 41
   5.1. Isomorphisms of Structures 41
   5.2. Compatibility 44
   5.3. Union of Structures 44
6. Games 44
   6.1. Model Checking Games 46
   6.2. EMC Games 52
7. Reduced Games 62
   7.1. Game Tree 62
   7.2. Position Equivalence 64
   7.3. Game Equivalence 66
   7.4. Reduced Games Definition 69

Chapter 6. Outline of Algorithm and Work in Progress 75
1. Proof of Algorithm 75
2. Agda Code and Proof 80

Chapter 7. Conclusion 87

Bibliography 89
CHAPTER 1

Introduction

Wouldn’t it be lovely if NP hard graph problems weren’t NP hard? Perhaps this was what Bruno Courcelle was thinking when he came up with his eponymous theorem. Courcelle’s Theorem is a very general algorithm which tries to wriggle out of the trap of NP-hardness. On a high level, the theorem states that if you ask certain kinds of questions about certain kinds of graphs, you can get a sort of linear time answer. The qualifications on the questions, graphs, and linear-ness of the answer are quite large; the theorem is unfortunately no magic trick.

In this thesis, we present our progress towards a formal verification of Courcelle’s Theorem. In order to properly introduce the work, we first explain the background necessary to understanding the statement of Courcelle’s Theorem: the kinds of questions we can ask, the kinds of graph we can ask questions about, and the kind of algorithm promised by the theorem. Following this, we delve into a quick description of verification with proof assistants, and discuss our verification tool, Agda. Then we will describe the contributions of this thesis, our work in progress, and our future directions. Finally, before embarking into the main part of the thesis, we provide a roadmap for this journey.

1. Courcelle’s Theorem

We begin our dive into the details of Courcelle’s Theorem: what kinds of questions can we ask, on what kinds of structures, and what sort of a linear algorithm do we get? A slightly more formal statement of the theorem is as
follows: Any problem that can be expressed in monadic second order logic can be decided in fixed-parameter tractable linear time on a structure of bounded treewidth.

1.1. What Kinds of Questions: Monadic Second Order Logic Questions. The kinds of questions we can ask are those expressible in Monadic Second Order Logic, or $MSO$. MSO is an extension of first order logic which, along with all of the capabilities of first order logic, introduces set variables, membership in these sets, and quantification over sets. MSO can also be seen as a restriction of second order logic where quantification is only allowed over unary relations, i.e. sets. However, no matter which way we look at it, we cannot quantify over relations that are not sets, i.e. that are not unary. For example, suppose we model a graph $G$ as a set of vertices $V$ and a binary relation $E(x,y)$, where $x$ and $y$ are vertices, to symbolize edges. Then suppose we wanted to see if there existed a binary relation $U(x,y)$ on $G$ such that

$$\exists U(x,y)(U(x,y) \leftrightarrow E(x,y) \land E(y,x))$$

In other words, are there bidirectional edges in $G$? We cannot ask this question because it quantifies over a binary relation, $U$. However note that if we instead model $G$ as just one set where set elements can represent vertices or edges, we can recast this formula in MSO as shown below. Here $ends(e,x,y)$ is a ternary relation where $e$ is a set element representing an edge, and $x,y$ are set elements represeting vertices, and $ends(e,x,y)$ represents the fact that $e$ is an edge between $x$ and $y$.

$$\exists P(e) \forall e(P(e) \leftrightarrow \exists x,y,e'(ends(e,x,y) \leftrightarrow ends(e',y,x)))$$

So including edges in our set, instead of casting them as a relation, expands the set of formulas that Courcelle’s Theorem applies to. This quick fix is taken from
Courcelle[5], and we implement it our code. Instead of expressing edges as a binary relation, we model graphs as simply a set, where the set elements can represent vertices or edges of the graph. This way, edges are simply set elements, which we can quantify over in MSO logic, and so we can express more formulas than by representing edges as a relation. Note that this requires having a relation *ends* that essentially labels edges with the vertices they connect. Additionally, in order to ensure that the two different types of set elements do not get confused in our code, we have set elements carry a bit of data with them, saying what type (node or edge) they are.

Even without quantification over non-unary relations or the quick fix mentioned above, MSO manages to express quite a few NP hard problems: *k*-colorability for fixed *k*, vertex cover, and dominating set, to name a few. Below are the MSO representations of Vertex Cover and Dominating Set. *R* is a set of vertices, in each formula, and *Edge(x, y)* denotes the fact that there is an edge between vertices *x* and *y*, i.e. the tuple (*x*, *y*) ∈ the relation *Edge*.

\[
\text{Vertex Cover}(G) = \exists R \forall x \forall y (\neg \text{Edge}(x, y) \lor x \in R \lor y \in R)
\]

\[
\text{Dominating Set}(G) = \exists R \forall x (x \in R \lor (\exists y (y \in R \land \text{Edge}(x, y))))
\]

MSO does have its limitations. There is no way to count in MSO, so while finding out that there is a vertex cover of a dominating set of a given graph is possible, there is no way to express the notion of a minimum vertex cover minimum dominating set[1]. This is a major limitation since this minimized form is the most common form in which these questions are asked. In order to remedy this, some groups have made extensions of Courcelle’s Theorem to include optimized versions of MSO-expressible problems. We will cover these improvements in the related works section.
1.2. What Kinds of Graphs: Graphs of Bounded Treewidth. The kinds of graphs that we can ask MSO questions about are graphs of bounded treewidth. Treewidth can be thought of as a measure of the extent to which a graph is "tree-like". If a graph has bounded treewidth, then all of its information can be encoded in a tree in a finite fashion. In order to formally understand treewidth, we first have to understand tree decompositions. Tree decompositions are defined as follows[27, 19]:

A tree decomposition of a graph $G = (V, E)$ is a pair $(T, X)$ where $T$ is a tree and $X$ is a family of sets indexed by the vertices of $T$, such that

1. $\bigcup X_i = V$
2. $\forall e \in E, \exists i$ such that the vertices incident on $e$ are present together $X_i$\(^1\)
3. If $i, j, k$ are nodes of $T$ such that $i$ is on a path between $j$ and $k$, then $(X_j \cap X_k) \subseteq X_i$.

The sets of vertices, $X_i$, are called bags. The width of a tree decomposition is the size of the largest bag $X_i$ minus one. The treewidth of a graph is the minimum width over all possible tree decompositions.

We can see that a tree decomposition of a graph is an encoding of a graph into a tree: it preserves all the information of the original graph within it. It has all of the vertices of the original graph by the first condition. It keeps track of all of the edges in the graph by the second condition. To see why the third condition is necessary, we need to give an overview of what the algorithm given by Courcelle’s Theorem looks like. The idea is that we check the input formula $\phi$ over only the part of the structure present in each bag $X_i$ of a tree decomposition, and merge that information in a way which will be explained later. In light of

\(^{1}$Note that this is not in MSO: this is the mathematical definition for a tree decomposition, which has nothing to do with MSO on its own.
this, to illustrate why condition three is needed, consider the two figures below. We have a graph $G$ that is a triangle, and a "tree decomposition" $T$ which satisfies the first and second, but not the third, condition of tree decompositions.

**Figure 1.** Our graph $G$

![Figure 1: Our graph $G$](image)

**Figure 2.** Our "tree decomposition" $T$

![Figure 2: Our "tree decomposition" $T$](image)

Suppose the formula we wanted to check on this graph was

$$\exists x, y, z (\text{Edge}(x, y) \land \text{Edge}(y, z) \land \text{Edge}(z, x))$$

i.e. that there is a triangle in $G$. See that there is no bag of $T$ that tells us to check the formula on $\{x, y, z\}$ all at once—so we never come across the information that $\{x, y, z\}$ form a triangle, and the algorithm would return that there is no triangle in $G$. However, if we enforce condition three, then we would have to change the root of $T$ to have bag $\{x, y, z\}$, so we would come across the information, and return the correct answer. So, we see that condition three is required.

We include an example of a proper tree decomposition. Consider the graph below, $G$. We can see it is certainly not a tree, as it has cycles. However, we want to encode $G$ as a tree, and preserve all of its information.

Here is a tree decomposition of $G$. 

5
In this diagram, the large ovals containing letters in them depict the nodes of the tree, and the letters depict the elements in the bag at that node. A quick inspection can convince the reader that the three conditions of a tree decomposition hold on this structure: all vertices of the original graph are present in the union of the bags of the tree decomposition. For every edge \((i, j)\) of the original graph, we can find at least one bag of this tree decomposition containing both \(i\) and \(j\). The root contains edges \((b, g)\) and \((d, g)\), the children of the root contain \((a, b)\) and \((a, d)\), and \((d, g)\) over again, and the three children of the second child
contain the data for the rest of the edges. Finally, for all \( i, j, k \), nodes of \( T \) such that \( i \) is on a path between \( j \) and \( k \), we see that \( (X_j \cap X_k) \subseteq X_i \). For example, consider the path between the nodes corresponding to bags \( \{b,d\} \) and \( \{c,d,f\} \). The intersection of these two bags is \( \{d\} \). Note that every node on the path between these two nodes has \( d \) as an element of its bag. This property holds throughout the tree decomposition pictured.

This tree decomposition is not necessarily optimal, and it is definitely not unique. For example, the nodes at the bottom with singleton bags \( \{a\} \) and \( \{b\} \) are not necessary: all of the node and edge information is already encoded in the tree decomposition without them. So, an alternate tree decomposition of the same graph would not include these nodes. The tree decomposition pictured has width two, since the size of the largest bag is three. It may have occurred to the
reader that every finite graph \( G \) has finite treewidth. To see this, note that one possible tree decomposition of a finite graph \( G \) is a tree of one node, with a bag \( X_i \) containing all vertices of \( G \) within it. This satisfies all requirements: \( X_i \) contains all vertices and edges, and the connectivity is held because there is only one node. This means that the treewidth of any graph cannot be greater than the number of vertices, or elements of the set, \(|G|\). However, as we will soon see, using this tree decomposition would render Courcelle’s Theorem useless.

Finding the treewidth of a graph is a technically an NP-hard problem, and the version of Courcelle’s Theorem that this work verifies assumes that a tree-decomposition of the graph in question is given as a part of the input. Bodlaender proved that finding the optimal tree decomposition of an input graph is strongly fixed-parameter tractable linear, i.e. linear in the sense that Courcelle’s theorem gives a linear algorithm, which we will explain below [2]. However, the algorithm originally presented by Bodleander is known to be extremely impractical; for one thing, the constants in the algorithm are prohibitive to implementation, and also the algorithm, in turn, relies upon having a non-optimal tree decomposition to start with. Thankfully, there are several good heuristics for computing treewidth, which are covered nicely in Chapter 13 in Downey and Fellows’ 2013 book Fundamentals of Parametrized Complexity[8] as well as in several survey papers by Bodleander[3]. These heuristic algorithms may be the most efficient method for finding a tree decomposition as an input to the algorithm, since there is really no need that the tree decomposition be entirely optimal. The width being slightly larger will just increase the time the algorithm takes. However, the algorithm verified in this paper pays no attention to this part of the process, though other proofs of the theorem have taken a different approach. We only mention this
problem here to discuss the feasibility of actually using the algorithm as a way to quickly decide MSO-expressible hard problems on graphs of bounded treewidth.

1.3. What Kind of Linear: Fixed Parameter Tractable Linear. Finally, we come to the condition on linear-ness: the algorithm we get for deciding an MSO question on a graph of bounded treewidth is fixed-parameter tractable linear, or FPT linear. FPT linear means linear in the size of the input, with a "constant" that is a function of a fixed parameter, or several fixed parameters. In this case, we have an algorithm that is linear in the size of the input graph $G$, and our parameters are the encoding of the monadic second order logic sentence $\phi$ and the treewidth $t$ of $G$. So, the algorithm described by Courcelle’s theorem takes time

$$f(t, (|\phi|)) \cdot (|G|)$$

for some function $f$. Unfortunately, in the case of Courcelle’s Theorem, this function $f$ is very large. In fact, unless $P=NP$, the function cannot be upper bounded by an iterated exponential of height bounded by $\phi$ and $t$\cite{19, 6, 9}.

Bringing these definitions together, we can now understand the formal statement of Courcelle’s Theorem\cite{19, 6, 9}:

**Theorem.** Courcelle’s Theorem\cite{19, 6, 9}

Let $P$ be an MSO problem and $\omega$ be a positive integer. There is an algorithm $A$ and a function $f : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ such that for every graph $G = (V, E)$ of order $n := |V|$ and treewidth at most $\omega$, $A$ solves $P$ on input $G$ in time $f(|\phi|, \omega) \cdot n$, where $\phi$ is the MSO formula defining $P$ and $|\phi|$ is its length. Furthermore, unless $P = NP$, the function $f$ cannot be upper bounded by an iterated exponential of bounded height in terms of $\phi$ and $\omega$. 9
Upon reading about this ghastly constant, it may seem that the possibility of practical usage of Courcelle’s Theorem is quite small. However, not all hope is lost. There is work being done on making Courcelle’s theorem usable, as will be discussed in the related work section.

1.4. Verification and Agda. The reader might wonder why we go so far into detail on a theorem that has already been proven. As mentioned earlier, this thesis is a progress towards a formal verification of Courcelle’s Theorem. We perform the verification with a proof assistant, Agda\cite{26}. Verification via proof assistants is a method of ensuring that either a mathematical proof or a program is correct, in a way that removes much of the possibility for human error. Verification is currently used in a few main situations. The first is when there is a mathematical proof that is either too long for a human to reliably complete or a completed proof that is not well accepted by the community, for example the Jordan Curve theorem, or the Four Color theorem, which have both been verified by proof assistants\cite{21, 12}. Another area of use is for programs run in high-stakes situations, such as medical software, software used by NASA, or that used to create Intel hardware\cite{25, 14}. Finally, proof assistants are also often used simply to record mathematical proofs in a standard fashion. The motivation behind this project is straddled between these latter two goals. Courcelle’s Theorem is a large, important mathematical accomplishment, and we hope to add it to the list of verified proofs. However, although the algorithm in Courcelle’s Theorem does not have any high-stake implementations that the authors are aware of, the theorem seems to have potential of becoming useful in the field of relational databases. If a relational database is created someday that relies upon Courcelle’s Theorem to run its queries, a formal verification of the theorem may be useful in situations
where queries are being run on sensitive data where far-reaching conclusions may be drawn from the queries on that data (medical data, census data, etc).

It should be mentioned that verifying with a proof assistant is very different from verifying with an automated theorem prover. Automated theorem provers minimize the level of human interaction with the program, but they have much less expressive power than proof assistants—usually limited to that of first order logic[11]. Of course, since proof assistant code is still written by humans, there is still a possibility for error, but this space where this error can occur is minimized. The only place where a formal verification by a proof assistant can go wrong is in the definitions put forth in the program: these definitions may not exactly correspond to what the user is actually trying to verify. In other words, the only way a formal verification can go wrong is that the user is unwittingly proving the wrong thing, since they defined their goal incorrectly[11].

However, this means that as long as we agree with the definitions in a formal verification done with a proof assistant, we can trust that whatever logical implication is drawn from these definitions is proven absolutely correctly through the rules of type theory[11].

Due to this fact, a large portion of this work will be dedicated to explaining the definitions we have created, and convincing the reader that they correctly represent the mathematical definitions so that our proof may be trusted.

The tool that we use to verify Courcelle’s Theorem, as we mentioned above, is called Agda[26]. Agda is a dependently typed functional programming language based on Haskell. It was created by Ulf Norell in his PhD thesis at Gotenburg University[26]. In order for Agda’s underlying logic to remain consistent, Agda also enforced termination checking, i.e. all possible patterns in a program must be matched, and all recursion must be performed on smaller subproblems.
Agda has some of the usual trappings of regular programming languages, for example, data types, pattern matching, let expressions, and modules. Its syntax is similar to that of Haskell. One quite beautiful feature of Agda, is its use of metavariables in the code. A user can define a function and leave the proof as a metavariable, often referred to as the goal. Agda will display the expected type of the metavariable, and the user can incrementally refine it, by putting pieces of code in bit by bit. Agda will update the expected types of the remaining missing pieces.

1.5. Contributions and Future Work. The goal of this project is to complete a formal verification of Courcelle's Theorem in Agda. At this stage, we have completed our translation of the definitions given in Courcelle's Theorem: A Game Theoretic Approach[19] into Agda. This involved reformulating several complex mathematical concepts into machine-checkable definitions. Specifically, we define symbols, signatures, structures, expansions of structures, restrictions of structures, tree decompositions, MSO formulas, isomorphisms of structures, model checking games, extended model checking games, equivalence between game positions and subgames, and finally a notion of a reduced game. The bulk of this thesis document is a detailed explanation of the fact that the mathematical definitions we formalize correspond to the way we encode them. This is in an effort to convince the reader that, once we do finish the complete proof, it will be correct since our definitions are correct, for the only way a proof assistant can be incorrect is by the author of the program unintentionally encoding the wrong statement of the theorem. Additionally, we have defined the types of the functions corresponding to the lemmas in the paper that prove Courcelle's Theorem, however we have yet to implement them. We also have yet to prove the fact that the algorithm is fixed-parameter tractable linear; this is next on our agenda after completing
the proof of the algorithm itself. After this work is complete, we hope to extend our verification of Courcelle’s Theorem to some of its applications. One probable direction we will pursue is verifying the algorithm in the paper above by Grohe et al. We are also considering applications to phylogenetics and database theory, upon greater research into the applications of Courcelle’s Theorem.

1.6. Road Map. The rest of the paper will proceed in the following manner:

Chapter Two, Helpful Background and History, will expand a little bit on the three disciplines that this theorem draws upon. The first is graph minor theory, which is the origin of tree decompositions. The second is (monadic second order) logic, and some extensions of Courcelle’s Theorem which apply to questions formulated in slightly more expressive logics. We also touch on logic’s connections to graphs via graph rewriting, which is how Courcelle’s Theorem takes in its input graphs. Finally, we will touch on parametrized complexity, which is where we get the notion of fixed-parameter tractable linear from. After briefly engaging with these topics, we will allude to some common methods of proving Courcelle’s Theorem, and note their strengths and weaknesses.

Following this, in Chapter Three, we give an outline of the proof of Courcelle’s Theorem in Courcelle’s Theorem: A Game Theoretic Approach by Kneis et al.

In Chapter Four, Related Work and Applications, we touch on other verification work in this space and some applications of Courcelle’s Theorem in graph theoretic algorithms, phylogenetics, and database theory before heading into the main part of the paper.

Chapter Five, Definitions, contains the bulk of the work of this thesis. Here, we go through the concepts defined in the Kneis et al. paper, present our translations of each concept in Agda, and show how the two are equivalent. This
is in an effort to convince the reader that, once we do finish the complete proof, it will be correct since our definitions are correct. Specifically, we define symbols, signatures, structures, and expansions of structures (3.1); restrictions of structures (3.2); tree decompositions (3.3); MSO formulas (3.4), isomorphismis of structures (3.5); compatibility (3.6); structure union (3.7); games, model checking games, and extended model checking games (3.8); and finally, position equivalence, game equivalence, and reduced games (3.9).

Our last chapter before the conclusion is Chapter Six, Outline of Algorithm and Work in Progress. Here, we give a mathematical proof of Courcelle’s Theorem based on the lemmas outlined by Kneis et al. [19]. Then we present with our progress on the proof in Agda, i.e. the types of the functions in Agda with which we will prove the theorem and how they fit together.
Helpful Background and History

The concept of treewidth was created by Halin [15] under the name S-function, but it only became prominent through the work of Robertson and Seymour leading up to their famous Graph Minor Theorem.[16] A minor of a graph $G$ is any graph $G'$ that can be produced by a series of the following three operations: deleting an edge, contracting an edge, and deleting an isolated node [23]. Robertson and Seymour set out on what ended up being a twenty-five year- (and twenty-three-paper-) long journey to prove something called Wagner’s Conjecture, which became the Graph Minor Theorem. Wagner’s Conjecture is based on Kuratowski’s Theorem in graph theory, which will help set up the intuition for the theorem. Kuratowski’s Theorem states that a graph $G$ is embeddable in the plane if and only if it does not contain a subgraph homeomorphic to the complete graph $K_5$ or the complete bipartite graph $K_{3,3}$ [22]. Wagner’s Conjecture asks for a much more general principle: that if a class of graphs $K$ is minor-closed (i.e. $\forall G \in K, G'$ is a minor of $G$, then $G' \in K$), then $K$ can be characterized by a finite number of excluded minors. Robertson and Seymour re-discovered treewidth in this context, as a purely mathematical tool to prove Wagner’s Conjecture. Graph Minor Theory by Laszlo Lovasz is a well-written, concise introduction to graph minor theory and to Robertson and Seymour’s results from a purely mathematical perspective[23]. Graph minors have their own applications to computer science via coloring problems and combinatorial optimization[16], but computer scientists soon picked up on the notion of treewidth much more widely as a way of putting a constraint on
graphs in order to find faster algorithms that take advantage of this constraint. Some Recent Progress and Applications in Graph Minor Theory is a more extended resource to learn about the landscape of graph minor theory beyond just Robertson and Seymour’s work [16].

Graph Rewriting: An Algebraic and Logical Approach by Courcelle [5] provides a background to expressing graphs as logical structures, which is how Courcelle’s Theorem interprets graphs. It explains which graph properties are definable in first order logic (for example, having degree k is first order logic expressible, but not connectedness), monadic second order logic (e.g. connectedness, but not having the same number of edges in label sets A and B) and second order logic (e.g. that a graph is finite)[5].

Recall our discussion in the introduction about MSO, where we mentioned the fact that while MSO formulas can express the notion of k-colorability or vertex set, but because one cannot count in MSO, one cannot express the minimum vertex set or the minimum coloring. Successful efforts have been made to overcome this restriction. For example, Seese et al. introduced Extended MSO, or EMS, which allows for optimizations of MSO-definable problems, as well as problems over graphs with weight functions, and several new problems besides. A more complete description of which problems are MSO and EMS definable are in [1].

Kneis et al. develop their own extension, called LinMSO, that works specifically for linear optimizations of MSO definable problems [19]. Both Seese et al. and Kneis et al. have proven that their extensions of MSO satisfy Courcelle’s Theorem, i.e. that problems in EMS and LinMSO can be solved in FPT-linear time on graphs of bounded treewidth. [19, 1]. The formulation of Courcelle’s Theorem with Extended MSO is very widely used, but in this paper we only verify the original version.
Parametrized complexity, the field that gives us the notion of fixed parameter tractable algorithms, was developed by Robert Downey and Michael Fellows in the 1990s [8]. The main idea in parametrized complexity is thinking of complexity in multiple dimensions rather than in just one. Downey and Fellows explain their idea as a "deal with the devil" to compensate for the inevitable combinatorial explosion in most interesting algorithms: they limit the explosion to one attribute, which becomes the parameter, of the input[7]. In our case, the inevitable exponential blowup is checking every element of an underlying set of a structure at each universal or existential quantifier in the input formula. This blowup becomes confined to the treewidth of the graph and the encoding of the MSO sentence.

In Downey and Fellows' parametrized complexity book, they demonstrate one of many ways to prove Courcelle’s Theorem, using the Myhill-Nerode theorem and the method of test sets[7][Chapter 6]. However, this method has never been implemented. There are several different ways to prove, and implement, the Theorem. The most common way to prove the theorem is through the automata-theoretic approach: a tree automaton $A_\tau$ is constructed in time that is exponential only in the treewidth $t$ and the size of the formula $\phi$. $A_\tau$ accepts a tree decomposition of width $t$ if and only if the corresponding graph satisfies $\phi[9, 8]$.

The problem with this approach is that at every universal and existential quantifier, the automaton expands exponentially. Since automata are so well understood, there are tools to minimize these generated automata, such as the MONA tool [18]. However, one particularly frustrating thing about the automata-theoretic method is that often, most of the states created are never used. Sometimes, even though the minimized automata are small, the initial computations are so expensive that even the MONA tool fails to run the algorithm—this can occur on graphs as small as $2 \times n$ grids [19]. There are a few different ways to
circumvent this issue; they all involve some version of creating what you need as you go. Courcelle and Durand use functions instead of tables to model automata in an effort to make the algorithm faster and dub the method using fly-automata [4]. The name is due to the fact that the state-transition functions are created "on-the-fly". Alternatively, this can be done by scrapping the idea of automata all together. Ganzow and Kaiser bypass the use of automata at all and directly manipulate the input formulas [10]. This is also the approach that Kneis et al. take in their proof of Courcelle’s Theorem, which is the version of the proof this work aims to verify.
CHAPTER 3

Outline of Kneis et al. Algorithm

Kneis, Langer, and Rossmanith do away with automata all together in their paper Courcelle’s Theorem–A Game Theoretic Approach[19], which is the algorithm we use to verify Courcelle’s Theorem in our work. Kneis et al. come up with a new strategy to tackle Courcelle’s Theorem, borrowing tools from game theory: namely, model checking games. The algorithm they outline to decide the formula on the structure is a dynamic programming algorithm on the tree decomposition[19]. As mentioned above, the problem with the automata theoretic approach is that many of the states created are never actually used to decide the formula on the input graph, so prohibitively large amounts of time and space are spent creating things that will never be used. Kneis et al. suggest this is because this proof and implementation method does not pay attention to the attributes of the specific graph and MSO formula given. Instead, the automata theoretic implementation does a very similar thing no matter what the input is[19]. To fix this problem, Kneis et al. take the details of the input into account, and in light of the input, systematically trim down the information stored to be only what is necessary[19]. As alluded to earlier, there is evidence that this implementation runs in a reasonable amount of time (from a few seconds to under 24 hours) for graphs up to treewidth 12 for minimum vertex cover, dominating set, and 3-colorability[19]. An explanation of the algorithm is in the last chapter of this thesis; however, we give a quick intuitive idea. At each node $i$ of the tree decomposition, there is a bag $X_i$ of vertices of the original graph. Additionally,
we can associate with each node a set $A_i$, which is the union of all bags on the tree starting at the leaves and up until, and including, the bag $X_i$. The paper shows that if an MSO sentence $\phi$ is shown to hold over a restriction of the graph $G$ to the set $A_i$, called $G[A_i]$—a notion that we will formalize later but essentially means that $G[A_i]$ is the graph $G$ with all the elements that are not in $A_i$ thrown out of it—then $\phi$ is true over the whole graph. And similarly if the $\phi$ can be shown to not hold over $G[A_i]$, then it does not hold over the whole graph. (The way in which the formula is “shown to hold” is via a model checking game, which again, we will explain in the pages to come.) The algorithm traverses the tree decomposition bottom up, evaluating whether or not $\phi$ holds for each $G[A_i]$ via a game: if the formula holds, the algorithm exits and returns that the formula holds, similarly if the formula is shown to be false on the graph. Alternatively, if the data is insufficient, then there is a special combining algorithm that combines the game on $G[A_i]$ with the game on $G[X_h]$, where $h$ is the next node above $j$. Note that the combining algorithm takes the recursive call and only the bag at the next node, not the entire set of all the bags below it. Because of this, the algorithm never checks the formula over the entire graph, but only over the subsets of the graph in the bags $X_i$ of the tree decomposition. Note that by definition of treewidth, the size $X_i$ will always be less than the graph’s treewidth $t$. So, even though there can be exponential blow up to check if $\phi$ holds on $G[X_i]$, the time is only proportional to $|X_i|$, which is a constant. The computation at each node, then, is constant. Thus, the algorithm is linear in the size of the input graph. The algorithm checks the formula over the restriction at each tree decomposition node, where the number of nodes in the tree decomposition is linearly related to how many nodes are in the graph, and the computation at each node is constant time. Unfortunately, as mentioned, the algorithm on these restrictions is exponential.
in the size of the bags, which is where the nasty exponential constant in the size of the parameters partially comes from. However, the algorithm in Kneis et al. addresses this problem as well. They introduce a notion of isomorphism between games, so that the algorithm keeps track of as few games as possible, which cuts down on the exponential constant. The details of these ideas will be explained in the pages to come.
CHAPTER 4

Related Work and Applications

As far as the author of this thesis is aware, there are no other attempts to formally verify Courcelle’s Theorem. There have been papers verifying processes related to Courcelle’s theorem. For example, Martin Streker verified some systems of graph rewriting, which is central to Courcelle’s Theorem [28]. But there is no literature we can find on tackling Courcelle’s Theorem specifically.

Some application areas of Courcelle’s Theorem include graph theoretic algorithms: for example, a paper on constructing graph minor decompositions by Reed et al. uses Courcelle’s Theorem in the proof [24]. Another application area is phylogenetics, where scientists need to answer questions about graphs called agreement forests, which depict how close two different proposed phylogenetic histories are. The questions they need to ask are expressible in MSO, and in practice, one researcher, Stephen Kelk, points out that agreement forests often have low treewidth. A good, fast implementation of Courcelle’s Theorem, then, would likely speed up the research of phylogeneticists [17]. Another potentially exciting area of application is to database theory, which deals mostly with first order logic questions asked about relational databases, which are of course covered by MSO. Courcelle’s Theorem immediately extends to relational databases with bounded treewidth [13]. So, Courcelle’s Theorem promises a faster implementation for SQL queries, which, if brought to fruition, could potentially cause a huge increase in querying speed.
CHAPTER 5

Definitions

In this chapter, we systematically go through the definitions stated by Kneis et al. [19], present our Agda translations of these definitions, and then explain how the two connect. These definitions feed into one another, both mathematically and in the code, and we do our best to present the definition in an intuitive fashion. Broadly, the first three groups of definitions, Structures (which includes extensions and restrictions), Tree Decompositions, and MSO Formulas, are the input types to the overall algorithm. As we will see below, structures are a generalized class of objects that include graphs, so structures take the place of graphs in our in our input definitions. The next few definitions, Isomorphisms of Structures, Compatibility, and Union of Structures, is how we can meaningfully relate structures to one another. The next section, Games, including Model Checking Games and EMC Games, describes the objects that the Kneis algorithm and our code use to determine if a formula is true on a structure. Finally, the reduced games definitions show how we can limit the information we can store in our games, i.e. in our methods of proving formulas true on structures, in order to save time in the algorithm.

1. Structures

The version of Courcelle’s Theorem that is verified in this paper generalizes beyond graphs as the input object to the algorithm, and instead has something called a structure as input. We show that any MSO sentence can be decided on a
structure of bounded treewidth in FPT-linear time. In order to define structures, we have to introduce some other concepts first.

A **symbol** $S$ is an object with an arity, $\text{arity}(S) \geq 0$. If a symbol has arity 0, then it is a **constant**; if it has arity greater than 0, it is a **relation**.

A **signature** $\Sigma$ is a finite set of symbols.

A **structure** $A$ is a tuple $A = (A, (R^A)_{R \in \text{rel}(\Sigma)}, (c^A)_{c \in \text{constants}(\Sigma)})$, where $A$ is a finite set called the universe of $A$, and $R^A$ and $c^A$ are interpretations of the relations and constants in $\Sigma$ using the elements of $A$. More precisely, $R^A \subset A^{\text{arity}(R)}$, and $c^A$ is either an element of $A$, or $c^A = \text{nil}$.

Note that a constant can be interpreted as $\text{nil}$. If a structure $A$ interprets some $c^A$ as $\text{nil}$, we say that $c^A$ is uninterpreted. If the structure leaves a blank spot at that symbol $c^A$, it does not interpret it as anything. We say that $c \in \text{interpreted}(A)$ if and only if $c^A = a \in A$, i.e. $c^A \neq \text{nil}$. We often call the constants $c$ nullary constants because they have arity zero.

It may be useful to think of the signature as a mold to pour clay into, where clay is a finite set. Or, the reader may imagine symbols in the signature as placeholders that are waiting to be filled by elements of a set. When a structure interprets a constant $c$ as $\text{nil}$, no clay goes into that part of the mold, $c$ remains empty.

We illustrate the definition of structures with the example of graphs. A possible signature for graphs is $\Sigma_G = \text{Edge}(x, y)$, where edge is a binary relation that determines which vertices have an edge in between them. This signature is not a graph itself. An actual graph would be a structure $G$ over the graph signature $\Sigma_G$ with a set, $V$, of vertices, $G = (V, \text{Edge}(x, y)^G)$. $\text{Edge}(x, y)^G \subset V \times V$ is an interpretation of the $\text{Edge}(x, y)$ relation that specifies all of the vertices that have an edge between them for that specific graph. Note that there are multiple
different graphs $G'$ that have the same underlying $V$: they have different subsets of $V \times V$ for the interpretation of the edge relation, which makes $G'$ have different edges than $G$.

1.1. Expansion. Necessary to understanding the code is the idea of an expansion of a structure. Suppose $\Sigma$ is a signature, and $\{R_1,...R_l,c_1,...c_m\}$ is a set of symbols, each of which is not in $\Sigma$. Then we say a signature $\Sigma' = \{\Sigma, R_1,...R_l,c_1,...c_m\}$ is an expansion of $\Sigma$. Then, if $A$ is a $\Sigma$-structure, and $A'$ is a $\Sigma'$-structure, such that $(R^A) = (R^{A'})$ for all relations symbols $R \in \Sigma$, and $c^A = c^{A'}$ for all $c \in \Sigma$, we say $A'$ is a $\Sigma'$-expansion of $A$. Suppose we have a $\Sigma$-structure $A$, and suppose that $\Sigma' = \{\Sigma, c_1\}$. Also suppose we have an element $u_1 \in (A \cup nil)$. (We say $(A \cup nil)$ and not simply $A$ since as explained above, a constant can be interpreted as a set element, or not interpreted, i.e. interpreted as $nil$.) Then we can write the $\Sigma'$-structure $A'$ as $A' = (A, u_1)$, to show that $A'$ is a $\Sigma'$-expansion of $A$, and that $u_1$ is the interpretation for the new symbol $c_1$. Similarly, if $\Sigma' = \{\Sigma, R_1\}$ for some relation symbol $R_1$, and we have a set $U_1 \subset A^{\text{arity}(R)}$, then we could write a $\Sigma'$-expansion of $A$ as $A' = (A, U_1)$. We will see this in the Agda code.

We designed our algorithm to respect the idea of structures as inputs, but in this particular implementation, we tailored it to expect graphs as input. As mentioned earlier, in the non-extended form of MSO we are using in this work, one cannot decide formulas on graphs that quantify over edges. In order to remedy this, we instead think of a graph as a set with two different types of elements: nodes and edges. This way, we can quantify over edges, since they are simply constants in the signature rather than a binary relation. However, including edges as elements of the underlying set of a structure $G$ instead of as a binary relation has the implication that edges and nodes are indistinguishable to the algorithm, which
can pose problems. For example, suppose we want to have a graph $G_{\text{red}}$ that has a special set of red nodes. Then there would be a unary relation in this graph's signature, $\text{Red}(x)$, and an interpretation in the structure, $\text{Red}(x)^{G_{\text{red}}} \subset G_{\text{red}}$, that specified which elements of the underlying set $G_{\text{red}}$ were the red vertices. However, the problem here is that some of these set elements of $\text{Red}(x)^{G_{\text{red}}}$, the "red vertices", might actually be edges, since vertices and edges in the underlying set are indistinguishable to the relation. So how do we make sure that only vertices are allowed in this set?

We fix this problem in the code by introducing the data type $\text{Tp}$, which has an edge constructor and a node constructor. We then require in our datatype for symbols, $\text{SigThing}$, that constants (constructor $i$) specify which kind of set elements they want, nodes or edges, by carrying a $\text{Tp}$ with them. Relations (constructor $r$) specify whether they want a node or edge for each argument, by carrying a list of $\text{Tp}$ with them. This solves the problem illustrated above because the relation $\text{Red}(x)$ would really be a one-element list ($\text{node}::{[]}\rangle$, specifying that it only allows nodes in its interpretation-set. Another, related reason why we introduce $\text{Tp}$ is to make sure that when we have quantifiers in a formula, we specify whether that quantifier is for edges or for nodes, in order to ensure that we have as small a search space as possible. The code for $\text{Tp}$ and $\text{SigThing}$, symbols, is below.

```haskell
data $\text{Tp}$ : Type where
  node : $\text{Tp}$
  edge : $\text{Tp}$

data $\text{SigThing}$ : Type where
  $i$ : $\text{Tp} \rightarrow \text{SigThing}$
  $r$ : List $\text{Tp} \rightarrow \text{SigThing}$
```

Now we come to our Agda definition of a signature. A Signature in Agda is a list of SigThings, i.e. a list of symbols. And, a list is finite by definition, so we satisfy the requirement that the signature is a finite set of symbols. We have two functions on Signatures, which allows us to extend them: the function ,i allows us to add another constant onto our Signature, and the function ,r allows us to add another relation onto our Signature. These functions take a Signature and either a constant or a relation, and return a Signature which is the original Signature appended with the constant.

\[ \text{Signature} = \text{List} \text{SigThing} \]

\[ _\_,i_ : \text{Signature} \rightarrow \text{Tp} \rightarrow \text{Signature} \]

\[ \Sigma, i \tau = i \tau :: \Sigma \]

\[ _\_,r_ : \text{Signature} \rightarrow \text{Args} \rightarrow \text{Signature} \]

Coming up with a definition for a structure was a little bit more complicated, since there are a few more moving parts. Recall the definition of structure above: A structure \( A \) is a tuple \( A = (A, (R^A)_{R \in \text{rel}(\Sigma)}, (c^A)_{c \in \text{constants}(\Sigma)}) \), where \( A \) is a finite set called the universe of \( A \), and \( R^A \) and \( c^A \) are interpretations of the relations and constants in \( \Sigma \) using the elements of \( A \). In order to represent this in Agda, we needed to define what the underlying set would be. A set is a collection of elements, so before we could define sets, we had to define what sort of elements would be making up these collections. We called the objects in our space, the elements of our sets, our clay that would fill our signatures, Individs. An Individ takes a Tp, i.e. data as to whether it is a node or an edge, and returns a type. This type is the underlying data that our input graphs will be made up of. In our implementation, we chose this type to be a string. One could change the code to model the set elements as any kind of structure one might like.
Individ : Tp \to Type
  Individ node = String
  Individ edge = String

Args : Type
  Args = List Tp

Individs : Args \to Type
  Individs [] = Unit
  Individs (\tau :: \tau s) = Individs \tau s \times Individ \tau

Now that we have our objects, we can define sets of objects. Our notion of set is called a \textbf{Subset} (because set is a reserved word in Agda). We defined \textbf{Subset} as a predicate on elements (\textbf{Individ}) of a certain type. So, edge elements and node elements can have different stipulations that determine whether or not they are in a given set. In case the difference between \textbf{Tp} and \textbf{Individ} is confusing, think of a \textbf{Tp} \tau as a label for a certain slot in a signature that says what type of thing can go in it, and an \textbf{Individ} of type \tau is the thing that goes in.

\textbf{Subset} = (\tau : Tp) \to Individ \tau \to Type

\textbf{DecidableSub} : (S1 : Subset) \to Type

\textbf{DecidableSub} S1 = \forall \{\tau\} (x : Individ \tau) \to Dec (S1 \tau x)

For now, our construction \textbf{Subset} does not guarantee that the set is finite as is required in the definition of a structure. We anticipate revising this once we see what we need in the rest of the proof— we will definitely add the conditions of finiteness and decidableness to the subset, but there may be other qualities that we require that are as of yet unseen. Adding these qualifications to the subset is a trivial task. We would simply add a bit more data that we could abstract away and leave the code unchanged, so please pay this small detail no mind.
After defining \textbf{Subset}, we defined \textbf{IndividS}, which is simply an \textbf{Individ} together with the data that it is in a specified \textbf{Subset}. This will become useful later on.

\[
\text{IndividS} : \text{Subset} \to \text{Tp} \to \text{Type}
\]

\[
\text{IndividS} A \tau = \Sigma \lambda (x : \text{Individ } \tau) \to A \tau x
\]

\[
\text{IndividsS} : \text{Subset} \to \text{Args} \to \text{Type}
\]

\[
\text{IndividsS} A [] = \text{Unit}
\]

\[
\text{IndividsS} A (\tau :: \tau s) = \text{IndividsS} A \tau s \times \text{IndividS} A \tau
\]

We also defined a datatype \textbf{OC}, with constructors open and closed. This datatype will be part of the information that the Agda type for structures carry with them. A structure being open corresponds to possibly it having constants interpreted as \textit{nil}, and a structure being closed means that all constants in \(\Sigma\) are interpreted in \(A\).

\textbf{data} \textbf{OC} : \text{Type} \textbf{where}

- \textbf{Open} : \textbf{OC}
- \textbf{Closed} : \textbf{OC}

Then, combining \textbf{Subset}, \textbf{OC}, and \textbf{Signature}, we defined something called a \textbf{StructureS}, which we build on to finally define a structure. Think of the \textbf{StructureS} as the part of the structure tuple that consists of all the interpretations of the signature, (i.e. all the \(R^A, c^A\)). A \textbf{StructureS} is a datatype that requires information saying whether it is open or closed (\textbf{OC}), a \textbf{Subset}, and a \textbf{Signature}. Now would be a useful time to review the notion of expansions of structures given slightly above this definition. \textbf{StructureS} has four constructors: empty ([]), an expansion with an element of the underlying set (,.is, expansion with \textbf{IndividS} A \(\tau\)), an expansion with a \textit{nil} (,.none, which mandates that the \textbf{StructureS} be open), and an expansion with a relation tuple (,.rs, with an expansion with \textbf{IndividsS} A \(\tau s\)). Note that
the constructors with extensions return a \texttt{StructureS} over an extended \texttt{Signature},
adding a constant symbol of type \(\tau\) in the \texttt{is} and \texttt{none} cases \((\Sigma, i \tau)\), and adding
a relation symbol \(r\) in the \texttt{rs} case \((\Sigma, r \tau s)\).

\textbf{data} \texttt{StructureS} : \texttt{OC} \to \texttt{Subset} \to \texttt{Signature} \to \texttt{Type}
where

\[
\emptyset : \forall \{oc A\} \to \texttt{StructureS} oc A \emptyset
\]

\[\_\_\_\text{is} \_\_ : \forall \{oc A \Sigma \tau\} \to \texttt{StructureS} oc A \Sigma \to \texttt{IndividS} A \tau \to \texttt{StructureS} oc A (\Sigma, i \tau)
\]

\[\_\_\_\text{none} \_\_ : \forall \{oc A \Sigma \tau\} \to \texttt{StructureS} oc A \Sigma \to \texttt{StructureS} \texttt{Open} A (\Sigma, i \tau)
\]

\[\_\_\_\text{rs} \_\_ : \forall \{oc A \Sigma \tau s\} \to \texttt{StructureS} oc A \Sigma \to (\texttt{IndividS} A \tau s \to \texttt{Type}) \to \texttt{StructureS} oc A (\Sigma, r \tau s)
\]

We then create the type \texttt{Structure}, which requires an \texttt{OC} and a \texttt{Signature}, and
consists of a product of a \texttt{Subset} and a \texttt{StructureS} that goes with that \texttt{Subset}, i.e.
a tuple of a set, and interpretations of a given \texttt{Signature} from that set. This is
exactly what a structure is defined to be mathematically.

\texttt{Structure} : \texttt{OC} \to \texttt{Signature} \to \texttt{Type}
\texttt{Structure oc Sig} = \Sigma \lambda (A : \texttt{Subset}) \to \texttt{StructureS oc A Sig}

2. Restrictions

Suppose we have a structure \(A = (A, (R^A)_{R \in \texttt{rel}(\Sigma)}, (c^A)_{c \in \texttt{constants}(\Sigma)})\). We can
restrict this structure to a smaller subset \(A_1 \subseteq A\), and call this structure \(A[A_1]\).
In this new structure, we insist that we only have the set elements in \(A_1\), and so
we only have the constant and relation interpretations that are contained entirely
inside \(A_1\). Formally, given a structure \(A\) and a set \(A_1 \subseteq A\), the restriction of
\(A\) to \(A_1\), \(A[A_1]\), is a structure with universe \(A_1\), \((R^{A[A_1]}) = R^A \cap A_1^{arity(R)}\), and
\(c^{A[A_1]} = c^A\) if \(c^A \in A_1\), and becomes interpreted as \texttt{nil} otherwise.

In Agda, we encode this notion by a helper function \texttt{restrictionS} on \texttt{StructureS}'s,
and from that we build the function \texttt{restriction} on \texttt{Structures}. \texttt{restrictionS} requires
a Structure $A_1$, a subset $S_1$, a proof that $S_1$ is decidable ($\text{DecidableSub } S_1$), and a proof that $S_1 \subseteq A$ ($\text{Sub } S_1 (A_1)$). It returns the interpretations of $A_1$ restricted to the set $S_1$. Note that in our helper function $\text{restrictionS}$, we ask for a proof that the subset we restrict to is decidable; when we fix our notion of subset this will no longer be necessary. The code is relatively straightforward and there is no need to go through it in detail; essentially this code runs through each $\text{IndividS}$ that interprets a constant in the $\text{Signature}$ in the $\text{StructureS } A_1'$, and if it is not in the restriction subset $S_1$, it replaces this interpretation with a $\text{nil}$. And it only includes relation tuples that are entirely within $S_1$. This is because in the last case, we stipulate that $v$ (in $(\lambda v \to U (\text{promoteIndividsS } sb \ v))$), which is the tuple of set elements that interprets the relation $rs$, has type $\text{IndividsS}$, in $S_1$. (Having type $\text{IndividsS}$ in $S_1$ means that $v$ is tuple of set elements, $\text{Individs}$, in $S_1$.) This is done by the function $\text{promoteIndividsS}$, which we will not go into, but shuttles the type of set elements between sets—i.e. if $a$ is a set element of $A$, and $B \subseteq A$, then certainly $a$ should be a set element of $B$ as well. $\text{promoteIndividsS}$ ensures this.

$$\text{restrictionS} : \forall \left\{ \Sigma \right\} \left\{ \text{oc1} \right\} \left\{ A_1 \right\} (A_1' : \text{StructureS } \text{oc1 } A_1 \Sigma) (S_1 : \text{Subset}) \rightarrow \text{DecidableSub } S_1 \rightarrow \text{Sub } S_1 (A_1) \rightarrow \text{StructureS Open } S_1 \Sigma$$

$\text{restrictionS } [] \ S_1 \text{ dec sb } = []$

$\text{restrictionS } (A_1', \text{is x}) \ S_1 \text{ dec sb with dec (fst x)}$

... | $\text{Inl inS} = \text{restrictionS } A_1' \ S_1 \text{ dec sb ,is (fst x , inS)}$

... | $\text{Inr out} = \text{restrictionS } A_1' \ S_1 \text{ dec sb ,none}$

$\text{restrictionS } (A_1', \text{none}) \ S_1 \text{ dec sb } = \text{restrictionS } A_1' \ S_1 \text{ dec sb ,none}$

$\text{restrictionS } (A_1', \text{rs U}) \ S_1 \text{ dec sb } = \text{restrictionS } A_1' \ S_1 \text{ dec sb ,rs (} \lambda v \to U (\text{promoteIndividsS } sb \ v))$

The code for $\text{restriction}$ simply attaches the appropriate underlying $\text{Subset}$, $S_1$, to the restricted interpretations given by $\text{restrictionS}$, $\text{StructureS Open } S_1 \Sigma$. 

31
restriction takes essentially the same inputs as restrictionS, except that it requires and returns a Structure and not a StructureS.

\[
\text{restriction} : \forall \{\Sigma\} \{\text{oc1}\} (A1 : \text{Structure oc1 } \Sigma) (S1 : \text{Subset}) \\
\rightarrow \text{DecidableSub} S1 \rightarrow \text{Sub} S1 (\text{fst} A1) \rightarrow \text{Structure Open } \Sigma \\
\text{restriction} (A1', \text{struc}) S1 \text{ dec sb} = S1, \text{restrictionS struc S1 dec sb}
\]

3. Tree Decompositions

Recall our definition of tree decompositions above. The formal definition upon which we based our code is more general than the definition given above, because the game-theoretic algorithm works over all structures and not simply over graphs. The definition for a tree decomposition of an arbitrary structure is as follows:

A tree decomposition of a structure \( A \) over a signature \( \Sigma \) is a tuple \((T, X)\), where \( T = (T, F) \) is a rooted tree and \( X = (X_i)_{i \in T} \) is a collection of subsets \( X_i \subset A \), (where \( A \) is the underlying subset of \( A \)) such that

1. \( \bigcup X_i = A \)
2. \( \forall \) \( p \)-ary relation symbols \( R \in \Sigma \), and all \( (a_1, a_2, \ldots, a_p) \in R^A \), \( \exists i \in T \) such that \( (a_1, a_2, \ldots, a_p) \subset X_i \), and
3. If \( i, j, k \) are nodes of \( T \) such that \( i \) is on a path between \( j \) and \( k \) in \( T \), then \( (X_j \cap X_k) \subseteq X_i \).

Furthermore, in the algorithm that Kneis et al.[19] outline, they use only a type of tree decompositions called nice tree decompositions. Nice tree decompositions are directed, and every edge is directed away from the root. Nice tree decompositions have exactly four types of nodes: leaves, forget nodes, introduce nodes, and join nodes.

- A leaf \( i \) in nice tree decomposition has an empty bag, i.e. \( X_i = \emptyset \).
• A forget node is a node $i$ that has exactly one child, $j$, and there exists some $a \in A$, $X_i = X_j / a$.

• An introduce node is a node $i$ that has exactly one child, $j$, and there exists some $a \in A$, $a \notin X_j$ such that $X_i = X_j \cup a$.

• A join node is a node $i$ with exactly two children, $j$ and $k$, where $X_i = X_j = X_k$. We say that node $i$ is below node $j$ if there is a path from $i$ to $j$ in $T$.

Recall our diagrams of a graph and its tree decomposition from the introduction. We add here a depiction of a nice tree decomposition on that same graph. We can turn a tree decomposition into a nice tree decomposition in linear time, so requiring a nice tree decomposition does not add a significant amount of time to the algorithm.

Finally, keep this idea in mind: for every node $i$ of a nice tree decomposition of a $\Sigma$-structure $A$, let $A_i = \bigcup_{j \text{ at or below } i} X_j$. See that $A_i \subseteq A$. Then for every node $i$, we can associate with $i$ a substructure $A_i[A_i]$, i.e. a restriction of $A$ to $A_i$.

In Agda, we modeled a nice tree decomposition as a datatype with four constructors, \texttt{Empty}, \texttt{Delete}, \texttt{Intro}, and \texttt{Join}, one for each type of node. Each node carries two subsets with it; its bag ($X$) and the set of all elements in all bags at and below it ($B$). More precisely, at node $i$, $B = \bigcup_{n \text{ at or below } i} X_n$. We will prove that this is the case after we give an explanation of each constructor.

\begin{verbatim}
 data TreeDecomp : Subset -> Subset -> Type where
  Empty : TreeDecomp empty empty
  Delete : \forall \{\tau\} (X : Subset) (B : Subset) (x : Individ \tau)
           -> TreeDecomp (union X (singleton \{\tau = \tau\} x)) B
           -> TreeDecomp X B
  Intro : \forall \{\tau\} (X : Subset) (B : Subset) (x : Individ \tau)
          -> TreeDecomp (intersection X (singleton \{\tau = \tau\} x)) B
          -> TreeDecomp X B
  Join : \forall \{\tau\} (X : Subset) (B : Subset) (x : Individ \tau)
       -> TreeDecomp (intersection X (singleton \{\tau = \tau\} x)) B
       -> TreeDecomp X B

end
\end{verbatim}
Figure 1. A nice tree decomposition of the graph $G$ from Figure 1.2

\[
\rightarrow (xnew : (\text{Sub (singleton } \{\tau = \tau\} \times) \text{ (complement } B)))
\]

\[
\rightarrow (\forall \{\tau s\} \rightarrow (xs : \text{IndividsS} (\text{fst } A) \tau s)
\]

\[
(\text{inBandx} : \text{allin } \tau s (\text{union } B (\text{singleton } \{\tau = \tau\} \times)) (\text{nosubset } \tau s (\text{fst } A) \timess))
\]
The Empty constructor corresponds to the mathematical definition of the leaf of a tree decomposition. It has an empty set for its bag and an empty set for the elements below. This corresponds to the definition of a nice tree decomposition given above: the leaves are required to have no nodes below them, necessitating that \( B = \bigcup_{j \text{ at or below } i} X_j = \emptyset \), and and also to have empty bags, necessitating that \( X_i = \emptyset \).

The delete constructor, i.e. the forget node, takes a tree decomposition of a bag \( X_i \cup x \) and a set \( B \) of elements below, and returns a tree decomposition with just the bag \( X_i \) and the same set \( B \) of elements below.
CHAPTER 5. DEFINITIONS

The Intro constructor, for the introduce node, requires a few additional pieces of information. Firstly, it needs a proof (\(x_{\text{new}}\)) that the element (\(x\)) being introduced is indeed new. In the code this is described as \(x\) being an element of the complement of \(B\), the set of objects below the Intro node in the tree decomposition. Secondly, it needs a proof showing that for all relations \(R\) that have tuples made up exclusively of \(x\) and and elements of \(B\), all of the elements in these tuples are in fact contained in \(X \cup x\). This corresponds to the next five lines of code after \(x_{\text{new}}\). This condition is necessary in order to satisfy the second and third conditions of the definition of tree decompositions. To illustrate this, consider some introduce node \(i\). Suppose the second proof were false, i.e. there exists some tuple \(p\) consisting exclusively of elements of \(B\) and \(x\) such that \(p \notin X_i\). Recall that the second condition of a tree decomposition requires that every tuple of every relation be in some bag \(X_j\). We know by the first required proof that \(x\) is not in \(B\), so any relation containing \(x\), including \(p\), must be either in \(X_i\) or in some \(X_j\) above \(X_i\). So since \(p \notin X_i\), \(p\) must be in some some \(X_j\) above \(X_i\). Suppose one of the elements \(b \in B\), \(b \in p\) is in some bag \(X_h\), with \(h\) below \(i\) in \(T\). Suppose \(b \notin X_i\). Then \(b\) must be in \(X_j\), since \(b \in p\). Then, by condition three of tree decompositions, since there is a path between \(X_h\) and \(X_j\) through \(X_i\), \((X_h \cap X_j) \subset X_i\). This means that \(b \in X_i\). But this is a contradiction, so it must in fact every tuple consisting entirely of elements of \(B\) and \(x\) must be contained in \(X_i\).

Finally, we come to the Join constructor. First, note that for the join node of a tree decomposition, the bags \(X_i = X_j\) for the nodes \(i, j\) being joined, but the \(B_i, B_j\), the sets of all elements in all bags below \(i\) and \(j\) respectively, may not necessarily be the same. In fact, they likely are not. The Join constructor in Agda requires two proofs, (written as three since we did not make equality for sets): that \(X = B_i \cap B_j\), and that for all relation-tuples in \(B_i \cup B_j\), either the entire relation
tupel is in $B_i$ or it is in $B_j$. In other words, there is no relation tupel in $B_i \cup B_j$ that contains elements from both $B_i \setminus B_j$ and $B_j \setminus B_i$. The first proof is required to maintain condition three of tree decompositions. Suppose there is some join node $i$, which joins nodes $j$ and $k$ where $\exists x \in X_i$ such that $x \notin B_j \cap B_k$. However, note that by construction, $X_i = X_j = X_k$, so $x \in X_j \subset B_j$, and $x \in X_k \subset B_k$, so $x \in B_j \cap B_k$, and we have reached a contradiction. So $X_i = X_j = X_k \subset B_j \cap B_k$.

Suppose $\exists b \in B_j \cap B_k$ such that $b \notin X_i$. Now, since $b \in B_j$, there is some node $m$ at or below $j$ such that $b$ is in $X_m$. And, since $b \in B_k$, there is some node $l$ at or below $k$ such that $b$ is in $X_l$. Since node $i$ joins nodes $j$ and $k$, there is a path between $m$ and $l$ through node $i$. Therefore, by condition three, $X_m \cap X_l \subset X_i$. So $b$ must be in $X_i$. However, we assumed the opposite, so we have reached a contradiction. Therefore $B_j \cap B_k \subset X_i$. Putting these two proofs together, we see it must be that $B_j \cap B_k = X_i$.

The second proof is necessary to preserve the second and third condition of tree decompositions. Suppose that there exists some relation tupel $r \in R^A$ such that $r \in B_j \cup B_k$, and $r \notin B_j$ and $r \notin B_k$. That is, there exists some $b_j \in r$ such that $b_j \in B_j \setminus B_k$ and some $b_k \in r$ such that $b_k \in B_k \setminus B_j$. Then by condition two of tree decompositions, there must be some node $m$ in the tree decomposition such that $r \in X_m$. However, since $r$ contains elements not in $B_j$ and elements not in $B_k$, node $m$ must be above both $j$ and $k$. And since $r \notin B_j \cap B_k = X_i$, $r \notin X_i$ as well. So $m$ must be above $i$. However, there is some node $a$ below node $j$ such that $b_j \in$ the bag $X_a$. And, there must be a path between node $a$ and node $m$, and it must go through node $i$, since $i$ is the only connection between $B_j$ and $B_k$. Therefore, $b_j \in X_a \cap X_m \subset X_i$. A symmetric argument can be used to show that $b_k \in X_i$. Therefore, $b_j, b_k \in X_i = B_j \cap B_k$. However, we assumed this was not
the case; therefore we have reached a contradiction, and so it must be that for all relation-tuples in $B_i \cup B_j$, either the entire relation tuple is in $B_i$ or it is in $B_j$.

Now we run through a proof of that fact that for all nodes $i$ of a TreeDecomp, $B = \bigcup_{n \text{ at or below } i} X_n$. We do this by showing that this property holds at each constructor.

We prove by induction. For the base case, we have that if node $i$ is an Empty, then it has $\bigcup_{n \text{ at or below } i} X_n = \emptyset$ since the bag at a leaf is empty by definition, and there are also no nodes below a leaf. Then since $B = \emptyset$ as well, we have $\bigcup_{n \text{ at or below } i} X_n = B$, as desired.

Now we run through our inductive steps, one for each constructor. Consider a node $i$. Firstly, suppose $i$ is a Delete node. By definition, we know that the node directly below $i$, call it $i-1$, had $X_{i-1} = X_i \cup x$ for some $x$. And, we know that $B_{i-1} = B_i$. By our inductive hypothesis, we have that $B_{i-1} = \bigcup_{n \text{ at or below } i-1} X_n$. Note that $\bigcup_{n \text{ at or below } i} X_n = (\bigcup_{n \text{ at or below } i-1} X_n) \cup X_i$. But since $X_{i-1} = X_i \cup x$, then $X_i \subset X_{i-1}$, $X_i$ brings in nothing new. So $\bigcup_{n \text{ at or below } i} X_n = \bigcup_{n \text{ at or below } i-1} X_n = B_i$, as desired.

Suppose $i$ is an Introduce node. Then by definition, $X_i = X_{i-1} \cup x$ for some $x \notin B_{i-1}$, and $B_i = B_{i-1} \cup x$. By induction, we have that $B_{i-1} = \bigcup_{n \text{ at or below } i-1} X_n$. Note that $\bigcup_{n \text{ at or below } i} X_n = \bigcup_{n \text{ at or below } i-1} X_n \cup X_i$. But note that this is simply $B_{i-1} \cup x = B_i$, as desired.

Suppose $i$ is a Join node; suppose it joins nodes $j$ and $k$. By definition, we have $B_i = B_j \cup B_k$, and we have $X = B_j \cap B_k$. By the inductive hypothesis, we have that $B_j = \bigcup_{n \text{ at or below } j} X_n$, and $B_k = \bigcup_{n \text{ at or below } k} X_n$. Note that $\bigcup_{n \text{ at or below } i} X_n = B_j \cup B_k \cup X_i$. However, since $X = B_j \cap B_k$, this is simply $B_j \cup B_k$. Therefore, $\bigcup_{n \text{ at or below } i} X_n = B_j \cup B_k = B_i$, as desired.
4. MSO Formulas

Now we come to the definition for an MSO formula in Agda. First, we give a more formal definition of MSO than we did in the introduction. A question formulated in MSO is referred to as an MSO sentence. We define $MSO(\Sigma)$, the set of all MSO sentences over a given Signature $\Sigma$, as follows:

- $\forall$ $p$-ary relations $R$ and constants $c_1, c_2, ... c_p \in \Sigma, R(c_1, c_2, ... c_p) \in MSO(\Sigma)$. These are the atomic formulas.
- If $\phi, \psi$ are in $MSO(\Sigma)$, then $\neg \phi, \phi \lor \psi, \phi \land \psi$, are all in $MSO(\Sigma)$.
- If $\phi \in MSO(\Sigma \cup c)$ for some constant $c$, then $\forall c \phi$ and $\exists c \phi$ are in $MSO(\Sigma)$.
- If $\phi \in MSO(\Sigma \cup R)$ for some unary relation symbol $R$, then $\forall R \phi$ and $\exists R \phi$ are in $MSO(\Sigma)$. Note that the unary here is the "monadic" part of MSO: we are only allowed to quantify over sets.

Here is our code below, followed by an explanation:

```agda
data Terms (Σ : Signature) : List Tp → Type where
  [] : Terms Σ []
  _,t_ : ∀ {τ ts} → (i τ) ∈ Σ → Terms Σ ts → Terms Σ (τ :: ts)

data Formula (Σ : Signature) : Type where
  ∀i ∃i : (τ : Tp) → Formula (Σ, i τ) → Formula Σ
  ∀p ∃p : (τ : Tp) → Formula (Σ, r (τ :: [])) → Formula Σ
  _∧_, _∨_, _¬_ : Formula Σ → Formula Σ → Formula Σ
  ⊤ ⊥ : Formula Σ
  R ¬ R : ∀ {τs} → (r τs) ∈ Σ → Terms Σ τs → Formula Σ
```

Our encoding of formulas Agda follows the definition above pretty exactly: there is one constructor for every case in the definition, plus an extra constructor for the special formulas $\top$, $\bot$ which correspond to a notion that will become clear
when we define games in a later section. The first constructor corresponds to
the case in the mathematical definition where we have $\phi \in \{\exists c \psi, \forall c \psi\}$. It takes a
$\phi \in MSO(\Sigma \cup c)$, i.e. the $c$-expansion of $\Sigma$ of for some constant $c$, and turns it into
a $\Sigma$ formula. The second constructor corresponds to the case in the mathematical
definition where $\phi \in MSO(\Sigma \cup R)$, i.e. the $R$-expansion of $\Sigma$ of for some relation
$R$. This case does a similar thing to what the first did, but with relations, except
it makes the restriction that the added relation can only take one argument, i.e. it
must be a set. This is done in the code by specifying that the $R$ in $MSO(\Sigma \cup R)$
does not have an arbitrary list of argument types $\tau$s but rather a one-element
list of argument types: $r$ (the relation) has type $\tau :: []$. The third constructor
corresponds to creating a formula $\phi \land \psi$ or $\phi \lor \psi$ out of two formulas $\psi$ and $\phi$,
as in the definition. Then, we add the special formulas corresponding to true and
false, i.e. $\top$ and $\bot$.

Now we come to the most interesting case for the code, the atomic relation
formulas. We introduce a datatype called Terms in order to make the definition
work. Terms has two constructors, $[]$ (empty) and $,t$ (add one term). In the add
one term case, we see that Terms $\Sigma \tau$s makes sure that there are symbols in $\Sigma$
that have type $\tau$ for all $\tau \in \tau$s. In other words: recall that a Signature $\Sigma$ is a list
of symbols, or placeholders, each of a certain type $\tau$. Terms ensures that there are
actually placeholders in $\Sigma$ that want elements of type $\tau$ for all $\tau \in \tau$s. Then the
atomic formula constructor of Formula takes a tuple of set elements, $r$ of type $\tau$s,
along with evidence that there is a relation in $\Sigma$ that takes tuples of type $\tau$s, and
out of these elements, creates a formula. This ensures that $R(a_1...a_n)$ is formula
of $\Sigma$ only if $R \subseteq \{\tau_1...\tau_n\}$ and $a_1...a_n$ has type $\tau_1...\tau_n$.

Note that we do not create a not constructor. Instead, we have a function
*(\varphi) that takes a formula $\varphi$ to its negation, i.e. the dual.
At this point, we have defined the most basic, fundamental building blocks of Courcelle’s Theorem, and their interpretations in Agda. Now let’s build upon them, and make the definitions required to prove the theorem.

5. Relating Structures: Isomorphisms, Compatibility, Union

5.1. Isomorphisms of Structures. Now that we understand what structures are, we need to define a notion of isomorphism between structures. The mathematical definition for isomorphisms between structures, taken again from Kneis et al.[19], is as follows:

Two structures $A$ and $B$ are isomorphic if they are over the same signature $\Sigma$ and there is a bijection $h : A \rightarrow B$ between the underlying sets $A$ and $B$, such that

- $\forall$ constants $c \in \Sigma, c^A = \text{nil}$ if and only if $c^B = \text{nil}$.
- $h(c^A) = c^B$ for all constants $c \in \Sigma$.
- $\forall p$-ary relation symbols $R$ in $\Sigma$ and $a_1, a_2, ... a_p \in A$,
  
  $$(a_1, a_2, ... a_p) \in R_A \text{ if and only if } (h(a_1), h(a_2), ... h(a_p)) \in R_B.$$

In order to formalize this notion of isomorphism, which we call $\text{iso}$, in Agda, we make a helper function, $\text{preserves}$, which defines a function $f$ that satisfies the conditions of $h$ above—except the condition that $f$ is a bijection. Then our definition of $\text{iso}$ states that there is an isomorphism between two structures $A_1$ and $A_2$ is a function $f$ with the evidence that $\text{preserves } A_1 A_2 f$ holds, and also $f$ has an appropriate inverse. This ensures that $\text{iso } A_1 A_2$ holds only if there is a function $f$ between them that satisfies all the conditions of $h$, as desired.

$\text{preserves} : \forall \{\Sigma \text{ oc1 oc2}\} (A_1 : \text{Structure oc1 } \Sigma) (A_2 : \text{Structure oc2 } \Sigma)$

$$(f : \forall \{\tau\} \rightarrow \text{IndividS (fst A1) } \tau \rightarrow \text{IndividS (fst A2) } \tau) \rightarrow \text{Type }$$

$\text{preserves (A1 , []) (A2 , []) f = Unit}$
preserves (A1, s1, is x) (A2, s2, is xx) f = preserves (A1, s1) (A2, s2) f × (fst (f x) ≡ fst xx)
preserves (A1, s1, is x) (A2, s2, none) f = Void
preserves (A1, s1, none) (A2, s2, is x) f = Void
preserves (A1, s1, none) (A2, s2, none) f = preserves (A1, s1) (A2, s2) f
preserves (A1, s1, rs U1) (A2, s2, rs U2) f = ∑ (λ : preserves (A1, s1) (A2, s2) f)
        → (v : IndividsS A1 _)
        → (U1 v → U2 (mapIndividS f v))
        × U2 (mapIndividS f v) → U1 v)

Let’s see how the Agda definition preserves corresponds to the definition of isomorphism above, except for the condition that h (f) is a bijection. We prove this by induction over the signature Σ, which corresponds to the proof in the code: each condition in the definition above corresponds to one (or two) cases in the code.

As a base case, we have that both A1 and A2 are empty structures, i.e. they are structures over an empty Signature, so all of the conditions of h are satisfied vacuously.

Consider the cases in the code where the last symbol in the Signature is a constant symbol, but one structure interprets that symbol as nil and the other does not. (preserves (A1, s1, is x) (A2, s2, none) f = Void and preserves (A1, s1, none) (A2, s2, is x) f = Void in the code.) Void means false in Agda, so these cases in the code say that f cannot exist between two such structures. This corresponds to the part of the mathematical definition which stipulates ∀ constants c ∈ Σ, c^A = nil if and only if c^B=nil.

Consider the case in the code where the last symbol in the Signature is a constant symbol, and both structures interpret that symbol as elements of the underlying set. (preserves (A1, s1, is x) (A2, s2, is xx) f =
preserves \((A_1, s_1) (A_2, s_2) f \times (\text{fst (f x) } \equiv \text{fst xx})\) This case corresponds to the part of the mathematical definition that stipulates \(h(c^A) = c^B\) for all constants \(c \in \Sigma\). We see this because preserves is a recursive call along with a proof that \((\text{fst (f x) } \equiv \text{fst xx})\), i.e. \(h(c^A_1) = c^A_2\) for the last symbol in the Signature. The recursive call ensures that \(h(c^A_1) = c^A_2\) for all symbol interpretations in \((A_1, s_1)\) and \((A_2, s_2)\), and the proof \((\text{fst (f x) } \equiv \text{fst xx})\) ensures that the condition holds for the last symbol in the Signature.

Now consider the case in the code where the last symbol in each Signature is a relation tuple. In this case the code requires a proof that that the third condition of the mathematical definition holds for the relation tuples \(U_1 = R^A_1\) and \(U_2= R^A_2\), so it clearly corresponds to the third part of the mathematical definition of isomorphism. Thus, all requirements in the mathematical definition are mapped to different parts of the code.

\[
\text{iso : } \forall \{ \Sigma \text{ oc1 oc2} \} (A_1 : \text{Structure oc1 } \Sigma) (A_2 : \text{Structure oc2 } \Sigma) \rightarrow \text{Type}
\]

\[
\text{iso A1 A2 } = \Sigma \lambda (f : \forall \{ \tau \} \rightarrow \text{IndividS} (\text{fst A1} \) \tau \rightarrow \text{IndividS} (\text{fst A2} \) \tau)
\rightarrow \text{preserves A1 A2 f } \times (\forall \tau \rightarrow \text{IsEquiv} (f \{ \tau \})))
\]

Finally, we get to the actual Agda definition for an isomorphism between two structures \(A_1\) and \(A_2\). iso in Agda is a product type of the function \(f\) between set elements of \(A_1\) and \(A_2\) such that preserves \(A_1\) \(A_2 f\), i.e. that \(f\) satisfies all the conditions that preserves outlines; and a proof that \(f\) has an inverse. (This is the IsEquiv function.) So, we see that the Agda construction of isomorphism satisfies all the necessary properties. The helper function, preserves, ensures that \(\forall \text{ constants } c \in \Sigma, c^A = \text{nil}\) if and only if \(c^B = \text{nil}\) by returning void when this condition is not satisfied; and it ensures \(f(c^A) = c^B\) for all constants \(c \in \Sigma\) and \(\forall p\)-ary relation symbols \(R\) in \(\Sigma\) and \(a_1, a_2, ..., a_p \in A, (a_1, a_2, ..., a_p) \in R_A\) if and only if \((f(a_1), f(a_2), ..., f(a_p)) \in R_B\) by proofs it requires of the function \(f\) upon
the `IndividS` and relation expansions. And `iso` adds the condition that `f` must be a bijection.

5.2. Compatibility. Although we do not explicitly code a notion of compatibility, or compatible structures in Agda, it is a definition that mathematically underlies other concepts we will encode. Two `Σ`-structures `A_1` and `A_2` are compatible if:

- For all constants `c_i` such that `c_i^{A_1} ≠ nil` and `c_i^{A_2} ≠ nil`, then `c_i^{A_1} = c_i^{A_2}`
- The identity function is an isomorphism between `A_1[A_1 \cap A_2]` and `A_2[A_1 \cap A_2]`.

5.3. Union of Structures. For reasons that will become clear later, we define a notion of union over two structures `A_1` and `A_2`. If two `Σ`-structures `A_1` and `A_2` are compatible, we let the `Σ`-structure `A_1 ∪ A_2` be defined as follows. `A_1 ∪ A_2` has

- An underlying set `A := A_1 ∪ A_2`
- interpretations `R^{A_1∪A_2} := R^{A_1} ∪ R^{A_2}` for all relation symbols `R ∈ Σ`,
- for constants `c ∈ Σ`, `c^{A_1∪A_2} = c^{A_i}` if `c ∈ interpreted(A_i)` for some `i ∈ 1, 2`, otherwise `c^{A_1∪A_2} = nil`.

6. Games

Kneis et al. use games, instead of automata, or fly-automata, or another method, to determine if a given MSO sentence holds on a given structure[19]. Model checking games, or `MC` games, are based off of pebble games, which we define below.

**Definition 6.1. Pebble Game[19]** *Taken from Kneis et al.*
A pebble game \( G = (P, M, P_0, P_1, p_0) \) between two players, say Player 0 and Player 1, consists of a finite set \( P \) of positions, two disjoint sets \( P_0, P_1 \subset P \) which assign positions to the two players, an initial position \( p_0 \in P \), and an acyclic binary relation \( M \subset P \times P \), which specifies the valid moves in the game (i.e., how to get from one position to another). We only allow moves from positions assigned to one of the two players, i.e. we require \( p \in P_0 \cup P_1 \forall (p, p') \in M \). On the other hand, we do allow that positions without outgoing moves are assigned to players. Let \( |G| := |P| \) be the size of \( G \). At each round of play \( i, 1 \leq i \leq l \), the player assigned to position \( p_i \) has to pick a valid next position, i.e. a \( p_{i+1} \) such that \( (p_i, p_{i+1}) \in M \). A play of \( G \) is a maximal sequence \( (p_0, \ldots, p_l) \) of positions \( p_0, \ldots, p_l \in P_0 \cup P_1 \), such that \( (p_i, p_{i+1}) \in M \), i.e. all positions adjacent in the sequence have a valid move between them. Since \( M \) is acyclic, such a play is finite, and is said to have \( l \) rounds and end in position \( p_l \). If \( p_l \in P_0 \), Player 1 wins, and if \( p_l \in P_1 \), Player 0 wins. If \( p_l \notin P_0 \cup P_1 \) then the game is a draw. The object of the game is to force the opposite player into a place where they cannot move. A player is said to have a winning strategy on a game \( G \) if and only if they can win the game on every play, no matter what the choices of the other player. Player \( i \) has a winning strategy on a game \( G \) if and only if

- \( p_0 \in P_i \) and there is a move \((p_0, p_1) \in M\), for some choice of \( p_1 \) such that Player \( i \) has a winning strategy on \( \text{subgame}_G(p_1) \), or
- \( p_0 \in P_{1-i} \) and Player \( i \) has a winning strategy on \( \text{subgame}_G(p_1) \) for all valid choices of \( p_1 \).

A game \( G \) is said to be determined if either one of the players has a winning strategy, otherwise \( G \) is undetermined. The game returns either some indication either Player 1 or Player 2 has a winning strategy, or a proof that the game is a draw.
6.1. Model Checking Games. MC games are pebble games specific to a given formula and structure. In the case of MC games, we call one player the falsifier, who wants to show that the formula is false on the given structure, and one player the verifier, who wants to show that the formula is true on the given structure.

Definition 6.2. Model Checking Game Taking from Kneis et al.

A model checking game $MC(\mathcal{A}, \phi) = (P, M, P_f, P_v, p_0)$, over a $\Sigma$-structure $\mathcal{A}$ that fully interprets $\Sigma$, and $\phi \in MSO(\Sigma)$, is defined by induction over $\phi$ as follows: we define $p_0 = (\mathcal{A}[\bar{c}^\mathcal{A}], \phi)$, where $\bar{c}$ = all constants $c \in \Sigma$.

If $\phi$ is an atomic formula, i.e. $R(a_1, a_2, \ldots a_p)$ or $\neg R(a_1, a_2, \ldots a_p)$ for some $R \in \Sigma$, $(a_1, a_2, \ldots a_p) \in A^p$, then $MC(\mathcal{A}, \phi) = (p_0, \emptyset, P_f, P_v, p_0)$, and $p_0 \in P_f$ if the atomic formula is true (which will make $P_f$, the falsifier, lose) and $p_0 \in P_v$ if the atomic formula is false (which will make $P_v$, the verifier, lose).

If $\phi \in \forall R\psi, \exists R\psi$ for some relation $R$, then let $A_u = (A, U)$ for $U \subset A$ be the $(\Sigma, R)$-expansion of $\mathcal{A}$ with $R^\mathcal{A}U = U$, and let $MC(\mathcal{A}_U, \psi) = (P_U, M_U, P_{f,U}, P_{v,U}, p_U)$ be the corresponding model checking games over $\mathcal{A}_U$ and $\psi$. Then $MC(\mathcal{A}, \phi) = (P, M, P_f, P_v, p_0)$ where

- $P = p_0 \cup \bigcup_{U \subset A} P_U$
- $M = \bigcup_{U \subset A} (M_U \cup p_0, p_U)$
- $P_f = P'_f \cup \bigcup_{U \subset A} P_{f,U}$, where $P'_f = p_0$ iff $\phi = \forall R\psi$ and $\emptyset$ otherwise, and similarly
- $P_v = P'_v \cup \bigcup_{U \subset A} P_{v,U}$, where $P'_v = p_0$ iff $\phi = \exists R\psi$ and $\emptyset$ otherwise

If $\phi \in \forall c\psi, \exists c\psi$ for some nullary symbol $c$, then let $A_a = (A, a)$ for $U \subset A$ be the $(\Sigma, c)$-expansion of $\mathcal{A}$ with $c^\mathcal{A}u = a \in A$, and let $MC(\mathcal{A}_a, \psi) = \ldots$
(P_a, M_a, P_{f, a}, P_{v, a}, P_0) be the corresponding model checking games over A and ψ. Then \( MC(A, \phi) = (P, M, P_f, P_v, P_0) \) where

- \( P = P_0 \cup \bigcup_{a \in A} P_a \)
- \( M = \bigcup_{a \in A} (M_a \cup p_0, p_a) \)
- \( P_f = P'_f \cup \bigcup_{a \in A} P_{f, a} \), where \( P'_f = P_0 \) iff \( \phi = \forall c \psi \) and \( \emptyset \) otherwise, and similarly
- \( P_v = P'_v \cup \bigcup_{a \in A} P_{v, a} \), where \( P'_v = P_0 \) iff \( \phi = \exists c \psi \) and \( \emptyset \) otherwise

If \( \phi \in \psi_1 \lor \psi_2, \psi_1 \land \psi_2 \), then let \( MC(A, \psi) = (P, M, P_{f, \psi}, P_{v, \psi}, P_{0, \psi}) \) be the corresponding model checking games over \( A_{\psi} \) and \( \psi \in \psi_1, \psi_2 \). Then \( MC(A, \phi) = (P, M, P_f, P_v, P_0) \) where

- \( P = P_0 \cup \bigcup_{\psi \in \psi_1, \psi_2} P_{\psi} \)
- \( M = \bigcup_{\psi \in \psi_1, \psi_2} (M_\psi \cup p_0, p_\psi) \)
- \( P_f = P'_f \cup \bigcup_{\psi \in \psi_1, \psi_2} P_{f, \psi} \), where \( P'_f = P_0 \) iff \( \phi = \psi_1 \land \psi_2 \) and \( \emptyset \) otherwise, and similarly
- \( P_v = P'_v \cup \bigcup_{\psi \in \psi_1, \psi_2} P_{v, \psi} \), where \( P'_v = P_0 \) iff \( \phi = \psi_1 \lor \psi_2 \) and \( \emptyset \) otherwise

In our Agda formulation, we take the game-theoretic nature out of the algorithm. We do not have players and games per se. We treat games essentially as trees that map out different possible interpretations of formula variables over a given structure. At each node there is a formula constructor, and each branch connects that node to all its possible sub-formulae on the given structure. For example, if there is a \( \forall x \) in the formula, where \( x \) is a free constant variable, then there will be a node in the tree corresponding to \( \forall x \), and each branch coming from that node will correspond to a different possible interpretations of \( x \) from the available structure. If there is an "and" in the formula, then there will be an "and" node that has branches to each of its sub-formulae.
It has been proven that the verifier has a winning strategy on a model-checking game \( MC(A, \phi) \) if and only if \( A \models \phi \) [20].

The Agda formulation of \( MC \) games is an operation on structures and formulas we call \( \models c \), or "closed truth". Note that instead of \( \models c \) being a game, we simply say that for a Structure \( (A, SA) \) and a formula \( \phi \), \( (A, SA) \models c \phi \) holds if the verifier of an \( MC \) game \( MC((A, SA), \phi) \) has a winning strategy. If the falsifier has a winning strategy, \( (A, SA) \models c \phi \) fails, and \( (A, SA) \models c (\neg \phi) \) succeeds. This is how we take the game-theoretic element out of the algorithm; we simply model an object very similar to the definition of game given in the code. Other than this, however, the code, below, follows the definition of \( MC \) games pretty exactly.

We have to trudge through some background code before we can give the definition of \( \models c \) directly. We define three helper functions, \textit{Value}, \textit{get}, and \textit{gets}, which help us pick out actual elements from the underlying set to fill our symbols with when needed, that is, when we need to check that \( (c_1^A, \ldots, c_1^A) \in R^A \) for some atomic formula \( R(c_1 \ldots c_m) \), or that \( (c_1^A, \ldots, c_1^A) \notin R^A \) for \( \neg R(c_1 \ldots c_m) \), as the definition for \( MC \) games specifies.

\[
\text{Value} : \text{Subset} \to \text{SigThing} \to \text{Type}
\]

\[
\text{Value} A (i \tau) = \text{IndividS} A \tau
\]

\[
\text{Value} A (r \tau s) = (\text{IndividS} A \tau s \to \text{Type})
\]

\text{Value} takes a \text{Subset}, the underlying set of the structure in question, and a \text{SigThing}, i.e. a symbol. In the case that it is given a constant symbol, it returns the interpretation of that symbol. In the case that it is given a relation symbol, it returns a function that checks to see if a given tuple of the correct type is in the relation.
get : \forall \{ \Sigma \} \{ \text{st} : \text{SigThing} \}
\rightarrow \text{st} \in \Sigma
\rightarrow (A : \text{Structure Closed } \Sigma)
\rightarrow \text{Value} (\text{fst } A) \text{ st }
get i0 (A , (\_ ,\_ ,\_ ,\_ ,a)) = a
get i0 (A , (\_ ,\_ ,\_ ,\_ ,rs rel)) = rel
get (iS i1) (A , SA ,\_ ,\_ ,\_ ,\_ ) = get i1 (A , SA)
get (iS i1) (A , SA ,\_ ,\_ ,\_ ,\_ ) = get i1 (A , SA)

Now, get takes a SigThing (symbol) in a Signature, a closed structure on that same Signature, and returns one of the outputs of Value, listed above. The i0 cases are when the SigThing st we are looking for is the last element of the Signature, and the iS i1 cases are when we have to look further within the Signature, \Sigma, for st.

gets : \forall \{ \Sigma \} \{ \tau s : \text{List } Tp \}
\rightarrow \text{Terms } \Sigma \tau s
\rightarrow (A : \text{Structure Closed } \Sigma)
\rightarrow \text{IndividsS} (\text{fst } A) \tau s
gets [] A = < >
gets (x ,t \tau s) A = (\text{gets } \tau s A) , (\text{get } x A)

gets takes Terms \Sigma \tau, i.e. a tuple of constant symbols in \Sigma, a closed structure, and it returns a tuple of set elements from the underlying set of the structure that are of the correct type.

Finally, we can come to our code for closed truth. \models \phi requires a \Sigma-structure A (i.e. (A , SA)) and \Sigma-formula \phi. The input \Sigma-structure needs to be closed (recall
the datatype OC defined way above: this means, as is required in the definition of MC games, that no constants in Σ are interpreted as nil in A).

\[ \vdash _c \] : \forall \{\Sigma\} \to \text{Structure Closed} \to \Sigma \to \text{Formula} \to \Sigma \to \text{Type} \\
(A, SA) \vdash _c \forall \tau \psi = (a : \text{IndividS A} \tau) \to (A, (SA, is a)) \vdash _c \psi \\
(A, SA) \vdash _c \exists \tau \phi = \Sigma \lambda (a : \text{IndividS A} \tau) \to (A, (SA, is a)) \vdash _c \phi \\
(A, SA) \vdash _c \forall p \tau \phi = (P : \text{Unit} \times \text{IndividS} A \tau \to \text{Type}) \to (A, (SA, rs P)) \vdash _c \phi \\
(A, SA) \vdash _c \exists p \tau \phi = \Sigma \lambda (P : \text{Unit} \times \text{IndividS} A \tau \to \text{Type}) \to (A, (SA, rs P)) \vdash _c \phi \\
A \vdash _c (\phi_1 \land \phi_2) = A \vdash _c \phi_1 \times A \vdash _c \phi_2 \\
A \vdash _c (\phi_1 \lor \phi_2) = \text{Either} (A \vdash _c \phi_1) (A \vdash _c \phi_2) \\
A \vdash _c \top = \text{Unit} \\
A \vdash _c \bot = \text{Void} \\
A \vdash _c (R \text{ rel xs}) = \text{get rel A (gets xs A)} \\
A \vdash _c (\neg R \text{ rel xs}) = (\text{get rel A (gets xs A)}) \to \text{Void} \\

We now show that \((A, SA) \vdash _c \phi\) holds if the verifier of an MC game \(MC((A, SA), \phi)\) has a winning strategy. We start from the trivial cases: \(A \vdash _c \top\) and \(A \vdash _c \bot\). Recall that \(\top\) is the special game for which the verifier always has a winning strategy, and \(\bot\) is a similar thing for the falsifier. In our code, \(A \vdash _c \top\) is defined to be Unit, which is equivalent to true in our code, and \(A \vdash _c \bot\) is set to Void, which means that \(A \vdash _c \bot\) is false, it does not hold. So we see on these simple cases, \(\vdash _c\) holds only if the verifier has a winning strategy.

Now we move onto existentials and universal formulas over constants. If \(\phi = \forall c \psi\), for some constant symbol \(c\), then in order for \((A, SA) \vdash _c \phi, (A, (SA, is a)) \vdash _c \phi\) needs to hold for all \(a \in A\) (for all \(a : \text{IndividS} A \tau\)). This corresponds to the fact that in order for the verifier to have a winning strategy on \(MC(A, \forall i \psi)\) there needs to be a winning strategy on \(MC(A_A, \psi)\) for all \(a \in A\), where \(A_A\) is the
(Σ, c) expansion structure of A. (Since on universal formulas, the falsifier would go first.)

If φ = ∃cψ, for some constant symbol c, then in order for (A, SA) ⊨ c φ, there needs to exist some a ∈ A, i.e. some a : IndividS A τ, such that (A, (SA, is a)) ⊨ c φ.

(The Σ λ ((a...) → in the code means "there exists an a such that"). This, in turn, corresponds to the fact that in order for the verifier to have a winning strategy on MC(A, ∃cψ), there needs to be at least one a ∈ A, one next move, such that the verifier has a winning strategy on the subgame MC(Aa, ψ). A very similar explanation works with the cases φ ∈ ∀Rψ, ∃Rψ where R is written in the code as p.

Now consider the case where the formula is atomic, i.e. R(c1, c2,...cp) or ¬R(c1, c2,...cp) for some R ∈ Σ. In the first case, in order for the verifier to win, the formula needs to be true, i.e. that (c1A,...cpA) ∈ RA. And, in order for the verifier to win, this step needs to be the turn of the falsifier, so that the falsifier gets trapped. Since we have done away with the notion of playing a game, we just keep the condition that the formula is true, i.e. (c1A,...cpA) ∈ RA. The corresponding constructor (R rel xs) in the code is exactly a proof that (c1A,...cpA) ∈ RA. Firstly, rel is a symbol that corresponds to a relation and xs corresponds to the tuple of nullary constants (c1,...cm). gets xs A is an IndividS, i.e. a list of set elements of A which interpret the constant symbols specified in xs, and get rel A (gets xs A) checks to make sure that this list of set elements is actually inside rel, the interpretation of the relation symbol R in A. So, get rel A (gets xs A) holds only if, indeed, (c1A,...cpA) ∈ RA, and so A ⊨ c (R rel xs) only holds if this is true. A similar argument works for the negation case, but with the constructor only holding if the elements are not in R.
In the case where $\phi = \psi_1 \land \psi_2$, the falsifier would go first, so the verifier needs winning strategies on both subgames, i.e. $A \models \varphi$ requires that both $A \models c_1 \varphi_1$ and $A \models c_2 \varphi_2$.

### 6.2. EMC Games

Model checking games are all well and good, but some structures have nils in them, and MC games do not accept that. More pointedly, MC games on their own do not possess the qualities that we need them to have in order for them to be useful in giving an algorithm for Courcelle’s Theorem. Recall our explanation of the overall algorithm from Chapter 4. Essentially, in order to prevent exponential blow up for the algorithm at every quantifier (as occurs on the automata implementation of Courcelle’s Theorem), we never want to check a quantifier over the entire set of nodes and edges in a graph. Instead of looking at the whole graph at once, the algorithm in Kneis et al. looks at it in pieces: for a structure $A$, the pieces are the restrictions $A[A_i]$ associated with each node of the tree decomposition. (Recall that for a node $i$ of a tree decomposition, we let $A_i = \bigcup_{j \text{ at or below } i} X_j$, and $A[A_i]$ is the restriction of $A$ to that subset.)

The algorithm that we use to prove Courcelle’s theorem traverses the given tree decomposition bottom up, checking to see if the formula holds on $A[A_i]$ for each node, starting at the empty leaves, and adding a little bit of the set at a time.

It may be hard to believe that this algorithm returns the correct result. First of all, the subsets $A_i$ do not fully interpret the signature, and secondly, how do we know that if there is a winning strategy on one of these smaller games, it propagates to a winning strategy on the overall game? To satisfy the first point, we need games that can handle nil assignments, which is the first handy property of Extended Model Checking games, or EMC games. Additionally, consider the idea that, if you can satisfy an equation with a structure full of nil assignments, you can satisfy that equation on a structure with any interpretations for those
 Assignments, because the elements that are interpreted as nil are not doing anything consequential if they don’t prevent the MSO formula from being satisfied. This is because if the constants were important, if some \( c_i^A \) were interpreted as nil came up in some atomic formula \( R(c_1, ..., c_n) \), at the end of the game, then the game would be undefined by definition. But if the game comes back true, this means that that particular constant had nothing to do with the formula at all, since this situation never arose. So having a winning strategy on some game made with a restricted structure, chock full of nils as it may be, should propagate to having a winning strategy on the full game, since giving specific values to constants that weren’t affecting the formula anyway is not going to change anything. But even with this fact, we still have to come up with a way of propagating the winning strategy upward. We have to find a method of seamlessly transitioning from a game on a restricted structure to a game on the a larger restriction, or the whole structure.

Consider the fact that for each node \( i \) in a tree-decomposition \( T \) of a structure \( A \), while we have \( A_i = \bigcup_j X_j \) at or below \( i \), we can also consider \( A \setminus A_i := B_i \), the set of everything in \( A \) not yet taken into account by the tree decomposition. This set is often called, illustratively, the future of \( A_i \). Just as we can restrict \( A[A_i] \), we can define a structure \( A[B_i] \). Recalling our notion of the union of two structures from earlier, see that at any node \( i \), \( A = A[A_i] \cup A[B_i] \). So, the condition we want is that, if we have a winning strategy on a game for \( A[A_i] \) and \( \phi \), then we want to propagate this to a winning strategy on \( A[A_i] \cup A[B_i] \) and \( \phi \). Thankfully, Kneis et al. created EMCs with the property that are well-defined on taking the union of structures (provided the structures are compatible)\(^{[19]}\).

To summarize, Kneis et al. developed another form of model checking game, which does allow structures with nils, and is well defined over taking unions of
compatible structures, i.e. if there is a winning strategy for a game over $A$ and $\phi$, that translates to a winning strategy on $A \cup B$ and $\phi$ for two compatible structures $A$ and $B$. In the Kneis et al. paper, an EMC is defined to keep track of what bag of the tree decomposition it is on. We found this unnecessary in the code, but provide the original definition here for clarity.

**Definition 6.3. Extended Model Checking Game** [19] *Taken from Kneis et al.*

An extended model checking game $EMC(A, X, \phi) = (P, M, P_f, P_v, p_0)$, over a $\Sigma$-structure $A$, a set $X \subset A$, and $\phi \in MSO(\Sigma)$, is defined over induction over the structure of $\phi$ as follows. Let $p_0 = (A[\bar{c} A \cup X], \phi)$, where again $\bar{c} = $ all constants $c \in \Sigma$. If $\phi$ is an atomic formula, then $EMC(A, X, \phi) = (p_0, \emptyset, P_f, P_v, p_0)$, where $p_0 \in P_f$ if and only if there are no *nils* in the relation tuple and the formula is true; i.e. if the formula is $R(c_1, c_2, ... c_p)$ or $\neg R(c_1, c_2, ... c_p)$ for some $R \in \Sigma$, then each $c_i^A$ in $(c_1^A, ..., c_p^A)$ must not be *nil*, and the formula is actually true, i.e. $(c_1^A, ..., c_p^A) \in R^A$. Again, as explained for $MC$ games, this will make the verifier win. Alternatively, $p_0 \in P_v$ if the tuple $(c_1, c_2, ... c_p)$ is fully interpreted, and the atomic formula is false (which will make $P_v$, the verifier, lose).

If $\phi \in \{\psi_1 \wedge \psi_2, \psi_1 \vee \psi_2, \exists R \psi, \forall R \psi\}$ for some relation $R$, then $EMC$ games are defined exactly the same as $MC$ games above.

If $\phi \in \{\exists c \psi, \forall c \psi\}$ for some constant $c$, then the definition for an $EMC$ game is almost exactly the same as that of an $MC$ game, except that we also add the possibility of extending our structure $A$ with a *nil*, not just elements of $A$. That is, we let $A$ be the $(\Sigma, c)$ expansion of $A$ as before, but this time we let either $c^{A_u} = u \in A$ OR $c^{A_u} = nil$. Let $EMC(A_u, X, \psi) = (P_u, M_u, P_{u_f}, P_{u_v}, p_u)$ be the corresponding $EMC$ game over $A_u, \psi$ as before. Then, $EMC(A, X, \phi) = (P, M, P_f, P_v, p_0)$ where

- $P = p_0 \cup \bigcup_{u \in A \cup \text{nil}} P_u$
CHAPTER 5. DEFINITIONS

6. GAMES

- \[ M = \bigcup_{u \in A \cup \text{nil}} (M_u \cup p_0, p_u) \]
- \[ P_f = P'_f \cup \bigcup_{u \in A \cup \text{nil}} P_{f,u}, \text{ where } P'_f = p_0 \text{ iff } \phi = \forall c\psi \text{ and } \emptyset \text{ otherwise, and similarly} \]
- \[ P_v = P'_v \cup \bigcup_{u \in A \cup \text{nil}} P_{v,u}, \text{ where } P'_v = p_0 \text{ iff } \phi = \exists c\psi \text{ and } \emptyset \text{ otherwise} \] [19]

The way that we model EMC games in the code is through a datatype \( \models_o \), or open truth. Similar to our definition of closed truth above, we do away with the notion of a game and players, and just outline \( \models_o \) so that for a structure \( A \) and a formula \( \phi, A \models_o \varphi \) if and only if there is a winning strategy for the verifier on an EMC game \( EMC(A, X, \phi) \). Before we can get to this definition, however, we have to explain some of the code leading up to it.

\textbf{ValueOpen}: Subset \( \rightarrow \text{SigThing} \rightarrow \text{Type} \)

\begin{align*}
\text{ValueOpen} A (i \tau) &= \text{Maybe} (\text{IndividS} A \tau) \\
\text{ValueOpen} A (r \tau s) &= (\text{IndividsS} A \tau s \rightarrow \text{Type})
\end{align*}

\textbf{getOpen}: \( \forall \{ \Sigma \text{ oc} \} \{ \text{ st : SigThing} \} \)

\begin{align*}
&\rightarrow \text{ st } \in \Sigma \\
&\rightarrow (A : \text{Structure oc} \Sigma) \\
&\rightarrow \text{ValueOpen} (\text{fst} A) \text{ st}
\end{align*}

\begin{align*}
\text{getOpen} i0 (A, (\_ ,\_ , \text{is a})) &= \text{Some} a \\
\text{getOpen} i0 (A, (\_ ,\_ , \text{rs rel})) &= \text{rel} \\
\text{getOpen} i0 (A, (\_ ,\_ , \text{none})) &= \text{None} \\
\text{getOpen} (\text{iS i1}) (A, \text{SA}, \_ ) &= \text{getOpen} i1 (A, \text{SA}) \\
\text{getOpen} (\text{iS i1}) (A, \text{SA}, \text{rs } \_ ) &= \text{getOpen} i1 (A, \text{SA}) \\
\text{getOpen} (\text{iS i1}) (A, \text{SA}, \text{none}) &= \text{getOpen} i1 (A, \text{SA})
\end{align*}

\textbf{getsOpen}: \( \forall \{ \Sigma \text{ oc} \} \{ \text{ ts : List Tp} \} \)

\begin{align*}
&\rightarrow \text{Terms} \Sigma \text{ ts} \\
&\rightarrow (A : \text{Structure oc} \Sigma)
\end{align*}
→ Maybe (IndividsS (fst A) τ)

\[ \text{getsOpen } [] A = \text{Some} <> \]

\[ \text{getsOpen } (x,t xs) A \text{ with } (\text{getsOpen } xs A) | (\text{getOpen } x A) \]

... | Some vs | Some v = Some (vs , v)

... | _ | _ = None

Similar to our code for closed truth, we define ValueOpen, getOpen, and getsOpen, which perform almost the same functions that Value, get, and gets do, except that they all take into account the fact that for open truth and EMC games, we could be working with open structures, which may contain nils. So, ValueOpen, instead of returning an set element that is an interpretation when given a nullary constant, gives either none (for nil) or Some v (for some set element). This propagates through getOpen and getsOpen: getOpen looks up the set element used to interpret the symbol st in the Signature, failing if it finds none. getsOpen, when given Terms (a tuple of constant symbols) and a structure, returns a tuple of elements that interpret the constant symbols—except that, if even one of the symbols is not interpreted, it returns a none value instead of a tuple of set elements.

Differing from what we did for closed truth, we have three constructors for open truth: one for existential formulas, one for universal formulas, and one for atomic formulas. In order to explain this, we need to show some code that leads up to it.

We defined predicates on formulae that determined whether a given formula was existential \((\phi = \exists R \psi, \exists c \psi, \psi_1 \lor \psi_2)\) universal \((\phi = \forall R \psi, \forall c \psi, \psi_1 \land \psi_2)\), or an atomic relation \((R \text{ or } \neg R \text{ for some arity}(R)-tuple of constants)\). These predicates have different constructors for each kind of existential, universal, or atomic formula. For example, \(\text{isE}\) has a constructor for each of \(\{\exists i, \exists R, \lor, \bot\}\).
Based on these different types of formulae, we defined a notion of a branch, of which there are universal branches and existential branches. Think of a branch as an edge between two nodes in a game: it keeps track of which formula and structure the game is are traveling from and towards. (For as the algorithm goes through a game, the formula shrinks and the structure grows). So if there is a node (i.e. a position in the definition above) \((A, \varphi)\) where one possible next move is to \((A', \varphi')\), then there will be a branch \(\text{branch } A \varphi A' \varphi'\) too in the game to show that. We will see further below in our definition of game. Included below is only the universal branch code, as the existential branch code is quite similar. \text{Ubranch} has a different constructor for all the different kinds of branches required for different formulas. For an and node, \(\varphi = \psi_1 \land \psi_2\), we need a branch going to each of \(\psi_1\) and \(\psi_2\), and this is what ufstb and usndb are. If \(\varphi = \forall c \psi\), we need a
branch $A \rightarrow A'$ where $A'$ is a $c$-expansion of $A$ with $a$ interpreted as $c$. Finally, in the case that $\phi = \forall R\psi$, $\text{ubranch}$ creates a branch for all tuples $r$ of elements in $A$ to interpret $R$.

**data** $\text{ubranch} \{ \Sigma_1 : \text{Signature} \} : \forall \{ \Sigma_2 : \text{Signature} \} \{ \text{oc1 oc2} \}$

$\begin{align*}
(A_1 : \text{Structure oc1 } \Sigma_1) (\varphi_1 : \text{Formula } \Sigma_1) \\
(A_2 : \text{Structure oc2 } (\text{fst } A_1) \Sigma_2) (\varphi_2 : \text{Formula } \Sigma_2) \rightarrow \text{Type}
\end{align*}$

$u\text{fstb} : \forall \{ \text{oc} \} \{ A_1 : \text{Structure oc } \Sigma_1 \} \rightarrow \{ \varphi_1 \varphi_2 : \text{Formula } \Sigma_1 \}$

$\rightarrow \text{ubranch } A_1 (\varphi_1 \land \varphi_2) (\text{snd } A_1) \varphi_1$

$u\text{sndb} : \forall \{ \text{oc} \} \{ A_1 : \text{Structure oc } \Sigma_1 \} \rightarrow \{ \varphi_1 \varphi_2 : \text{Formula } \Sigma_1 \}$

$\rightarrow \text{ubranch } A_1 (\varphi_1 \land \varphi_2) (\text{snd } A_1) \varphi_2$

$u\text{forallb} : \forall \{ \text{oc} \} \{ \tau \} \{ A_1 : \text{Structure oc } \Sigma_1 \} \rightarrow \{ \varphi : \text{Formula } (i \tau :: \Sigma_1) \}$

$\begin{align*}
(a : \text{IndividS } (\text{fst } A_1) \tau) \\
& \rightarrow \text{ubranch } A_1 (\forall i \tau \varphi) ((\text{snd } A_1),a) \varphi
\end{align*}$

$u\text{forallnil} : \forall \{ \forall \tau \} \{ \text{oc} \} \{ A_1 : \text{Structure oc } \Sigma_1 \}$

$\rightarrow \{ \varphi : \text{Formula } (i \tau :: \Sigma_1) \}$

$\rightarrow \text{ubranch } A_1 (\forall i \tau \varphi) ((\text{snd } A_1),\text{none}) \varphi$

$u\text{forallr} : \forall \{ \forall \tau \} \{ \text{oc} \} \{ A_1 : \text{Structure oc } \Sigma_1 \} \rightarrow \{ \varphi : \text{Formula } (r (\tau :: []) :: \Sigma_1) \}$

$\begin{align*}
(r : \text{IndividS } (\text{fst } A_1) (\tau :: []) \rightarrow \text{Type}) \\
& \rightarrow \text{ubranch } A_1 (\forall p \tau \varphi) ((\text{snd } A_1),r) \varphi
\end{align*}$

And from notions of existential and universal branches, we get an overarching branch type: it has one constructor for branches from existential formulas ($\text{ebr}$), and one constructor for branches from universal formulas ($\text{ubr}$). We don’t need a constructor for atomic formulas because there will never be a branch from an atomic formula, only a branch to one.

**data** $\text{branch} \{ \Sigma_1 : \text{Signature} \} : \forall \{ \Sigma_2 : \text{Signature} \} \{ \text{oc1 oc2} \}$

$\begin{align*}
(A_1 : \text{Structure oc1 } \Sigma_1) (\varphi_1 : \text{Formula } \Sigma_1)
\end{align*}$
(A2 : StructureS oc2 (fst A1) Σ2)
(φ2 : Formula Σ2) → Type where

ebr : ∀ {Σ2} {oc1 oc2} {A1 : Structure oc1 Σ1} {φ1 : Formula Σ1}
  {A2 : StructureS oc2 (fst A1) Σ2} {φ2 : Formula Σ2}
  → isE φ1 → ebranch A1 φ1 A2 φ2 → branch A1 φ1 A2 φ2

ubr : ∀ {Σ2} {oc1 oc2} {A1 : Structure oc1 Σ1} {φ1 : Formula Σ1}
  {A2 : StructureS oc2 (fst A1) Σ2} {φ2 : Formula Σ2}
  → isU φ1 → ubranch A1 φ1 A2 φ2 → branch A1 φ1 A2 φ2

Finally, we can look at the code for open truth, our Agda encoding of an EMC game. It has four constructors: provesu (for proving universal formulas), provese (for proving existential formulas), provesbase and provesnotbase (both for atomic formulas). As noted when we explained closed truth and MC games, we take the game element out of the EMC game. We model EMC games with a datatype ⊩o such that for a structure A and a formula φ, A ⊩o φ if and only if there is a winning strategy for the verifier on an EMC game EMC(A, X, φ).

data _⊢o_ {oc : _} {Σ : _} (A : Structure oc Σ) : Formula Σ → Type where

provesu : {φ : Formula Σ}
  → isU φ
  → (∀ {Σ' oc'} {A' : StructureS oc' (fst A) Σ'} {φ'}
    → ubranch A φ A' φ' → (fst A , A') ⊩o φ')
  → A ⊢o φ

provese : ∀ {Σ' oc'} {A' : StructureS oc' (fst A) Σ'} {φ : Formula Σ} {φ'}
  → isE φ
  → ebranch A φ A' φ'
  → (fst A , A') ⊩o φ'
  → A ⊢o φ
provesbase : ∀ {τs} {rel} {xs : Terms _ τs} vs
→ ((getsOpen xs A) ≡ (Some vs))
→ (getOpen rel A) (vs) → A ⊩ o (R rel xs)
provesnotbase : ∀ {τs} {rel} {xs : Terms _ τs}
→ (vs : _) → ((getsOpen xs A ≡ (Some vs)))
→ ((getOpen rel A) (vs) → Void)
→ A ⊩ o (¬R rel xs)

We show that ⊩ o holds exactly when a verifier has a winning strategy on an EMC game. Consider the provesbase case in the code, which corresponds to the place in the EMC definition where the formula is \( R(c_1, \ldots, c_m) \) for some relation symbol \( R \) in \( \Sigma \). In the mathematical definition, in order for the verifier to win in this case, the atomic formula needs to be correct, and it should be the falsifier’s turn. Since we have taken the notion of players in the game out, we only need the proof that the atomic formula is true. And this is exactly what the provesbase constructor is: a proof that the formula holds. Given a tuple, \( vs \), of elements in the underlying set (i.e. \( vs : \text{IndividsS A τs} \)) \((\text{getsOpen xs A} ≡ (\text{Some vs}))\) is a proof that the \( xs \), i.e. the tuple \((c_1, \ldots, c_m)\) in the formula \( R(c_1, \ldots, c_m) \) is actually interpreted as this tuple of set elements \( vs \), i.e. there are no nils. We can see this because \( \text{getsOpen} \) takes a list of constant symbols \((\text{Terms } \Sigma \tau s)\) and a structure, and makes sure that each of those symbols is interpreted on that structure (i.e. returns \( \text{Some v in ValueOpen} \)). Then, \((\text{getOpen rel A} \) (vs)\) checks to make sure that \( vs \), the actual tuple of elements of \( A \) (the \( \text{IndividsS-A-τs} \)), is actually in \( R^A \). We see this because \( \text{getOpen} \) will return a \( \text{ValueOpen} \) for this relation \( rel \) on \( A \), i.e. a function that checks if a given tuple is in a given relation, and so in \((\text{getOpen rel A} \) (vs)\) we are using that given function to check if \( vs \in R^A \). A similar proof will show that the provesnotbase constructor is a proof
that $\neg R(c_1, \ldots, c_m)$ holds, i.e. $(c_1^A, \ldots, c_m^A) \not\in R^A$. The only real difference is that in
the \texttt{provesnotbase} constructor, the $((\texttt{getOpen rel A}) (\texttt{vs}) \rightarrow \texttt{Void})$ signifies that the
tuple \texttt{vs} is not in $R^A$.

Now let’s consider the \texttt{provesu} constructor. This maps to the case in the
definition where we have any universal formula: $\phi \in \forall c \psi, \forall R \psi, \psi_1 \land \psi_2$. The
\texttt{provesu} constructor is a proof that

1. The formula is universal (\texttt{IsU} $\phi$)
2. For all $\Sigma'$-structures $A'$ with the same underlying set as $A$, and all $\Sigma'$-
formulas $\phi'$, if there’s a branch (a \texttt{ubranch}, since we’re in the universal
case) between $A$, $\phi$ and $A' \phi'$, then $A' \models \phi'$

Note that depending on what exact formula $\phi$ is, the \texttt{ubranches} from $\phi$ will
branch to different $A'$, $\phi'$—we don’t make a separate constructor for each case. But
if, for example, the formula were $\forall c \psi$ for some constant $c$, then there would be a
\texttt{ubranch} for $A_{\Sigma} a$ for all $a \in A$, where $A_{\Sigma} a$ is a $\Sigma, c$ expansion of $A$ with $c$ interpreted
as $a$ for all $a \in A$. This corresponds to a winning strategy for the verifier because,
since the falsifier gets to choose the next move on a universal quantifier, the verifier
needs a winning strategy on every subgame that the falsifier could move to. We
interpret this as $A' \models \phi'$ for every possible $A'$, $\phi'$ we can branch to from $A$, $\phi$.

A very similar explanation to the one above works for the \texttt{provese} constructor.
In this case, we just need to show that the formula is existential, and that $A' \models \phi'$
for one possible $A'$, $\phi'$ we can branch to from $A$, $\phi$. This corresponds to the verifier
having a winning strategy on a corresponding EMC game because since the verifier
gets to choose the next move on existentials, we only need to find one subgame
that the verifier can win, i.e. where $A' \models \phi'$. Again, the specific subgames that
are checked depend on the \texttt{ebranch} constructor. We also have a definition, $\models \texttt{false}$,
which takes a structure and a formula as before and shows that the formula is
false on that structure. This corresponds to a falsifier having a winning strategy on an analogous EMC game, and we define it as a $\vdash \neg \varphi$ that proves the negation of the formula given.

7. Reduced Games

As described above, the algorithm used to prove Courcelle’s theorem in the Kneis et al.[19] paper runs through an entire EMC game on each node of the tree decomposition, and propagates undecided games upward through the tree decomposition until it has enough data to come to a definitive answer as to whether or not the formula holds on the given structure. In order to further reduce the time of the algorithm, we want to be running games as efficiently as possible. In order to do this, we want to keep as few possible next positions (analogously in our code, as few branches) as possible. This way, we have fewer places to move to at each step, and so our game becomes much smaller. We would like to keep track of only the necessary information as we propagate games upward. For example, if we are able to achieve a winning or losing strategy on a game at a given node, we want to exit the algorithm there since we’ve found out answer. If our game is undecided, we still don’t have to keep all information that is available to us: for example, if our formula is universal, we don’t have to keep track of all of the different subgames that prove the formula true.

7.1. Game Tree. With this in mind, we introduce another game-like structure in the Agda code, which will become the underlying type for something called a reduced game. A reduced game will keep track of only the information from any given game that we absolutely need. The explanation for all of this will come later, but because of the way the code is structured, we will introduce a constructor called $\vdash s$, which we call a raw game tree, now.
data _⊩_ s { oc : _ } { Σ : _ } (A : Structure oc Σ) (φ : Formula Σ) : Type where
leaf : (isR φ) → A ⊩ s φ
node : (bs : Branches A φ)
  → ( ∀ { Σ' } { oc' } { A' : StructureS oc' (fst A) Σ'} { φ' } b 
  → (branchto A' φ' b) ∈ bs → ((fst A , A') ⊩ s φ'))
  → A ⊩ s φ

A ⊩ s is a game that is either atomic, or has a certain set of branches. However, which branches these are is not specified. That is, it does not require the exact set of outward branches (subgames) that open truth or closed truth require. We enclose the code above. See that a ⊩ s is a datatype with two constructors, leaf and node. The leaf constructor is simply a proof that the formula is atomic, isR φ. There are no outward branches, i.e. subgames, here. The node constructor is a product type, consisting of (1) a list of branches (bs) from the given structure A and formula φ (i.e. the game position that has structure A and formula φ), and (2) a proof that for all structure, formula pairs A', φ' (i.e. positions) that any branch bi ∈ bs points toward, A' ⊩ s φ'. We can also think of (2) as a function that takes a branch b, a proof that b is in the list of branches of some game tree, and returns a game tree on the structure that b branches towards.

Note the difference between open truth, closed truth and game trees: as mentioned earlier, there are no specifications on the branches of a game tree—unlike for the open truth and closed truth datatypes, if the formula φ = ψ_1 ∧ ψ_2, there is no requirement in a game tree that there be a branch for both ψ_1 and ψ_2. There’s just some unspecified list of branches; and what goes in that list will become clear later.

Continuing from the discussion above about throwing out unnecessary information in games, how do we decide which next positions (or branches) to keep
and which ones to throw away? To help answer this question, we introduce a concept of equivalent games.

### 7.2. Position Equivalence

Consider a game \( EMC(A, X, \phi) \). Two positions \( p_1, p_2 \) are position-equivalent, denoted by \( p_1 \cong p_2 \) if and only if

- \( p_1 = (H_1, X, \phi) \) and \( p_2 = (H_2, X, \phi) \) for some formula \( \phi \) and set \( X \subseteq H_1 \cap H_2 \).
- There is an isomorphism \( h : H_1 \rightarrow H_2 \) such between \( H_1 \) and \( H_2 \), such that \( h(a) = a \) for all \( a \in X \).

Let’s take a look at the Agda encoding of position equivalence. We define position equivalence with two functions, \texttt{positionEquiv’} and \texttt{positionEquiv}. We discuss \texttt{positionEquiv} first—it may be helpful to look back at the definition and explanation of the function \texttt{iso} (for isomorphism between structures) above, or otherwise just accept that \texttt{iso} called on two structures gives us an isomorphism between them. Also note that \texttt{constants}(A) is the set of elements of \( A_i \) that were used as interpretations of constants in \( A \). (The code is below). \texttt{positionEquiv} takes two structures, \( A_1 \) and \( A_2 \), the bag \( X \) that they share, proof that \( X \subseteq A_1 \) and \( X \subseteq A_2 \) (so that \( X \subseteq A_1 \cap A_2 \)), and proof that \( X \) is decidable (which will go away when we change our definition of subset). \texttt{positionEquiv} holds on these inputs if there is a function \( h \) that satisfies \texttt{iso A1[X \cap constants(A)]} \texttt{A2[X \cap constants(A)]}, i.e. is an isomorphism between \texttt{A1[X \cap constants(A)]} \texttt{A2[X \cap constants(A)]}, such that \( \forall x \in X, h(x) = x \). The little functions such as \texttt{subLUB}, \texttt{constantsDec}, \texttt{unionDec}, and \texttt{promoteIndividS} are all just little fixes to convince Agda that everything is the correct type, in the correct subset, etc. (For example, \texttt{unionDec} is a proof that the union of two decidable sets is decidable.) We won’t go into these functions, but if the reader is curious, the source code is attached.
\[ \text{positionEquiv} : \forall \{ \Sigma \text{oc1 oc2} \} \text{(A1 : Structure oc1 } \Sigma ) \text{(A2 : Structure oc2 } \Sigma ) \]
\[ \text{(X : Subset) (XinA1 : Sub X (fst A1)) (XinA2 : Sub X (fst A2))} \]
\[ \rightarrow \text{DecidableSub } X \]
\[ \rightarrow \text{Type} \]
\[ \text{positionEquiv } A1 \text{ A2 } X \subseteq A1 \text{ X} \subseteq A2 \text{ decX} = \]
\[ \Sigma \lambda (h : \text{iso } (\text{restriction } A1 \text{ (union } (\text{constants } A1) \text{ X})) \]
\[ (\text{unionDec } \{ S1 = \text{constants } A1 \} \{ S2 = X \} \text{ (constantsDec } A1 \text{ )} \text{ decX}) \]
\[ (\text{subLUB } (\text{constantSub A1 X} \subseteq A1)) \]
\[ (\text{restriction } A2 \text{ (union } (\text{constants } A2) \text{ X})) \]
\[ (\text{unionDec } \{ S1 = \text{constants } A2 \} \{ S2 = X \} \text{ (constantsDec } A2 \text{ )} \text{ decX}) \]
\[ (\text{subLUB } (\text{constantSub } A2 \text{ X} \subseteq A2)) \]
\[ \rightarrow \forall \{ \tau \} \text{ (x : IndividS X } \tau) \rightarrow \]
\[ \text{fst } h \text{ (promoteIndividS} \]
\[ (\text{sublNR } \{ A = \text{constants } A1 \} \{ B = X \}) x) \equiv \]
\[ \text{promoteIndividS } (\text{sublNR } \{ A = \text{constants } A2 \} \{ B = X \}) x \]

Now we must show why this code is a sufficient definition for \text{positionEquiv} above—certainly, at first glance, it seems as if it may fall short because we don’t have an isomorphism over the entire structure but only the structure restricted to those elements that have been interpreted as constants and the relevant bag, X. But note that this is simply a difference in labeling, not of content. Recall that the position \( p_0 \) of a game \( EMC(A, X, \phi) = (P, M, P_f, P_v, p_0) \) is defined as \( p_0 = (A[\overline{c} \cup X], \phi) \), where \( \overline{c} = \text{all constants } c \in \Sigma \). Note that the structure at a position is by definition the overall structure restricted to the set elements that are interpreted as constants along with the relevant bag \( X \). However, in the Agda code, we don’t have a specific type for positions; we just have \( \models c \) and \( \models o \) which look like trees with branches between structures, with the entire structure as a
CHAPTER 5. DEFINITIONS

7. REDUCED GAMES

label for each node. So, we take this restriction to $A[\pi^A \cup X]$ only when we define position equivalence, to make it match the mathematical definition, where this restriction is implicit. So, the mathematical definition and the code in fact have the same requirements.

Quickly, we include our code for positionEquiv'; this does exactly the work that positionEquiv does except it changes the accepted input type slightly so that it works more seamlessly with the rest of the code.

positionEquiv' : $\forall \{\Sigma \text{ oc1 oc2}\} (A1 : \text{Structure oc1 } \Sigma) (A2 : \text{Structure oc2 } \Sigma)$

$\rightarrow \text{fixed } (\text{fst A1}) (\text{fst A2}) \rightarrow \text{Type}$

positionEquiv' A1 A2 (X , (X$\subseteq A1$ , X$\subseteq A2$) , decX)

= positionEquiv A1 A2 X X$\subseteq A1$ X$\subseteq A2$ decX

7.3. Game Equivalence. We say that two games $G_1 = (P_1, M_1, p_1), G_2 = (P_2, M_2, p_2)$ are equivalent, denoted $G_1 \cong G_2$, if $p_1 \cong p_2$, and there is a bijection $\pi : \text{subgames}(G_1) \rightarrow \text{subgames}(G_2)$, such that $G' \cong \pi(G) \forall G' \in \text{subgames}(G_1)$.

Now, let’s look at our code for proving that two games are equivalent. We have two functions, gameEquiv and gameEquiv’, defined mutually, that do this job. These functions rely on a helper definition, BranchBijection. BranchBijection, which is enclosed below, does exactly what it sounds like: it is the proof of a bijection between two sets of branches. (The type Branches is simply a list of branches, which we defined earlier).

BranchBijection : $\{\Sigma : \text{Signature}\} \{\text{oc1 oc2 : _}\} (A1 : \text{Structure oc1 } \Sigma)$

$\rightarrow \text{Branches } A1\varphi \rightarrow \text{Branches } A2\varphi \rightarrow \text{Type}$

BranchBijection A1 A2 $\varphi$ branches1 branches2 =

$\forall \{\Sigma' \varphi'\} \rightarrow$
**Equiv** \( \Sigma (\lambda \text{oc}' \rightarrow \Sigma (\lambda (A' : \text{StructureS oc'}) (\text{fst A}1) \Sigma')) \rightarrow \Sigma \lambda b \rightarrow (\text{branchto A' } \varphi' b) \in \text{branches1}) \)

\( (\Sigma (\lambda \text{oc}' \rightarrow \Sigma (\lambda (A' : \text{StructureS oc'}) (\text{fst A}2) \Sigma')) \rightarrow \Sigma \lambda b \rightarrow (\text{branchto A'} \varphi' b) \in \text{branches2}) \)

**BranchBijection** takes two structures \( A_1, A_2 \), a formula \( \phi \), and the list of branches that come out of each structure. **BranchBijection** holds on these elements if there is a bijection between the set of structures \( \bigcup_i A_{1,i} \) that \( A_1 \) branches to, and the set of structures \( \bigcup_i A_{2,i} \) that \( A_2 \) branches to. The type of the result of **BranchBijection** is a product type of a function from \( \bigcup_i A_{1,i} \) to \( \bigcup_i A_{2,i} \), and a proof that that function is a bijection, i.e. an inverse function \( \bigcup_i A_{2,i} \) to \( \bigcup_i A_{1,i} \).

(This comes from the **Equiv** in the code; but we won’t go into an explanation of that).

Now we come to the **gameEquiv’** and **gameEquiv** functions themselves. **gameEquiv** takes two \( \Sigma \)-structures, \( A_1 \) and \( A_2 \), a \( \Sigma \)-formula \( \phi \), a game tree for \( \phi \) on \( A_1 \) and \( \phi \) on \( A_2 \), and a bag \( X \) (this is the fixed \( (\text{fst A}1) (\text{fst A}2) \)). **gameEquiv** holds on these inputs if **positionEquiv’** holds for the two structures and the bag (\( f \) is the bag in the code), and **gameEquiv’** holds on all the inputs as well.

**gameEquiv** : \( \forall \{\Sigma \text{oc1 oc2}\} (A_1 : \text{Structure oc1} \Sigma) (A_2 : \text{Structure oc2} \Sigma) \)

\( (\phi : \text{Formula} \Sigma) \rightarrow A_1 \parallel \text{s} \phi \rightarrow A_2 \parallel \text{s} \phi \rightarrow \text{fixed} (\text{fst A}1) (\text{fst A}2) \rightarrow \text{Type} \)

**gameEquiv A1 A2 \( \varphi \) g1 g2 f = positionEquiv’ A1 A2 f \times gameEquiv’ A1 A2 \( \varphi \) g1 g2 f**

**gameEquiv’** : \( \forall \{\Sigma \text{oc1 oc2}\} (A_1 : \text{Structure oc1} \Sigma) (A_2 : \text{Structure oc2} \Sigma) \)

\( (\phi : \text{Formula} \Sigma) \rightarrow A_1 \parallel \text{s} \phi \rightarrow A_2 \parallel \text{s} \phi \rightarrow \text{fixed} (\text{fst A}1) (\text{fst A}2) \rightarrow \text{Type} \)

**gameEquiv’ A1 A2 \( \varphi \) (leaf x1) (leaf x2) f = Unit**

**gameEquiv’ A1 A2 \( \varphi \) (leaf x1) (node bs x2) f = Void**

**gameEquiv’ A1 A2 \( \varphi \) (node bs1 x1) (leaf x2) f = Void**
gameEquiv' A1 A2 φ (node bs1 x1) (node bs2 x2) f =
\[ \Sigma \lambda (\text{BranchBijection} A1 A2 \phi \text{bs1 bs2}) \rightarrow \]
\[ \forall \{ \Sigma' \text{oc}' \} \{ A' : \text{StructureS oc'} (\text{fst A1}) \Sigma' \} \{ \phi' \} \text{bi} \rightarrow \]
\[ (\text{brnchi} : \text{branchto A' } \phi' \text{ bi } \in \text{bs1}) \rightarrow \]
\[ \text{gameEquiv} ((\text{fst A1}), A') ((\text{fst A2}), \]
\[ (\text{fst (snd (fst b (_ , A' , bi , brnchi))))}) \phi' \text{ (x1 bi brnchi)} \]
\[ (x2 _ (\text{snd (snd (fst b (_ , A' , bi , brnchi))))})) f) \]

gameEquiv' does most of the work of the recursive calls. It takes the same input as gameEquiv, but here we have one case for each of the possible combinations of constructors of the two game trees are made up of. If the game trees are both leaves, there are no subgames (i.e. structures that are branched to) and so no recursive calls are needed, the subgames are equivalent vacuously, and gameEquiv' returns Unit, or True. If one game tree is a leaf and the other is a node, there is no possible way for them to be equivalent, so gameEquiv returns Void, or false. Now the interesting case: both game trees are nodes. Then, gameEquiv' A1 A2 φ (A1 \models s \phi) (A2 \models s \phi) X holds if there exists a BranchBijection b between the list of branches in the (A1 \models s \phi) game tree and the list of branches in the (A2 \models s \phi) game tree, such that for all structures A' (with underlying set A1) and formulas \phi' that A1 and \phi branches to, gameEquiv A' b (A') \phi' A' \models s \phi' b (A') \models s \phi' X holds. Through mutual recursion, the two functions determine if two games are equivalent, with gameEquiv checking the initial positions are equivalent and gameEquiv' continuing to send the correct subgames to gameEquiv to check for position equivalence.

If two games are equivalent, they will have the same outcome[19]. Certainly, for each game G, we only need to keep one representative of each equivalence class of subgames(G) modulo \cong. Additionally, note that for any game G, we can
discard the subgames where the opposing player, which in our case will always be the falsifier, wins. There is no use in checking a path that is known to fail[19].

7.4. Reduced Games Definition. With this in mind, we give a definition of a reduced game. A reduced EMC game $G$ over a $\Sigma$-structure $A$ and a $\Sigma$-formula $\phi$, $\text{reduce}(A, X, \phi)$ is defined by induction as follows:

- If $\phi$ is an atomic formula, then the reduced game is defined exactly as a regular EMC game.

- If $\phi$ is not an atomic formula, then all subgames $G'$ of $G$ must be undecided, and further, $\forall G'_1 \in \text{subgames}(G), \exists G'_2 \in \text{subgames}(G)$ such that $G'_1 \not\cong G'_2$. That is, all subgames come from distinct equivalence classes in the $\cong$ relation.

It may be more instructive to think about how to turn an EMC into a reduced EMC than to simply ponder the definition. Given an EMC, Kneis et al. describe an algorithm for making a reduced EMC $G_{red}$ as follows: if the formula in the game is atomic, treat the reduced EMC game as a regular EMC game. Otherwise, recursively compute all of the reduced subgames of the original EMC, and:

- If the formula is universal, and there is a subgame $G'$ where the falsifier has a winning strategy, then return that the falsifier won. If the falsifier does not have a winning strategy on any subgame, then we define the set of subgames of $G_{red}$, $\text{subgames}(G_{red}) = (G'_1, ..G'_n)$ such that $\forall i G_i$ is undecided, and there does not exist a $j$ such that $G_i \cong G_j$.

- If the formula is existential, and there is a subgame $G'$ where the verifier has a winning strategy, then return that the verifier won. If the verifier does not have a winning strategy on any subgame, then we define the
set of subgames of $G_{red}$, \( \text{subgames}(G_{red}) = \left(G_1', .. G_n'\right) \) such that \( \forall i \ G_i \) is undecided, and there does not exist a \( j \) such that \( G_i \cong G_j \).

So, a reduced game \( G \) is a game that has not been decided yet, but that retains as little information as possible in order to speed up the overall algorithm. It may seem counterintuitive that we can throw away so much information and still get the same answer as an \( EMC \) game, which is what we would like. Let’s think about this a bit. Certainly, we want a game that is undecided, because otherwise we can simply leave the entire algorithm and say we’ve found that the formula is either true or not true on the structure. Once we’ve established that the game is undecided, however, still not all information is necessary. For example, if our formula is universal, we don’t have to keep track of all of the different subgames that prove the formula true. It doesn’t help the future of the algorithm to keep track of the fact that one particular interpretation of symbols satisfied the formula, because all the algorithm looks for is one interpretation that does not satisfy the formula. If one subgame in a universal formula returns true, we breathe a sigh of relief and then discard it, and focus on the games that are undecided. Similarly, if a formula is existential, we don’t have to keep track of the subgames that end up being false—because we are just looking for one satisfying interpretation of variables. If the reader can accept this, then all that is left to be convinced of is that the result of a reduced game on a structure \( A \) and a formula \( \phi \) will always be the same as an \( EMC \) game on the same formula and structure. A proof of this fact is on our agenda—we will speak more about what is still to be done in this project in the last chapter.

Now we transition to explaining our encoding of a reduced game. There are a few helper functions for this definition. The moving parts are: \( \models s \) defined above, which is sort of the underlying type for a reduced game, though not a reduced
game itself. We can combine this with another definition isRed, a proof that a \( \vdash_s \) game tree is in fact a reduced game. Then, we trivially put these two pieces together in provesR, which is our formulation of a reduced game, i.e. a \( \vdash_s \) that satisfies isRed \( \vdash_s \).

We go through the Agda code for isRed, a proof that a game tree is a reduced game.

\[
isRed : \forall \{ \Sigma \; oc \} \ (A : \text{Structure} \; oc \; \Sigma) \ (\varphi : \text{Formula} \; \Sigma) \to A \vdash_s \varphi \to
\]

\[
(X : \text{fixed1} \ (\text{fst} \ A)) \to \text{Type}
\]

\[
isRed \ A \ \varphi \ \text{(leaf} \ x) \ \text{fix} = \text{Unit}
\]

\[
isRed \ A \ \varphi \ \text{(node} \ x) \ \text{fix} =
\]

\[
(\forall \{ \Sigma' \; oc' \} \ \{A' : \text{Structure}S \; oc' \ (\text{fst} \ A) \; \Sigma') \ \{\varphi'\} \ bj
\]

\[
\to (\text{prfbr} : \text{branchto} \ A' \ \varphi' \ bj \in \text{bs})
\]

\[
\to \text{isRed} \ (\text{fst} \ A, A') \ \varphi' \ (x \ bj \text{prfbr} \text{fix})
\]

\[
\times
\]

\[
(\text{isU} \ \varphi \to \forall \{ \Sigma' \; oc' \} \ \{A' : \text{Structure}S \; oc' \ (\text{fst} \ A) \; \Sigma') \ \{\varphi'\} \ bi \to
\]

Either (branchto A' \ \varphi' \ bi \in \text{bs})

Either ((fst A, A') \vdash_o \varphi')

(\Sigma (\lambda \oc'' \to \Sigma (\lambda (A'' : \text{Structure}S \; oc'' \ (\text{fst} A) \; \Sigma')) \to
\]

\[
\Sigma (\lambda \ bj \to \Sigma (\lambda (\text{pfbr2} : \text{branchto} A'' \ \varphi' \ bj \in \text{bs}) \to
\]

\[
(\text{gi} : (\text{fst} A, A') \vdash_s \varphi') \to
\]

isRed (fst A, A') \ \varphi' \ \text{gi} \ \text{fix} \to

\[
\text{gameEquiv} \ (\text{fst} A, A') \ (\text{fst} A, A'') \ \varphi' \ \text{gi} \ (x \ bj \text{pfbr2} \ (\text{lemma1} \_ \text{fix}))))))))
\]

\[
\times
\]

\[
(\text{isE} \ \varphi \to \forall \{ \Sigma' \; oc' \} \ \{A' : \text{Structure}S \; oc' \ (\text{fst} A) \; \Sigma') \ \{\varphi'\} \ bi \to
\]

Either (branchto A' \ \varphi' \ bi \in \text{bs})

Either ((fst A, A') \vdash_o \varphi' \ \text{false})
isRed takes a structure, $A$, a formula $\phi$, a game tree $A \vDash s \varphi$, and a bag $X$. In the case that the game tree is a leaf, which means it has no outward branches, isRed is true vacuously and so the definition for that case is $\text{Unit}$, or true. In the case that the game tree $A \vDash s \varphi$ is a node, we have a double product type; there are three requirements appended together. The first one (the code before the first $\times$) requires that for all $\text{StructureS}$’s $A'$ and $\text{Formulas} \varphi'$ such that there is a branch $bj \in bs$ that connects $(A, \phi) \rightarrow ((\text{fst } A, A'), \phi')$, there is a reduced game on $((\text{fst } A, A'), \phi')$, i.e. $\text{isRed} (\text{fst } A , A') \varphi' (x \ bj \ \text{prfbr})$ fix holds. In case it is confusing that $(x \ bj \ \text{prfbr})$ is miraculously a game tree $(\text{fst } A , A') \vDash s \varphi'$. recall from the definition of $\vDash s$ that the second argument can be thought of as a function that takes a branch and a proof that that branch is in the list of branches corresponding to some game tree, and returns a game tree on the structure that the branch points to. In the code here, we are calling that second argument $x$, and we see that $bj$ is a branch, and $\text{prfbr}$ is a proof that $bj$ is in $bs$, i.e. the list of branches of $A \vDash s \varphi$. So, $(x \ bj \ \text{prfbr})$ is a game tree $(\text{fst } A , A') \vDash s \varphi'$.

Now, for the next two parts of the product type, the first corresponds to the case where the formula is universal, the other to the case where the formula is existential (the atomic cases will only happen with a leaf.) Let’s tackle the first of these two, which corresponds to the formula being universal. The first thing required is a proof that the formula is universal, isU $\varphi$. Then we need a proof
that for all StructureS’s A’ and formulas φ’ such that there is a branch bi between nodes (A, φ) and ((fst A , A’), φ’), either

- there is a proof that branch is a part of the list of branches of A ⊩ s φ or
- Either ((fst A , A’) ⊩ o φ’) i.e. the verifier has a winning strategy on the subgame labeled by that structure, formula pair; or
- there is another StructureS, A'', such that
  - there is a branch between nodes (A, φ) and ((fst A, A’), φ’);
  - there is a subgame (fst A, A’’) ⊩ s φ’ that is reduced, i.e. isRed (fst A , A’’) φ’ gi fi (where gi = (fst A, A’’) ⊩ s φ’ and fix = the bag X), and importantly, that there is a proof that this subgame is equivalent to (fst A , A’’) ⊩ s φ’, i.e. gameEquiv (fst A , A’) (fst A , A’’)

φ’ gi (x bj pfbr2) (lemma1 fix).

See that this corresponds exactly to the definition that we outlined for reduced games above: in the case of universal formulas, all subgames are true or undecided, and we only keep track of (keep in our special list of branches bs) undecided subgames that are not equivalent to one another.

The third part of this product type is very similar. The only difference is as follows. Instead of insisting that for all subgames, either the verifier has a winning strategy on the subgame, there is an isomorphic subgame to it in the list of subgames kept alive, or the subgame is in that list itself; we ask that for all subgames, either the falsifier has a winning strategy (Either ((fst A , A’) ⊩ o φ’ false)) or one of the other options. Again, we see this matches the mathematical definition for reduced games nicely.

And finally, we come to provesR, a combiner that defines a reduced game.

provesR : ∀ {Σ oc} (A : Structure oc Σ) (φ : Formula Σ) (X : fixed1 (fst A)) → Type
provesR A φ X = Σ (λ (game : A ⊩ s φ) → isRed A φ game X)
provesR takes a structure A, a formula φ, a game tree A ⊩ s φ, and a bag X, and gives a proof that there is a game tree A ⊩ s φ such that isRed A φ (A ⊩ s) φ holds.
CHAPTER 6

Outline of Algorithm and Work in Progress

1. Proof of Algorithm

While we will prove our own version of the overall proof of the algorithm with our code, we outline the proof suggested by Kneis et al. here[19]. We have extracted this proof from the more complex algorithm Kneis et al. give for a version of Courcelle’s Theorem that allows for linear optimization questions in MSO[19]. However, in an effort not to reproduce too much of their paper, we simply rely upon several lemmas from the paper by Kneis et al. We reproduce the statement of the lemmas below, but for the proof, please refer to the Kneis et al. paper. Additionally, note that we refer to two algorithms not mentioned yet, combine and combineForget. These are algorithms laid out in the Kneis et al. paper which we will not reproduce, but please think of them as similar to the agda functions combineIntro, combineJoin, and combineForget which will be described below, after the proof.

We show that given a $\Sigma$-structure $A$ that fully interprets $\Sigma$, a $\Sigma$-formula $\phi \in MSO(\Sigma)$, and a tree decomposition $T$ on $A$ and $\phi$, we can return a game that determines whether or not $A$ satisfies $\phi$. The following are the lemmas from the Kneis et al. paper that we use[19]:

**Lemma 11** Let $A_1$ and $A_2$ be compatible $\Sigma$-structures, $\phi$ in MSO($\Sigma$) and let $X_1 \subset A_1$ and $X_2 \subset A_2$ with $A_1 \cap A_2 = X_1 \cap X_2$.

Then, $\text{combine} (\text{reduce}(A_1,X,\phi),\text{reduce}(A_r,X,\phi)) \cong \text{reduce}(A_1 \cup A_2,X,\phi)$. 

75
Lemma 12 Let $A$ be a $\Sigma$-structure, $X \subset A$, and $x \in X$. Further, let $\text{reduce}(A, X, \phi) \not\in \bot, \top$. Then $\text{reduce}(A, X, \phi) \equiv \text{combineForget}(G, x)$.

Lemma 13 Let $A$ be a $\Sigma$-structure that fully interprets $\Sigma$, $X \subset A$, and $\phi \in MSO(\Sigma)$. Then the result of $MC(A, \phi)$ returns that $A \models \phi$ if and only if $\text{convert}(\text{reduce}(A, X, \phi))$ returns that $A \models \phi$. Here think of convert as the function $\text{provesRtoClosed}$, which will be explained later.

First, we prove that given a $\Sigma$-structure $A$, a $\Sigma$-formula $\phi \in MSO(\Sigma)$, and a tree decomposition $T$ on $A$ and $\phi$, we can create a reduced game over the entire structure by putting together games over the structure restricted to bags of the tree decomposition $T$. In other words, we show how the only games we ever compute are games restricted to the bags $X_i$ of the tree decomposition.

Suppose node $i$ is a leaf. By definition of a leaf of a tree decomposition, this means that $A_i = \emptyset$ and $X_i = \emptyset$. Here, we simply return $\text{reduce}(A[\emptyset], \emptyset, \phi)$ since there are no lower nodes to combine with. So, the desired result is achieved.

Suppose node $i$ is a join node, structure $A[A_l \cup A_r]$ and formula $\phi$. Then by definition of a join node, there are two tree decompositions $T_l$ and $T_r$, corresponding to the left and right children $i$. $T_l$ is over $A[A_l]$, $\phi$, where $A_l = \bigcup_{i \text{ at or below } l} X_i$, and $T_r$ is over $A[A_r]$, $\phi$, where $A_r = \bigcup_{i \text{ at or below } r} X_i$. We want to show that $\text{combine}(\text{reduce}(A_l, X, \phi), \text{reduce}(A_r, X, \phi)) = \text{reduce}(A_l \cup A_r, X, \phi)$. Recall Lemma 11 defined above; we wish to apply this lemma to prove our proposition.

Let $X_1 = X_l = X$, $X_2 = X_r = X$, $A_1 = A[A_l]$ and $A_2 = A[A_r]$. The requirements of Lemma 11 are that $A[A_l]$ and $A[A_r]$ are compatible, $X \subset A_l$, $X \subset A_r$, and that $X \cap X = A_l \cap A_r$.

First we show that $A[A_l]$ and $A[A_r]$ are compatible. Recall that two $\Sigma$-structures $A_1$ and $A_2$ are compatible if:

- For all constants $c_i$ such that $c_i^{A_1} \neq \text{nil}$ and $c_i^{A_2} \neq \text{nil}$, then $c_i^{A_1} = c_i^{A_2}$
The identity function is an isomorphism between \( A_1[A_1 \cap A_2] \) and \( A_2[A_1 \cap A_2] \).

By definition of a restriction of a structure, \( c_i^{A[A_r]} \) must equal \( c_i^{A[A_l]} \) if \( c_i \in \text{interpreted}(A[A_r]) \cup \text{interpreted}(A[A_l]) \). And again, by definition of a restriction of a structure, \( A[A_l][A_l \cap A_r] = A[A_r][A_l \cap A_r] \), so since \( A[A_l][A_l \cap A_r] = A[A_r][A_l \cap A_r] \), the identity function is indeed an isomorphism between the two structures, since the structures are the same. So, we have that the two structures are compatible.

Now we show that the rest of the conditions of Lemma 11 hold, namely that \( X \subset A_l \), \( X \subset A_r \), and that \( X \cap X = A_l \cap A_r \). Recall the definition of \( A_i \) for a node \( i : A_i = \bigcup_{j \text{ at or below } i} X_j \) for all bags \( X_j \) for nodes \( j \) equal to or below \( i \). So, by definition of \( A_l \) and \( A_r \), \( X \subset A_l \) and \( X \subset A_r \) since \( X \) is the bag at nodes \( l \) and \( r \). We see that \( X \cap X = A_l \cap A_r = X \). Firstly note that \( X \subset X = A_l \cap A_r \) necessarily because the bag of a join node is defined to be \( X = X_r = X_l \), and therefore \( X \subset A_r \). To see the other containment, recall that by condition three of a tree decomposition, all bags that contain the same element must be connected. If there exists a \( y \in A_l \cap A_r \), \( y \not\in X \), then this condition would not hold. If \( y \in A_l \) and \( y \in A_r \), there must exist some node \( l' \) at or below \( l \) and a node \( r' \) at or below \( r \) such that \( y \in X_{l'} \) and \( y \in X_{r'} \). Then by condition 3 of tree decompositions, there must be a path between these two nodes such that \( y \) is contained in every bag of every node on that path. However, the only connection between \( l' \) and \( r' \) is through the node that connects \( r \) and \( l \), which has bag \( X \), so \( y \) must be in \( X \). Therefore, \( A_l \cap A_r = X \) as desired. With these qualifications in place, we can apply Lemma 11 and get that \( \text{combine}(\text{reduce}(A_l, X, \phi), \text{reduce}(A_r, X, \phi)) = \text{reduce}(A_l \cup A_r, X, \phi) \).

Suppose \( i \) is an introduce node, over structure \( A[A_j \cup x] \), (where \( j \) is the node directly below \( i \) and \( x \not\in A_j \)) and formula \( \phi \). Then by definition of an introduce
CHAPTER 6. ALGORITHM AND PROGRESS

1. PROOF OF ALGORITHM

node, we have a tree decomposition $T_j$ at node $j$ where the tree decomposition is over structure $A[A_j]$ and $\phi$. We want to show that

$$combine(reduce(A[A_j], X_j, \phi), reduce(A[X_j \cup x], (X_j \cup x), \phi))$$

$$= reduce(A_j \cup x, X_j \cup x, \phi)$$

We can use Lemma 11 again: we let $X_1 = X_j$, $X_2 = X_j \cup x$, $A_1 = A[A_j]$ and $A_2 = A[X_j \cup x]$. Again we require that $A[A_j]$ and $A[X_j \cup x]$ are compatible, $X_j \subset A_j$, $X_j \cup x \subset X_j \cup x$, and that $X_j \cap (X_j \cup x) = A_j \cap (X_j \cup x)$. The proof of compatibility for the two structures is exactly the same as the proof above, substituting the relevant subsets. For the rest of the requirements, note that again $X_j \in A_j$ by definition of $A_j$. $X_j \cup x \subset X_j \cup x$ since they are the same set. $X_j \cap (X_j \cup x) = X_j$ since by assumption $x \notin X_j$, and $A_j \cap (X_j \cup x) = X_j$ as well since by definition $X_j \subset A_j$, but $x \notin A_j$. Therefore $X_j \cap (X_j \cup x) = X_j = A_j \cap (X_j \cup x)$ as required, so by Lemma 11 we have the desired result that

$$combine(reduce(A[A_j], X_j, \phi), reduce(A[X_j \cup x], (X_j \cup x), \phi)) =$$

$$reduce(A_j \cup x, X_j \cup x, \phi).$$

Suppose $i$ is a forget node. Then by definition of a forget node, we have a tree decomposition $T_j$ at node $j$ directly below $i$ such that $A_j = A_i$ but $X_j = X_i \cup x$ for some $x \notin X_i$. $T_j$ is over structure $A[A_j]$ and $\phi$. We want to show that $combineForget(reduce(A[A_i], (X_i \cup x), \phi), x) = reduce(A[A_i], X_i, \phi)$. Then recall Lemma 12, which we will use for this proof. Let $A = A[A_i]$, $X = X_i \cup x$, and $x = x$. The lemma requires that $X_i \cup x \subset A[A_i]$, which is true by construction, and that $x \in X_i \cup x$, which is also true by construction. So, by Lemma 12, we have the desired result that $combineForget(reduce(A[A_i], (X_i \cup x), \phi), x) = reduce(A[A_i], X_i, \phi)$. 78
Now, given that we have a proof that our method of building a reduced game from smaller pieces returns a game that is isomorphic to a reduced game made over the entire structure, we need to show that that reduced game gives the same final result as a brute-force model checking game, which as mentioned previously, is proven to return a winning strategy for the verifier if and only if the input MSO sentence holds on the given structure. But, note that this result is already given to us in Lemma 13: since by the precondition to our overall algorithm, the input structure \( A \) fully interprets \( \Sigma \), and by construction, \( X \subset A \), all the conditions for Lemma 13 are satisfied. So the result of \( MC(A, \phi) \) returns that \( A \models \phi \) if and only if \( convert(reduce(A, X, \phi)) \) returns that \( A \not\models \phi \). Therefore, given a \( \Sigma \)-structure \( A \) that fully interprets \( \Sigma \), a \( \Sigma \)-formula \( \phi \in MSO(\Sigma) \), and a tree decomposition \( T \) on \( A \) and \( \phi \), we can return a game that determines whether or not \( A \) satisfies \( \phi \).

This proof is fine and dandy, but it does not mention the time bound of the algorithm, which is arguably the entire reason why Courcelle's Theorem is interesting. Unfortunately, proving the time bound is one of the parts of the Courcelle's Theorem that is left for the future of this project. However, we present an intuitive idea of the proof. Note that the brute force \( MC \) game algorithm takes as many steps as there are quantifiers or conjunctions in the MSO formula, and that it takes exponential time in the size of the input structure because you have to try every element in the input structure at every quantifier. However, note that the only place we ever apply the brute force \( MC \) game on the structure is on games where the structure is restricted to the bag of a given node. But these bags, by definition of a tree decomposition, have size less than or equal to \( t \) where \( t \) is the treewidth of the input graph \( G \). So, each time we run the brute force algorithm, even though the algorithm is exponential in the size of the input structure, it ends up being constant time because the time it takes is not related
to the size of the overall input graph, but only the treewidth. The algorithms combine and forget are linear because they only ever work over reduced games, and the of the size of a reduced game is small enough to keep these calculations under FPT linear time as explained in the Kneis et al. paper[19].

2. Agda Code and Proof

Keeping the above proof of the algorithm in mind, we now go into the Agda code that was promised. The overarching proof is described by these three functions: \texttt{algorithm}, \texttt{provesRtoClosed}, and \texttt{openToClosed}. Picking up from towards the end of the mathematical proof, the idea here is that \texttt{algorithm} provably takes: a subset \(B\), a bag \(X\), a \(\Sigma\)-structure \(A\), a \(\Sigma\)-formula \(\phi\), a proof that \(B\) is decidable, proof that \(B \subset A\), and finally tree decomposition \(TD\) on \(A\), \(X\), and \(\phi\), and returns either that

- the verifier has a winning strategy on \(EMC(A, X, and \phi)\),
- that the falsifier has a winning strategy on \(EMC(A, X, and \phi)\),
- or a reduced undecided game \(reduced(A[B], X, \phi)\).

The idea of what the proof of this function will look like is that, like the proof above, it will case on the tree decomposition. It will use the functions \texttt{combineJoin} and \texttt{combineIntro} to take care of combining the recursive calls where the mathematical proof used Lemma 11, and it will use \texttt{combineForget} where the mathematical proof used Lemma 12. If \texttt{algorithm} returns an undecided reduced game, that game will be sent to \texttt{provesRtoClosed}, which is our interpretation of Lemma 13 from the proof above, and if \texttt{algorithm} returns a decided \(EMC\) game (i.e. an open truth), that result will be sent to \texttt{openToClosed} which is our interpretation of another lemma from the paper.
We see quite clearly that \texttt{openToClosed} takes a structure and a formula, a $\models_o$, i.e. an \textit{EMC} game on that pair, and then sends it to a $\models_c$, i.e. an \textit{MC} game, on that pair. \texttt{provesRtoClosed}, similarly, takes reduced games to \textit{MC} games, as lemma 13 does. The general idea of both of these proofs is essentially that if your structure is closed, you can simply take out the branches where constants were interpreted as \textit{nil} to keep options open for future expansions, (since all possible options are already expressed), and see if the closed \textit{MC} game left behind has a winning strategy for the verifier. But even more simply, if a game is still undecided even when the entire underlying set is available to the game, then the game is true if it is a universal game and false if it is an existential game. This is because the \textit{nil} is kept as a sort of open space for more possible interpretations of constants, if more of the underlying set is to be merged in. If a universal game is undecided, it must be because the verifier has a winning strategy on all branches except those with \textit{nil}, because if the falsifier had a winning strategy on even one subgame, the game would have been terminated. However, if the \textit{nil} doesn’t really exist since all possible interpretations have been explored, this must mean that the formula does work for every possible interpretation and the universal formula is true on the structure. A dual argument works for existential formulas being false.

\texttt{algorithm} : $\forall \{\Sigma\} (B : \text{Subset}) (X : \text{fixed1} B) (A : \text{Structure Closed} \Sigma) (\varphi : \text{Formula} \Sigma)$
\hfill $(\text{decB} : \text{DecidableSub} B) (\text{BinA} : \text{Sub} B (\text{fst} A))$
\hfill $\rightarrow (\text{TD} : \text{TreeDecomp} \{\Sigma = \Sigma\} \{A\} (\text{fst} X) B)$
\hfill $\rightarrow \text{Either} (\text{Either} (A \models_o \varphi) (A \models_o \varphi \text{ false})) (\text{provesR (restriction A B decB BinA) } \varphi X)$

\{-EMC $\Rightarrow$ MC i.e. Lemma 11 -\}

\texttt{openToClosed} : $\forall \{\Sigma\} \rightarrow (A : \text{Structure Closed} \Sigma) (\varphi : \text{Formula} \Sigma)$
\hfill $\rightarrow (\text{otruth} : A \models_o \varphi) \rightarrow A \models_c \varphi$
Now that we have a grasp on the outer layer of the algorithm, we take a look at one layer down: what will go inside the cases of algorithm, which cases on the constructors of the given tree decomposition. First, we need to trudge through some background code that is used throughout these lemmas. Recall that fixed and fixed1 are the types for bags in a tree decomposition, re-defined below. fixed is just a proof that for given subsets $A_1$ and $A_2$, there exists a subset $X$ such that $X$ is in each of $A_1$ and $A_2$ and $X$ is decidable. Similarly, fixed1 says that given a subset $A_1$, there exists a subset $X$ such that $X \subseteq A_1$ and $X$ is decidable. fixed2fixed1 and fixed2fixed2 show that fixed $A_1 A_2$ implies fixed1 $A_1$ and fixed1 $A_2$. fixed2union asserts that if you have fixed $A_1 A_2$, then you must have fixed1 (union $A_1 A_2$). Then, fixed1Sub asserts that if $X$ is a possible bag of a set $A_1$ and $A_1 \subseteq A_2$ then $X$ could be a bag of $A_2$ as well. decSingleton states that a singleton set is always decidable. IndividSinSubset is essentially a dance around Agda’s strict type system. It says that if an element $x$ is in a given subset $A$, then the singleton set $\{x\}$ is a subset of $A$. These lemmas are unimportant as they are just small, obvious proofs to aid the larger ones; they are outlined so as not to confuse the reader but we do not include their code.

As mentioned earlier, we translate Lemma 11 into Agda with two functions, one to be used in the case that we are combining two subgames from the join node of a tree decomposition called combineJoin, and one to be used in the case that we are combining two games in an introduce node of a tree decomposition, called combinIntro. The types of these functions are below. See that combineJoin
takes a structure $A$, two decidable subsets $B_1$ and $B_2$ (where $A_1$ in the definition of the lemma is supposed to be represented by $A[B_1]$ and similar for $B_2$), a bag $X$ (fixed $B_1 B_2$), and proofs that both $B_1 \subset A$ and $B_2 \subset A$, and two undecided reduced games, $\text{reduced}(A[B_i] \phi X)$ for $i \in 1, 2$. \texttt{combineJoin} returns either a proof that the verifier has a winning strategy for $EMC(A[B_1], X, \phi)$, a proof that the falsifier has a winning strategy for that game, or that the game is reduced and undecided. Note that the result type is the same as that of the overarching \texttt{algorithm}, as required because \texttt{algorithm} returns the result of \texttt{combineJoin}. The proof of this function is in process, as with the rest outlined in this chapter.

\begin{align*}
\texttt{combineJoin} &: \forall \{\Sigma\} \{\text{oc}\} (B_1 B_2 : \text{Subset}) (A : \text{Structure oc } \Sigma) \\
&\quad (\phi : \text{Formula } \Sigma) \\
&\quad \rightarrow (\text{decb1} : \text{DecidableSub } B_1) (\text{decb2} : \text{DecidableSub } B_2) (X : \text{fixed } B_1 B_2) \\
&\quad (b1subAset : \text{Sub } B_1 (\text{fst } A)) (b2subAset : \text{Sub } B_2 (\text{fst } A)) \rightarrow \\
&\quad \quad (\text{recurcall1} : (\text{provesR } (\text{restriction } A B_1 \text{ decb1 } b1subAset) \phi (\text{fixed}2\text{fixed1 } B_1 B_2 X))) \\
&\quad \quad (\text{recurcall2} : (\text{provesR } (\text{restriction } A B_2 \text{ decb2 } b2subAset) \phi (\text{fixed}2\text{fixed2 } B_1 B_2 X))) \\
&\quad \rightarrow \textbf{let } B_1 \cup B_2 = (\text{restriction } A (\text{union } B_1 B_2) (\text{unionDec } \{S_1 = B_1\} \{B_2\} \text{ decb1 decb2}) \\
&\quad \quad (\text{subLUB b1subAset b2subAset})) \\
&\quad \quad \textbf{in} \\
&\quad \quad \text{Either} (\text{Either } (B_1 \cup B_2 \vdash-o \phi) \\
&\quad \quad \quad (B_1 \cup B_2 \vdash-o \phi \texttt{ false})) \\
&\quad \quad \quad (\text{provesR } B_1 \cup B_2 \phi (\text{fixed}2\text{union } B_1 B_2 X))
\end{align*}

\texttt{combineIntro} is quite similar to \texttt{combineJoin}. See that \texttt{combineIntro} takes a decidable subset $B$; a structure $A$, a set element $x$ of $A$, (where $A_1$ in the definition of the lemma is supposed to be represented by $A[B]$ and similar for $A_2$ and $[X \cup x]$), a proof that $x \notin B$, a bag $X$ (fixed $B$), a proof that $B \subset A$, and an
undecided reduced game, \(\text{reduced}(A[B], X, \phi)\), and a naive (i.e. brute force) algorithm of whether the structure restricted to the bag \(X \cup x\) satisfies the formula. \(\text{combineIntro}\) returns either a proof that the verifier has a winning strategy for \(EMC(A[B \cup X], X \cup x, \phi)\), a proof that the falsifier has a winning strategy for that game, or that the game is reduced and undecided. Note that the result type is again the same as that of the overarching \(\text{algorithm}\), as required because \(\text{algorithm}\) returns the result of \(\text{combineIntro}\).

\[
\text{combineIntro} : \forall \{\Sigma\} \{\tau\} (B : \text{Subset}) (A : \text{Structure Closed} \Sigma) (\varphi : \text{Formula} \Sigma)
(x : \text{IndividS} (fst A) \tau)
(xnew : (Sub (singleton \{\tau = \tau\} (fst x)) (complement B)))
(decb : \text{DecidableSub} B) (bsubAset : Sub B (fst A)) (X : \text{fixed1} B) \rightarrow
(recurcall : provesR (restriction A B decb bsubAset) \varphi X) \rightarrow
(nai : provesR (restriction A (union (fst X) (singleton \{\tau = \tau\} (fst x))))
(unionDec \{S1 = (fst X)\} \{singleton (fst x)\} (snd (snd X)) (decSingleton (fst x)))
(subLUB (subtrans (fst (snd X)) bsubAset) (individSinSubset (fst A) x)))
\varphi ((fst X), (subINL, (snd (snd X)))))
\rightarrow \text{let} B \cup x = (\text{restriction} A (\text{union} B (\text{singleton} \{\tau = \tau\} (fst x))))
(unionDec \{S1 = B\} \{singleton (fst x)\} decb (decSingleton (fst x)))
(subLUB bsubAset (individSinSubset (fst A) x)))
in
Either (Either (B \cup x \vdash o \varphi)
(B \cup x \vdash o \varphi \text{false}))
(provesR B \cup x \varphi ((fst X), ((\text{subtrans} (fst (snd X)) \text{subINL}), (snd (snd X)))))

Now we have \(\text{combineForget}\), which corresponds to Lemma 12 in the paper and is used in the case where the tree decomposition in the function \(\text{algorithm}\) is a forget node. \(\text{combineForget}\) takes a decidable subset \(B\); a structure \(A\), a
Σ-formula \( \phi \), a bag \( X \) (fixed \( B \)), a set element \( x \) of \( A \), a proof that \( x \notin X \), and a reduced undecided game \( \text{reduced}(A[B], X \cup x, \phi) \), (which is supposed to represent the result of a recursive call to \text{algorithm} for a node below the current node of the tree decomposition). \text{combineForget} only returns a reduced undecided game \( \text{reduced}(A[B], X, \phi) \), because Lemma 12 states that the game needs to be undecided for the lemma to hold.

\[
\text{combineForget} : \forall \{ \Sigma \} \{ \tau \} (B : \text{Subset}) (A : \text{Structure Closed } \Sigma) (\phi : \text{Formula } \Sigma) \\
(X : \text{fixed1 } B) (x : \text{IndividS } (\text{fst } X) \tau) \\
(\text{decB} : \text{DecidableSub } B) (\text{bsubAset} : \text{Sub } B (\text{fst } A)) \\
(\text{xgone} : (\text{Sub } (\text{singleton } \{ \tau = \tau \} (\text{fst } x))) (\text{complement } (\text{fst } X))) \rightarrow \\
(\text{recurcall} : (\text{provesR } (\text{restriction } A B \text{ decB bsubAset} \phi (X)))) \rightarrow \\
(\text{provesR } (\text{restriction } A B \text{ decB bsubAset} \phi X))
\]

As noted in the mathematical proof, there is no need for a combine algorithm for the base case, because we simply call \text{naive}, the brute force algorithm of determining if a formula holds on a structure. (For those who are familiar with the Kneis et al. paper, this is our eval.) The type of \text{naive} is included below: see that takes a formula \( \phi \), a structure \( A \), a bag \( X \), and returns whether the falsifier or the verifier has a winning strategy on \( EMC(A, X, \phi) \), or if the game is undecided. Unlike the combine algorithms, there is no fancy footing here: this algorithm is exponential time, but as outlined above, it only is used on sets of constant size.

\[
\text{naive} : \forall \{ \Sigma' \} \{ \text{oc} \} \rightarrow (\phi : \text{Formula } \Sigma') (A : \text{Structure } \text{oc } \Sigma') \rightarrow (X : \text{fixed1 } (\text{fst } A)) \\
\rightarrow \text{Either } (\text{Either } (A \models o \phi) (A \models o \phi \text{ false})) (\text{provesR } A \phi X)
\]

After the proof of the overall algorithm is finished, we plan to formalize in Agda how the algorithm outlined meets the FPT criteria for Courcelle’s Theorem. After we verify the time bound, then we will move on to verifying some
applications of this theorem—some promising directions seem to be in phyloge
netics and other graph algorithms, especially those having to do with graph minor
theory as explained in the related work section.
CHAPTER 7

Conclusion

We are well en-route to completing a verification in Agda of Courcelle’s Theorem, a general graph algorithm stating that questions definable in Monadic Second Order Logic can be decided in fixed-parameter tractable time on graphs of bounded treewidth. We have encoded all of the necessary definitions for this project, namely symbols (SigThing), signatures (Signature), structures (Structure), restrictions of structures (restriction), isomorphisms between structures (preserves, iso) set elements (Individ and IndividS), sets (Subset), tree decompositions (TreeDecomp), model checking games (|=c), extended model checking games (|=o), position equivalence and game equivalence (positionEquiv and gameEquiv), and finally, educed games (isRed, |=s, provesR). We are in the process of constructing the naive, or brute force algorithm (naive), which will run only on leaves of the tree decomposition and restrictions of the input structure of bags of the tree decomposition. We have defined the types for all of the lemmas required to prove the algorithm overall (algorithm, provesRtoClosed, openToClosed, combineIntro, combineJoin, combineForget), and the proofs of these types is in progress as well. Once this is completed, we hope to prove the time bound of the algorithm, and then verify its use in the graph minor decomposition paper by Reed et al.

This thesis could become useful if Courcelle’s Theorem becomes practicable in the field of relational databases, as the fact that it is verified could provide a sense of security to manipulating sensitive data more quickly. It could also allow
Bruno Courcelle to take a quick sigh of relief, since Agda agrees that his proof is correct.
Bibliography


