Implementing the completeness proof for the modal μ-calculus

Ben Sheffield

MInf Project (Part 1) Report
Master of Informatics
School of Informatics
University of Edinburgh

2020

Abstract

When analysing a logic, one property of interest is its' completeness. A logic is complete iff every valid formula is provable, and by proof we mean every formula is derivable from a deductive system which comprises axioms and rules of inference.

In 1983 Kozen presented μ-calculus in the form we know it today and gave a complete deductive system for a restricted version. It wasn't until 1995 when Walukiewicz proved completeness for the full language with a stronger deductive system[18] then with Kozen's original axiomatization[19] in 2000. In this report, we present an implementation of Walukiewicz original proof procedure.

Acknowledgements

Thanks to my supervisor, Julian Bradfield, for his many helpful comments throughout the project.

Contents

Chapter 1

Introduction

1.1 Motivation

Formal verification is a technique used to prove the correctness of hardware, algorithms and programs. It's used in industry for a range of critical applications, from cryptography to software that controls rockets, where problems with a program can have fatal consequences. There are two main approaches to this task, model checking and deductive verification.

The goal of model checking is to construct a mathematical model of the program, such that properties about the program can be expressed in terms of some kind of logic on the model. The advantage of this approach is that model construction and verification is often fully automatic, however, even moderately simple programs can have large models which are intractable.

The other approach is by deductive verification whereby the semantics of the programming language and the correctness properties to be proven are defined in some theorem prover such as Isabelle[7], Coq[4], Agda[2], etc. A proof of those properties is derived by manually applying rules. This requires a deep understanding of the problem and a great deal of ingenuity. While it's possible to fully automate this with satisfiability modulo theories (SMT) solvers, they can solve only the most basic problems.

In this report we're interested in modal μ-calculus which is a logic used in model checking to express correctness properties on Kripke structures. μ-calculus is a particularly interesting logic to look at; many of the other popular temporal logics can be translated to it including PDL, LTL, CTL and CTL*. Despite the expressive power of μ-calculus, the satisfiability problem is still decidable (albeit EXPTIME-complete). These properties make it attractive as a metalanguage for correctness properties in a formal verification tool.

1.2 Previous Work

I'm not aware of any project implementing any completeness proof for μ-calculus. The only project remotely close I found on GitHub by user *cipid*[8], who appears to have been a student at Edinburgh University. Although I was never able to build the project, it appears to apply a set of tableau rules to test for validity. This is equivalent to testing for a refutation in the system described here (see 2.17). The complete system described here goes much further, producing a proof tree from a set of axioms.

1.3 Literature

As stated in the abstract, this project implements the completeness proof as described in the earlier Walukiewicz paper[18]. The original goal for this project was to implement that latter paper[19] with Kozen's axioms. A great deal of time was spent trying to realise this, but I encountered several issues:

- The proof makes use of infinite games, and infinite automaton of which I wasn't familiar.
- It doesn't describe the construction of the automaton.
- The structure of the paper made it difficult to follow.

After completing this project and the semester 2 courses Formal Verification[5] and Algorithmic Game Theory and its Applications[3] I'm confident I can overcome these obstacles in part 2 of this project. We expand on this in section 5.1.

Walukiewicz earlier paper is still complex, but had some notable advantages with regard to successfully completing the project before the deadline, namely

- No use of infinite games.
- Full details on the construction of the automaton (see 3.4).

Despite these advantages, both papers were focused on proving completeness rather than giving an algorithm to produce proofs which is what we're interested in here. While Walukiewicz proof is constructive, it falls short of emitting a reasonable algorithm. The main challenge for this project was interpreting the complex proof and transforming it into a program.

1.4 Summary of results

The results of this project are summarised as follows:

- Parser for μ-calculus formulas, so the end user can input the formula they want to prove satisfiable.
- Refutation search and rendering of the respective tableau.
- Automaton construction and rendering of the automaton graph.

- Unwinding of the automaton graph and rendering the resulting tree.
- Sequent assignment to the nodes of the unwinding and rendering the resulting tree.

1.5 Structure of the report

Chapter 2 gives a brief summary of the background of μ-calculus and the completeness proof.

Chapter 3 walks through the notable parts of the implementation of the proof procedure.

Chapter 4 shows each stage of the construction with a simple example input, more examples can be found on my website[17].

Chapter 5 concludes the report and talks about the work that will be done next year for part 2 of this project.

Chapter 2

Background

2.1 Logic

2.1.1 Completeness

There are two notions of completeness.

Definition 2.1 (Strong Completeness)**.** For every set of premises Γ, any formula α that semantically follows from Γ is provable from Γ, that is:

$$\Gamma \vDash \alpha \implies \Gamma \vdash \alpha$$

Definition 2.2 (Weak Completeness)**.** Every valid formula α is provable, that is:

$$\vDash \alpha \implies \vdash \alpha$$

μ-calculus is not strongly complete since it fails the compactness theorem, thus when referring to completeness of μ-calculus we mean weak completeness.

2.1.2 Proof System

Definition 2.3 (Rule of Inference)**.** A function that maps a set of premises to a conclusion, for example modus ponens takes a premise of the form A and $A \to B$ and returns the conclusion B. As a proof tree this is denoted by:

$$\frac{A \to B \qquad A}{B}$$

A proof system comprises axioms and rules of inference. A tree constructed from a proof system forms a valid proof of a formula α if:

- The root is labelled α
- Every internal node is a conclusion of a rule
- Child nodes are labelled with the premises of the parent node
- Leaves are axioms

2.2 μ-calculus

2.2.1 Kripke Structures

Kripke structures are a way of modelling programs with states and transitions between states. Each state has a set of propositions which are true at that given time.

A Kripke structure given a set propositions $Prop$ and actions Act is a tuple $\langle S, R, \rho\rangle$ composed of:

- a finite set of states S
- a transition function $R : Act \to \mathcal{P}(S \times S)$ mapping an action to a binary relation.
- labelling function $\rho : Prop \to \mathcal{P}(P)$

Figure 2.1 is an example of a Kripke structure.

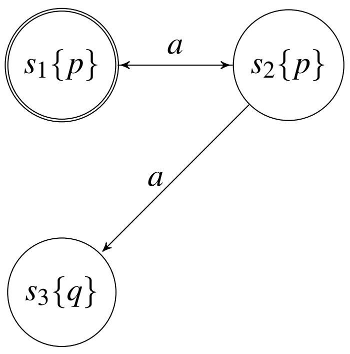

Figure 2.1: simple Kripke structure with labelled transitions

2.2.2 Modal Calculus

Hennessy–Milner logic (HML) extends propositional logic with modality operators. A formula in HML is defined by:

$$\Phi ::= \top \mid \bot \mid p \mid \neg\Phi \mid \Phi \wedge \Phi \mid \Phi \vee \Phi \mid \langle Act\rangle\Phi \mid [Act]\Phi$$

The expression $\langle a\rangle\alpha$ states that α must hold for some a step whereas the expression $[a]\alpha$ states that α must hold in all a steps.

Given a model $\mathcal{M} = \langle S, R, \rho\rangle$, a valuation function $V : Var \to \mathcal{P}(S)$ we define $||\alpha||_V^{\mathcal{M}}$ to be the set of states in which α is true. $\mathcal{M}$ is omitted when it causes no ambiguity.

$$\begin{aligned}
||\top||_V &= S \\
||\bot||_V &= \emptyset \\
||X||_V &= V(X) \\
||p||_V &= \rho(p) \\
||\neg\alpha||_V &= S - ||\alpha||_V \\
||\alpha \wedge \beta||_V &= ||\alpha||_V \cap ||\beta||_V \\
||\alpha \vee \beta||_V &= ||\alpha||_V \cup ||\beta||_V \\
||\langle a\rangle\alpha||_V &= \{s : \exists t.(s,t) \in R(a) \wedge t \in ||\alpha||_V\} \\
||[a]\alpha||_V &= \{s : \forall t.(s,t) \in R(a) \implies t \in ||\alpha||_V\}
\end{aligned}$$

We write $\mathcal{M}, s, V \vDash \alpha$ when $s \in ||\alpha||_V$. For example, consider the model $\mathcal{M}$ in figure 2.1 and an empty evaluation function V following:

- $\mathcal{M}, s_1, V \vDash p$ holds since p holds in s_1
- $\mathcal{M}, s_1, V \vDash \langle a\rangle p$ holds since taking a a step in state s_1 results in state s_2 where p holds.
- $\mathcal{M}, s_2, V \vDash [a](p \vee q)$ holds since taking a a step in state s_2 results in either state s_1 or s_3 and in both states $p \vee q$ hold.

HML is very limited in the range of properties it can express, the solution is to combine it with fixpoint operators.

2.2.3 Fixpoints

Fixpoints are the key idea that gives μ-calculus its expressive power, its also what makes μ-calculus expressions hard to understand.

Definition 2.4 (fixpoint)**.** For a function $f : D \to D$ where D has a partial order $\leq$, x is a fixpoint of f if $f(x) = x$

Theorem 2.5 (Knaster–Tarski)**.** *For a complete lattice $\langle L, \leq\rangle$ and a monotone function $f : L \to L$, the set of fixpoints of f form a complete lattice.*

Now, the set of all states S combined with set inclusion form a complete lattice. A formula $\alpha(X)$ with a free variable X can be viewed as a function $f : \mathcal{P}(S) \to \mathcal{P}(S)$, and if $f(T) \subseteq f(T')$ whenever $T \subseteq T' \subseteq S$ then f is monotonic. By theorem 2.5 the fixpoints of f form a complete lattice, thus f has a minimal and maximal fixpoint.

Definition 2.6 (Maximal fixpoint)**.** Union of post-fixpoints.

$$\bigcup\{T \subset S \mid T \subseteq f(T)\}$$

Definition 2.7 (Minimal fixpoint)**.** Intersection of pre-fixpoints

$$\bigcap\{T \subset S \mid f(T) \subseteq T\}$$

2.2.4 μ-calculus

The syntax of μ-calculus is specified by:

$$\Phi ::= \top \mid \bot \mid p \mid X \mid \neg\Phi \mid \Phi \wedge \Phi \mid \Phi \vee \Phi \mid [Act]\Phi \mid \langle Act \rangle \Phi \mid \mu X.\Phi \mid \nu X.\Phi$$

where for all $\mu X.\alpha$ all variables X in α occur positively. We will use $\sigma X.\alpha$ to mean either $\mu X.\alpha$ or $\nu X.\alpha$.

The semantics extends HML

$$\|\mu X.\alpha(X)\|_V = \bigcap\{T \subseteq S : \|\alpha\|_{V[T/X]} \subseteq T\}$$
$$\|\nu X.\alpha(X)\|_V = \bigcup\{T \subseteq S : T \subseteq \|\alpha\|_{V[T/X]}\}$$

Definition 2.8 (Positive)**.** A formula α is in positive normal form if $\neg$ appears before a variable or constant only.

Definition 2.9 (Guarded)**.** Every bound variable is within the scope of some modality operator, $\langle\rangle$ or $[]$

Theorem 2.10 (Kozen)**.** *Every formula is equivalent to a positive guarded formula.*

The proof of theorem 2.10 can be found in Walukiewicz paper[18]. We will now fix an unsatisfiable positive guarded formula φ_0.

2.3 Completeness proof

In this section we will walk through the procedure for producing a proof that φ_0 is unsatisfiable.

2.3.1 Definition List

A definition list $\mathcal{D}_{\varphi_0}$ assigns labels $U_0, U_1, ...$ to each of the fixpoints in φ_0. The definition list is formed by the contraction operation:

1. $\llparenthesis p \rrparenthesis = \llparenthesis \neg p \rrparenthesis = \llparenthesis X \rrparenthesis = \llparenthesis U \rrparenthesis = \emptyset$
2. $\llparenthesis \neg\alpha \rrparenthesis = \llparenthesis \langle a \rangle \alpha \rrparenthesis = \llparenthesis [a]\alpha \rrparenthesis = \llparenthesis \alpha \rrparenthesis$
3. $\llparenthesis \alpha \wedge \beta \rrparenthesis = \llparenthesis \alpha \vee \beta \rrparenthesis = \llparenthesis \alpha \rrparenthesis \circ \llparenthesis \beta \rrparenthesis$
4. $\llparenthesis \mu X.\alpha(X) \rrparenthesis = ((U = \mu X.\alpha(X)), \llparenthesis \alpha(U) \rrparenthesis)$
5. $\llparenthesis \nu X.\alpha(X) \rrparenthesis = ((U = \nu X.\alpha(X)), \llparenthesis \alpha(U) \rrparenthesis)$

The operation $\llparenthesis \alpha \rrparenthesis \circ \llparenthesis \beta \rrparenthesis$ is defined as:

1. Make the definition constants in $\llparenthesis \alpha \rrparenthesis$ distinct from $\llparenthesis \beta \rrparenthesis$
2. If $(U = \gamma) \in \llparenthesis \alpha \rrparenthesis$ and $(V = \gamma) \in \llparenthesis \beta \rrparenthesis$, delete V from $\llparenthesis \beta \rrparenthesis$ and replace V with U in $\llparenthesis \alpha \rrparenthesis$
3. Repeat until no fixpoint is doubly defined

U_i is older than U_j if $i < j$ with respect to $\mathcal{D}$.

2.3.2 Tableau

The first step is the construction of an infinite tableau as developed by Niwiński and Walukiewicz[18]. The construction of the tableau is a series of reductions given by a set of tableau rules

Definition 2.11 (Tableau Rules).

$$(\wedge_t) \quad \frac{\alpha, \beta, \Gamma \vdash_{\mathcal{D}}}{\alpha \wedge \beta, \Gamma \vdash_{\mathcal{D}}}$$

$$(\vee_t) \quad \frac{\alpha, \Gamma \vdash_{\mathcal{D}} \quad \alpha, \Gamma \vdash_{\mathcal{D}}}{\alpha \vee \beta, \Gamma \vdash_{\mathcal{D}}}$$

$$(const) \quad \frac{\alpha(U), \Gamma \vdash_{\mathcal{D}}}{U, \Gamma \vdash_{\mathcal{D}}} \quad (U = \sigma X.\alpha(X)) \in \mathcal{D}$$

$$(\sigma_t) \quad \frac{U, \Gamma \vdash_{\mathcal{D}}}{\sigma X.\alpha(X), \Gamma \vdash_{\mathcal{D}}} \quad (U = \sigma X.\alpha(X)) \in \mathcal{D}$$

$$(\langle\rangle_t) \quad \frac{\alpha, \{\beta : [a]\beta \in \Gamma\} \vdash_{\mathcal{D}}}{\langle a \rangle \alpha, \Gamma \vdash_{\mathcal{D}}}$$

Note 2.12. Applying the expansion operation to the left-hand side the set of tableau rules become a sound set of logical rules.

Definition 2.13 (Tableau Axiom). A tableau sequent $\Gamma \vdash_{\mathcal{D}}$ is a tableau axiom iff there is some variable or propositional letter p, such that $p, \neg p \in \Gamma$.

Definition 2.14 (Tableau). For a positive guarded formula γ and definition list $\mathcal{D} = \llbracket \gamma \rrbracket$, a tableau is a pair $\langle K, L \rangle$ where K is a tree and L is the labelling function in which:

1. The root is labelled with $\gamma \vdash_{\mathcal{D}}$
2. If $L(n)$ is a tableau axiom, then it's a leaf
3. If $L(n)$ is not an axiom, then the sons $m_1, .., m_i$ are labelled according to the rules of the tableau system such that $L(m_1), ..., L(m_i)$ are the assumptions and $L(n)$ is the conclusion.

For modal calculus we can construct a proof directly from its tableau. The difference for μ-calculus is that tableaux are infinite due to the $(const)$ rule, thus we will need to work harder to produce a proof.

Definition 2.15 (Trace). A sequence of formulas $\alpha_1, \alpha_2, ...$ from an infinite path $v_1, v_2, ...$ such that $\alpha_n \in L(v_n)$ and:

- $\alpha_{n+1} = \alpha_n$ if α_n is not reduced in $L(v_{n+1})$
- Otherwise, α_{n+1} is one of the resulting formula reduced in $L(v_{n+1})$

Definition 2.16 (μ-trace). U regenerates of a trace $\alpha_1, \alpha_2, ...$ if for some i, $\alpha_i = U$ and $\alpha_{i+1} = U$ where $(U = \sigma X.\alpha(X)) \in \mathcal{D}$. If the oldest constant that regenerates infinitely often is a μ constant then the trace is a μ-trace. Otherwise the trace is a ν-trace.

Definition 2.17 (Refutation)**.** A tableau for γ is a refutation for γ iff every leaf is labelled with a tableau axiom and on every infinite path there exists a μ-trace.

Theorem 2.18 (Characterisation)**.** *For every positive guarded formula γ, γ is not satisfiable iff there is a refutation of γ.*

With this additional restriction and theorem 2.18, we have something from which we will be able to construct a proof for every unsatisfiable formula.

2.3.3 Automaton

Next we construct a special Rabin automaton $\mathcal{A}$ on trees which recognises paths of a tableau with a μ-trace. Clearly, there is a non-deterministic Büchi automaton with this property. Simply pick a formula such that the sequence of chosen formulas forms a trace, if the trace is a μ-trace then the automaton accepts.

Using Safra's construction this automaton can be determinized. To avoid redundancies the paper describes a construction from scratch, we will leave the details of this to section 3.4.

States of the automaton will be labelled ordered trees $T = \langle N, r, p, \prec, nl, el \rangle$ where

- N is a set of vertices
- $r \in N$ is the root
- $p : (N \setminus \{r\}) \to N$ is a parenthood function
- $\prec$ is a sibling ordering
- $\mathrm{nl}(v)$ is the node label for any vertex $v \in N$ comprised of a non-empty subset of $\mathrm{FL}(\gamma)$
- $\mathrm{el}(v)$ is the edge label for any vertex $v \in N \setminus \{r\}$ comprised of a definition constant from $\mathcal{D}_\gamma$

Theorem 2.19. *The automaton accepts a path $\mathcal{P}$ of a tableau iff there exists an infinite μ-trace.*

The proof of 2.19 is similar to the proof of correctness of Safra's construction and can be found in Walukiewicz paper[18].

Theorem 2.20 (Emerson)**.** *Suppose $\mathcal{A}$ is a Rabin automaton over a single letter alphabet. If $\mathcal{A}$ accepts some tree then there is a graph $\mathcal{G}$ with states of $\mathcal{A}$ as nodes which unwinds to an accepting run of $\mathcal{A}$.*

We run $\mathcal{A}_{\varphi_0}$ on all paths of the tableau, because it's deterministic we will obtain a tree. By theorem 2.20 there is a graph $\mathcal{G}_{\varphi_0}$ with states of $\mathcal{A}_{\varphi_0}$ as nodes. This unwinds to an accepting run of $\mathcal{TA}_{\varphi_0}$, the expansion of $\mathcal{A}_{\varphi_0}$ to a non-deterministic tree automaton.

2.3.4 Unwinding

The unwinding of graph $\mathcal{G}_{\varphi_0}$ into a finite tree with back edges will give us the structure of the proof tree, to which we will assign sequents. First we must find the nodes in which the induction rule is applied, so-called loop nodes.

Definition 2.21 (Ord)**.** For any node s of $\mathcal{G}_{\varphi_0}$, let $\mathrm{Ord}(s)$ be the list $(v_1, ..., v_k)$ such that each $i..k$ is defined by the following process:

1. Delete all nodes where one of $v_1, ..., v_{i-1}$ lights green.
2. Let C_i be the non-trivial strongly connected component of $\mathcal{G}_{\varphi_0}$ containing s
3. v_i is the smallest vertex (using a fixed arbitrary ordering on all vertices) such that v_i does not disappear in any node of C_i and lights green in some node of C_i.

Definition 2.22 (Loop node, green vertex)**.** s is a loop node of $\mathcal{G}_{\varphi_0}$ iff $\mathrm{Ord}(s) = (v_1, ..., v_k)$ and v_i lights green in s. v_i is the green vertex of s.

Definition 2.23 (Type 1 final node)**.** There is a prefix $m(m\downarrow, 0)$ of n such that for the green vertex of v of $m\downarrow$ we have:

1. Definition constant $\mathrm{el}(v)$ appears in the node label of v in $n\downarrow$
2. v appears in $\mathrm{Ord}(m'\downarrow)$ for any prefix m' of n longer than m

Definition 2.24 (Type 2 final node)**.** There is a prefix $m(n\downarrow, 1)$ of n with the property that for any longer prefix m' of n, $\mathrm{Ord}(n\downarrow)$ is a prefix $\mathrm{Ord}(m'\downarrow)$. In this case we will say that there is a back edge from n to m.

And now we can define the tree with back edges $\mathcal{T}_{\varphi_0}$.

- The root of $\mathcal{T}_{\varphi_0}$ is the singleton sequence, the root of $\mathcal{G}_{\varphi_0}$.
- Node n is final if its a type 1 or a type 2 final node.
- If n is not final and $n\downarrow$ is not a loop node then for any son s in $\mathcal{G}_{\varphi_0}$ of the last state in n we add node ns.
- If n is not final and $n\downarrow$ is a loop node then we add two nodes $n(n\downarrow, 0)$ and $n(n\downarrow, 1)$.

2.3.5 Sequent Assignment

Let Γ_n denote the set of all formulas appearing in the last state of n. Consider the simple sequent assignment where each node of T_{φ_0} we assign the sequent $\langle\!\langle \Gamma \rangle\!\rangle_{\mathcal{D}_{\varphi_0}} \vdash$. This assignment will be locally sound (note 2.12), but in general there may be leaves that are not assigned axioms.

The solution is to apply the induction rule to all nodes where $m\downarrow$ is a loop node. An application of the (ind) rule remembers a node that ends in a loop node represented as an auxiliary Z variable, when we get to the leaf we can use this information to obtain an axiom.

There is an issue, if a green vertex disappears, then we lose an auxiliary variable which causes the sequent assigned to be unprovable. The solution is to remember only auxiliary variables related to active nodes (AN below) and forgetting all other Z variables using the (cut) rule.

We only give a brief overview on what the sequent assignment is doing. The proof of its' soundness is too long to repeat here, so the reader is referred to Walukiewicz paper[18] for a full explanation.

Definition 2.25 (Active nodes)**.** For node n of $\mathcal{T}_{\varphi_0}$, the set $AN(n)$ is the smallest set such that:

- $m \in AN(n)$ if there is a back edge from n or some descendant of n to some proper ancestors m of n
- $AN(m) \subseteq AN(n)$ if $m \in AN(n)$

Let $AV(n) = \{v : m \in AN(n), v \text{ is the green vertex of } m\downarrow\}$

Definition 2.26. Let $AN(n)$ be the set of nodes m such that $m(m\downarrow, 0)$ is a prefix of n and the green vertex of $m\downarrow$ appears in $\mathrm{Ord}(m'\downarrow)$ for any prefix longer than m.

Definition 2.27. For node n of T_{φ_0} and vertex v of $n\downarrow$ its signature, $||v||_n$ is a sequence of formulas $(\phi_1, ..., \phi_{d_\mu})$ where d_μ is the number of μ-constants in φ_0. The formula ϕ_i is defined as follows:

1. Find the lowest ancestor u of v with $\mathrm{el}(u) = U_i$
2. Let $N = \{o \in AN(n) : \text{green vertex } v \text{ of } o \text{ is an ancestor of u and } \mathrm{el}(v) = U_i\}$
3. Let $m \in N$ be the closest node in T_{φ_0} to n
4. ϕ_i is defined by cases:
    - if $u \in AV_0(n)$ then $\phi_i = ff$
    - if $u \in AV(n)$ then $\phi_i = \mathcal{V}' \wedge \alpha_i(\mu X.\mathcal{V} \wedge \alpha_i(X))$
    - otherwise, $\phi_i = \mu X.\mathcal{V} \wedge \alpha_i X$

   Where $\mathcal{V} = \bigwedge\{Z_o : o \in N\}$ and $\mathcal{V}' = \bigvee\{Z_o : o \in N, o \neq m\}$

If $n\downarrow$ is a loop node and v is a green vertex of $n\downarrow$, then $\mathrm{el}(v) = U_i$ for some μ-constant U_i and $||v||_n$ differs from $\widehat{||v||}_n$ or $\overline{||v||}_n$ at index i, we let $\hat{\phi}_i = \mu X.\mathcal{V} \wedge Z_n \wedge \alpha_i(X)$ and $\bar{\phi}_i = \mu X.\mathcal{V} \wedge \alpha_i(X)$.

Definition 2.28. Γ_n is the set of all formulas in the root of $n\downarrow$ for every node n in T_{φ_0}

Definition 2.29 (ϕ-vertex)**.** For every $\phi \in \Gamma_n$ the ϕ-vertex is the lowest vertex in $n\downarrow$ containing ϕ

Definition 2.30. For node n of T_{φ_0} and $\varphi \in \Gamma_n$, let v be the ϕ-vertex of $m\downarrow$. Let $(\phi_1, ..., \phi_{d_\mu}) = ||v||_n$. $\mathcal{D}(n,\varphi)$ is $\mathcal{D}_{\varphi_0}$ where, for every $i = 1, ..., d_\mu$, the definition of constant U_i is replaced with ϕ_i. $\hat{\mathcal{D}}(n,\varphi)$ and $\bar{\mathcal{D}}(n,\varphi)$ are the same, but use $\widehat{||v||}_n$ and $\overline{||v||}_n$ respectively.

Finally, we're ready to define the sequent assignment.

Definition 2.31 (Sequent assignment)**.** For node n of T_{φ_0} we define:

$$F(n) = \bigwedge\{\langle\!\langle \varphi \rangle\!\rangle_{\mathcal{D}(n,\varphi)} : \phi \in \Gamma_n\}$$
$$\bar{F}(n) = \bigwedge\{\langle\!\langle \varphi \rangle\!\rangle_{\bar{\mathcal{D}}(n,\varphi)} : \phi \in \Gamma_n\}$$
$$\hat{F}(n) = \bigwedge\{\langle\!\langle \varphi \rangle\!\rangle_{\hat{\mathcal{D}}(n,\varphi)} : \phi \in \Gamma_n\}$$

If $n\downarrow$ is not a loop node, then we assign sequent:

$$F(n) \vdash \{\langle P* \rangle \hat{F}(m) : m \in AN(n)\}$$

Otherwise, we assign the sequent:

$$\bar{F}(n) \vdash \{\langle P* \rangle \hat{F}(m) : m \in AN(n)\}$$

Chapter 3

Implementation

This chapter describes the implementation of the proof procedure. Initially the project was written in Haskell but it became clear that the whole process was in fact a series of graph transformations; an imperative language seemed like a better choice.

Functional programming languages are not well suited to manipulating graphs, they're not as naturally expressed as, for example, trees. There has been some work in this area, Andrey Mokhov describes a algebra of graphs[15] from which the Haskell library `algebraic-graphs`[14] was based on. While this approach is elegant, the immaturity of this library meant a large amount of time would be required in order to implement missing features.

The final implementation presented here is written in Rust. Some reasons for choosing Rust (from most to least important):

1. A fully featured graph library, petgraph[9].
2. Proficiency with the language.
3. Great support for functional programming idioms, making it easy to translate the previous attempts.
4. Rust's notable features such as memory safety, zero cost abstractions, etc.

3.1 Formula

A μ-calculus formula is represented as an expression tree shown in figure 3.1. This allows us to easily extract a sub formula or substitute a variable by replacing a reference, the two most common operations. Another approach is to represent a formula as a vector of tokens. The performance gains from improved cache coherence is undermined by the penalty paid on moving and resizing memory when extracting sub formula or applying substitution.

The formula parser uses the popular parser combinator library Nom[10]. Parsing formula will never be a bottle neck for this procedure, so anything more sophisticated would have been a poor use of time.

```rust
pub enum Formula {
    Lit {
        lit: Lit,
        is_negatted: bool,
    },
    Binary {
        op: BinOp,
        lhs: Box<Formula>,
        rhs: Box<Formula>,
    },
    ...
}
```

Figure 3.1: Representation of `Formula`

```rust
pub struct TableauIndice(pub usize);

unsafe impl IndexType for TableauIndice {
    ...
}
```

Figure 3.2: `TableauIndice` type used to index `TableauGraph`

3.2 Refutation Generation

Since we will be dealing with many different graphs at later stages of the procedure, it's important to design our types to prevent incorrect indexing of graphs. For each graph we create a wrapper type (for example, figure 3.2) that nodes and edges of the graph will be indexed by. When we're dealing with several graphs and indices, the compiler will disallow the index of a graph by another graph's indice.

Tableau construction proceeds by recursive calls to `grow` and `branch`. `grow` takes a node and creates child node(s) as per the tableau rules. There may be multiple rules that match so we first shuffle the set of formula, then consider each one in turn. This randomisation gives rise to significant performance improvements in tableau generation.

The `branch` procedure checks whether the child nodes produced by `grow` are tableau axioms, or they form a loop which contains a μ-trace. If either condition holds, we stop developing that branch. If neither condition holds, we backtrack and try the next formula.

Unfortunately, we may end up with some particularly large tableau that will be intractable in later stages. Thus the tableau constructor has a time and node budget. We continuously generate tableau until we have one that is less than the node budget in size or the time has elapsed.

3.3 Stable Tree

`petgraph` has provided the `Graph` data structures used throughout the program, but in section 3.4 we want to use trees. We could include another library or create our own

```rust
#[inline(always)]
pub fn add_child(
    &mut self,
    parent: NodeIndex<Ix>,
    weight: N,
    edge_weight: E,
) -> (NodeIndex<Ix>, EdgeIndex<Ix>) {
    let child = self.graph.add_node(weight);
    let child_edge = self.graph.add_edge(parent, child, edge_weight);
    (child, child_edge)
}
```

Figure 3.3: `add_child` method

tree data structure, however we instead opt to create a wrapper around `Graph`. This wrapper alters the relevant methods to maintain the invariants that makes a graph a tree.

The advantage to this approach is that we can still use the generic graph procedures provided by `petgraph`. We loose some opportunities by not specializing, but raw performance is not the goal of this project.

`StableTree` will be used to represent states of the automaton described in section 3.4. We want to be able to the same node in any two states by the same indice, to do so we must ensure that the id's of other nodes are not altered when a node gets deleted. `StableGraph` maintains this invariant, so we will build the tree wrapper around `StableGraph`, hence the name `StableGraph`.

Most methods are identical to the methods on `StableGraph`, and thus just call the same method on the underlying graph. We don't want to pay any additional overhead by using the wrapper, so we force the compiler to always inline these function calls by adding the attribute `#[inline(always)]` above the relevant functions as seen in figure 3.3.

For a tree, it doesn't make sense to talk about a node with no outgoing edges. We cannot have an `add_node` method, otherwise we may create invalid trees, likewise `add_edge` could create non parent-child relationships. The solution is to always add a node and an edge (to its parent) at the same time. This is presented in figure 3.3.

3.4 Automaton Graph

In the tableau construction stage, we only produce tableau that are refutations, thus by theorem 2.19 we will always be able to construct an automaton graph.

The process of evolving the state of the automaton is as follows:

1. All nodes are coloured white.
2. Depending on which rule was applied we do one of the following:
    - If the $\langle\rangle$ rule was applied to the formula $\langle a\rangle\alpha$, we apply the following transformations to all formula in all nodes of the state:

```rust
let v = self
    .graph
    .node_indices()
    // U is older (or equal) to el(v)
    .filter(|&node_id| match self.edge_label(node_id) {
        Some(edge) => {
            let el = *edge.weight();
            el == u || el.is_older(u)
        }
        None => false,
    })
    // U is in the nl(v)
    .filter(|&node_id| self.graph[node_id].formula_set.contains(&u_con))
    // get the lowest vertex
    .sorted_by_key(|&node_id| -(self.graph[node_id].depth as i32))
    .next()
    // if there is no such vertex, let v be the root of s
    .unwrap_or(self.root);
```

Figure 3.4: Finding v for step 2

  - $[a]\beta \to \beta$ for any β
  - $\langle a\rangle\alpha \to \alpha$
  - delete all other formulas

- If the *const* rule was applied to constant U then let v be the lowest vertex such that:
  - U is older (or equal) to el(v)
  - U is in nl(v)

  If there is no such vertex, let v be the root. In all ancestors replace U with $\alpha(U)$ where $U = \sigma X.\alpha(X)$ in $\mathcal{D}$. Delete U from all *proper* decedents.

  If U or el(v) is a *mu*-constant, we create son w of v where el$(w) = U$ and nl$(w) = \{\alpha(U)\}$. Finally, w is made *prec*-bigger than its brothers with edge labels not younger than U and *prec*-smaller than all other brothers.

- For all other rules, replace the reduced formula with the resulting formulas in all vertices.

3. If a formula f appears in a vertex to the left of v, delete f from v.
4. Delete all empty vertices
5. For any vertex v such that el$(v) = U$ is a μ-constant and nl(v) is the sum of its U-sons, light v green and delete all descendants of v.

Consider v, defined in step 2. Figure 3.4 demonstrates Rust's extensive functional support noted at the start of chapter 3. This is extremely close to what you would write in Haskell (modulo syntax).

At the end of section 2.3.3, we describe a graph $\mathcal{G}_{\varphi_0}$ with states of the automaton as nodes. To produce this graph we start with the root state, then follow each path through

```rust
let state_edge_hash = hash(&(state.clone(), tableau_child_edges));
let search_result = state_hashes.binary_search_by_key(&state_edge_hash, |(_, h)| *h);
if let Ok(idx) = search_result {
    // state already exists in graph, add loop edge
    let (node_idx, _) = state_hashes[idx];
    graph.add_edge(parent_state_idx, node_idx, edge.rule.clone());
    // stop developing this path
    continue;
}
```

Figure 3.5: Hashing state and child edges for quick lookup

```rust
pub fn from_petgraph<G, FN, FE>(graph: G, fmt_node: FN, fmt_edge: FE) -> Self
    where
        G: IntoNodeReferences + IntoEdgeReferences,
        FN: Fn(G::NodeRef) -> GraphVizNode,
        FE: Fn(G::EdgeRef) -> GraphVizEdge,
```

Figure 3.6: Type of the `from_petgraph` function

the tableau producing states following the procedure above. Since tableaux can have loops, this will continue forever, so we need to detect whenever we produce a state that is already in $\mathcal{G}_{\varphi_0}$ so we can add a loop and stop developing that path.

To do this we maintain a sorted list of nodes indices and state hash pairs so we can quickly check whether a state is in $\mathcal{G}_{\varphi_0}$. Note, however, we may produce the same state but the onward paths in the tableau may be different; we hash not only the automaton state, but also the tableau edges. When we need to check whether a state (with its corresponding tableau edges) has already been encountered, we do a binary search on the list. The code is presented in figure 3.5.

3.5 GraphViz

GraphViz[6] is a program that produces graphs in various image formats from its own text format. Unfortunately, the Rust ecosystem does not yet have a GraphViz package with all the features required for this project, so a basic implementation was created from scratch.

Every structure that can produce a graph implements the `RenderGraphviz` trait with the single required method being `fn dot(&self) -> Dot`. `Dot` is constructed by calling `from_petgraph` (figure 3.6) which encodes every node and edge using the provided lambda functions.

`GraphVizNode` (figure 3.7) and `GraphVizEdge` are constructed using the builder pattern. The node must have an `id`, so its base constructor takes it as an argument. All other methods take the `GraphVizNode` by value and return the new `GraphVizNode`, allowing calls to be chained.

```
impl GraphVizNode {
    pub fn new(id: usize) -> Self { ... }
    pub fn with_label(mut self, label: String) -> Self { ... }
    pub fn with_html_label(mut self, label: String) -> Self { ... }
    pub fn with_color(mut self, color: X11Color) -> Self { ... }
    pub fn with_fill_color(mut self, color: X11Color) -> Self { ... }
    pub fn with_style(mut self, style: Style) -> Self { ... }
}
```

Figure 3.7: Methods for building a `GraphVizNode`

3.6 Dependent Types

The implementation shown here is in Rust, but a large amount of time was spent experimenting in Haskell. Much of the previous implementations are superseded by the work above, but its worth mentioning something that could not be done in Rust, dependent types.

Haskell does not have dependent types, though there is a proposal to add them[1]. There is, however, a neat way to to use Generalized algebraic datatypes (GADTs), type families and a host of other extensions to emulate dependent types. Richard A. Eisenberge and Stephanie Weirich introduced a library to generate the boiler plate required to produce these types[12].

One application I experimented with was enforcing the construction of valid proof trees. In figure 3.8 we make use of GADT's and the type level natural numbers `SNat` (generated using the singletons library) to represent formulas. By using this representation we encode the structure of the formula in their type's, for example $\neg a \vee b$ would have the type `Formula (Disj (Neg a) b)`. Finally we can use these types to encode the inference rules, figure 3.8 shows the rules for double negation and De Morgens.

This is a very limited approach to verification, we discuss this problem more in section 5.1.2.

```haskell
data FType :: * where
  Conj ::  FType -> FType -> FType
  Disj :: FType -> FType -> FType
  Neg :: FType -> FType
  Diamond :: n -> FType -> FType
  Prop :: FType
  Var :: FType

data Formula :: FType -> * where
  FConj :: Formula a -> Formula b -> Formula (Conj a b)
  FDisj :: Formula a -> Formula b -> Formula (Disj a b)
  FNeg :: Formula a -> Formula (Neg a)
  FDiamond :: SNat n -> Formula a -> Formula (Diamond n a)
  FProp :: Letter -> Formula Prop
  FVar :: Variable -> Formula Var

doubleNegation :: Formula (Neg (Neg a)) -> Formula a
doubleNegation (FNeg (FNeg f)) = f

demorgens :: Formula (Neg (Disj a b)) -> Formula (Conj (Neg a) (Neg b))
demorgens (FNeg (FDisj l r)) = FConj (FNeg l) (FNeg r)
```

Figure 3.8: Sample of experiments with Singleton types

Chapter 4

Results

Consider the formula $\varphi_0 = \mu X.(p \wedge \neg p) \vee \langle a \rangle X$. This can read as "in finitely many a transitions $p \wedge \neg p$ holds", but $p \wedge \neg p$ is false, so this formula is clearly unsatisfiable. This is perhaps the simplest non-trivial example, so we will present its proof here. Other larger examples can be found on my website[17].

4.1 Tableau

Figure 4.1 is a tableau for φ_0. The first thing to note is that the graph should be read as an infinite tree, i.e. where loops are unrolled into infinite paths. This example has one such infinite path, the edge that starts the loop is highlighted in red.

The μ-trace on the loop is highlighted in blue. That is:

$$\begin{aligned}
&\alpha_0 = (p_0 \wedge \neg p_0) \vee \langle a_0 \rangle (U_0)\\
&\alpha_1 = \langle a_0 \rangle U_0\\
&\alpha_2 = U_0\\
&\alpha_3 = (p_0 \wedge \neg p_0) \vee \langle a_0 \rangle (U_0)\\
&\alpha_4 = \langle a_0 \rangle U_0\\
&\alpha_5 = \ldots\\
&\quad\vdots
\end{aligned}$$

U_0 is the only constant on this trace, and it regenerates infinitely often at $\alpha_3, \alpha_6, \alpha_9, \ldots, \alpha_{3i}$. U_0 is a μ-constant, thus by definition 2.16 this is in fact a μ-trace. The only leaf (index 4) is a tableau axiom by definition 2.13, thus all loops have a μ-trace and all leaves are tableau axioms, thus this tableau is a refutation by definition 2.17.

4.2 Automaton

Figure 4.2 shows the automaton graph produced from the refutation in figure 4.1. By definition 2.17 every infinite path has an infinite μ-trace. Thus by the correctness of

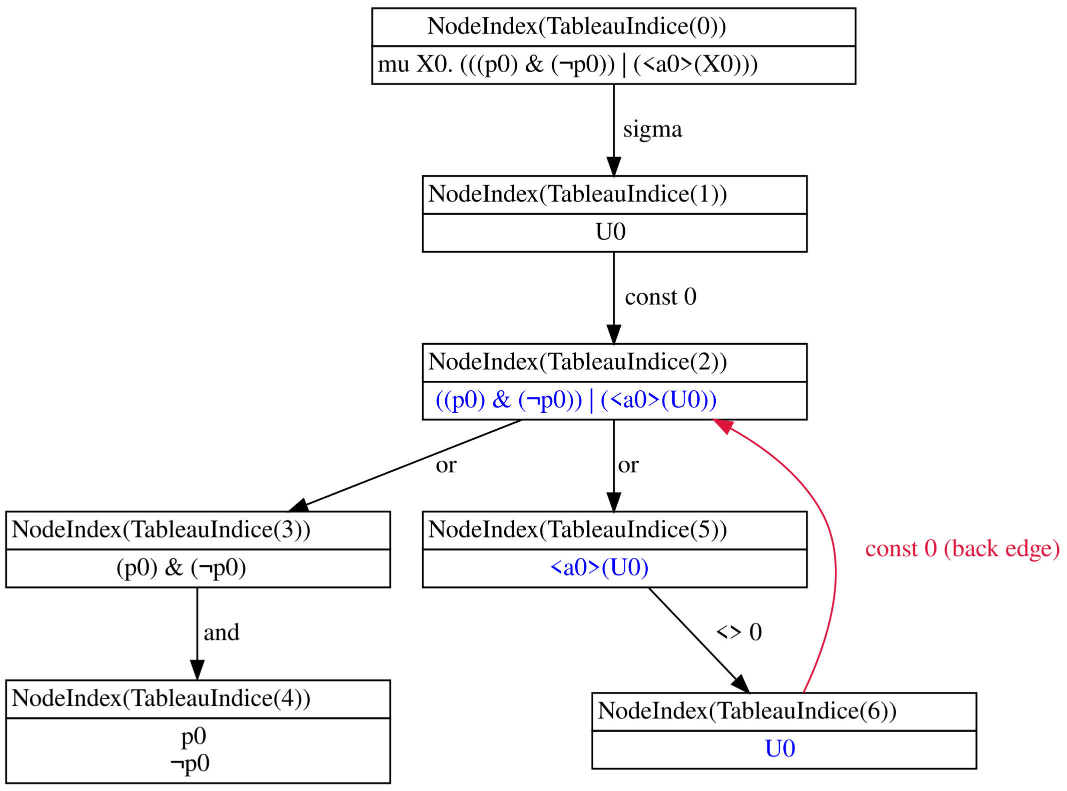
NodeIndex(TableauIndice(0))
mu X0. (((p0) & (¬p0)) | (<a0>(X0)))
sigma
NodeIndex(TableauIndice(1))
U0
const 0
NodeIndex(TableauIndice(2))
((p0) & (¬p0)) | (<a0>(U0))
or
or
NodeIndex(TableauIndice(3))
(p0) & (¬p0)
NodeIndex(TableauIndice(5))
<a0>(U0)
const 0 (back edge)
and
<> 0
NodeIndex(TableauIndice(4))
p0
¬p0
NodeIndex(TableauIndice(6))
U0

Figure 4.1: Tableau for φ_0

the automaton (theorem 2.19) we can always produce an automaton graph from a valid refutation.

Visually we can see that the automaton accepts the μ-trace, since the node that lights green does not disappear on the loop (index 5, 6, 7) and lights green i.o.

4.3 Unwinding

Unwinding the graph in figure 4.2 produced the tree with back edges in figure 4.3. To see why this is correct, we look at the leaves:

- Unwinding nodes 5, 12 and 17 ends with automaton node 4 which has no children, so they will obviously be leaves in the unwinding.
- Unwinding node 18 is a type 1 final node (2.23). The matching prefix is $0, 1, 2, 5, 6, 7, (7, 0)$, where $m\downarrow$ is 7.

  For the first condition, we see that the green vertex v in the automaton graph 4.2 (that is automaton state index 1) has $\text{el}(v) = U_0$. Now, $n\downarrow$ for unwinding node 18 is 6 and U_0 appears in the node label of v in 6, so the first condition is met.

  For the second condition, note that there are only two longer prefixes: one that ends with 5 and one that ends with 6. In nodes $5, 6, 7$ v does not disappear and they form a strongly connected component. It's clear that $\text{Ord}(5) = \text{Ord}(6) =$

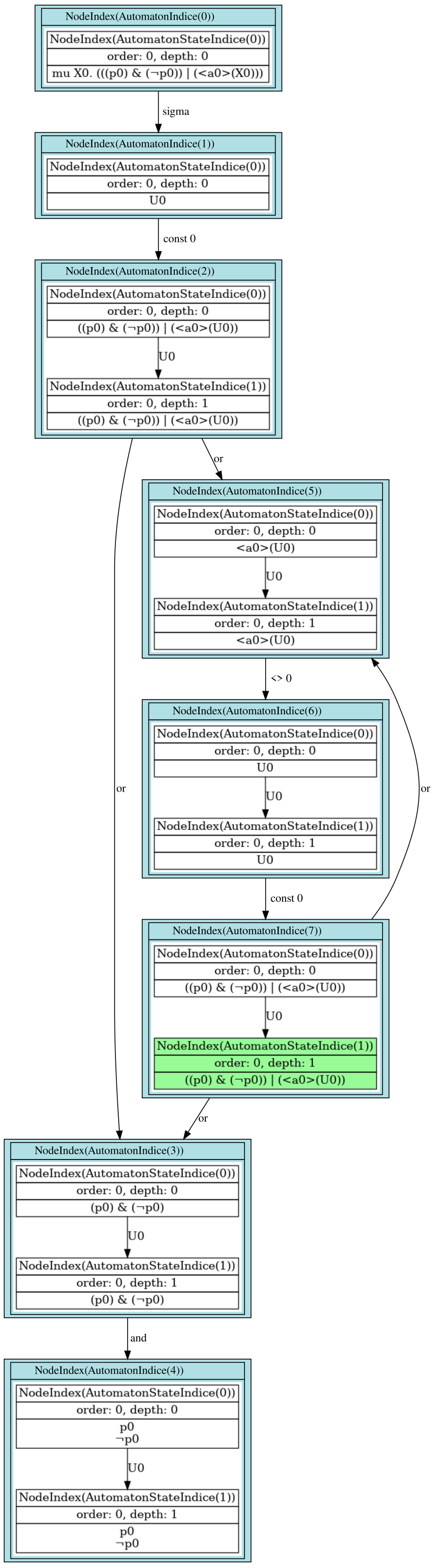
NodeIndex(AutomatonIndice(0))
NodeIndex(AutomatonStateIndice(0))
order: 0, depth: 0
mu X0. (((p0) & (¬p0)) | (<a0>(X0)))
sigma
NodeIndex(AutomatonIndice(1))
U0
const 0
NodeIndex(AutomatonIndice(2))
((p0) & (¬p0)) | (<a0>(U0))
NodeIndex(AutomatonStateIndice(1))
order: 0, depth: 1
or
NodeIndex(AutomatonIndice(5))
<a0>(U0)
<> 0
NodeIndex(AutomatonIndice(6))
NodeIndex(AutomatonIndice(7))
and
NodeIndex(AutomatonIndice(3))
(p0) & (¬p0)
NodeIndex(AutomatonIndice(4))
p0
¬p0

Figure 4.2: Automaton graph for φ_0

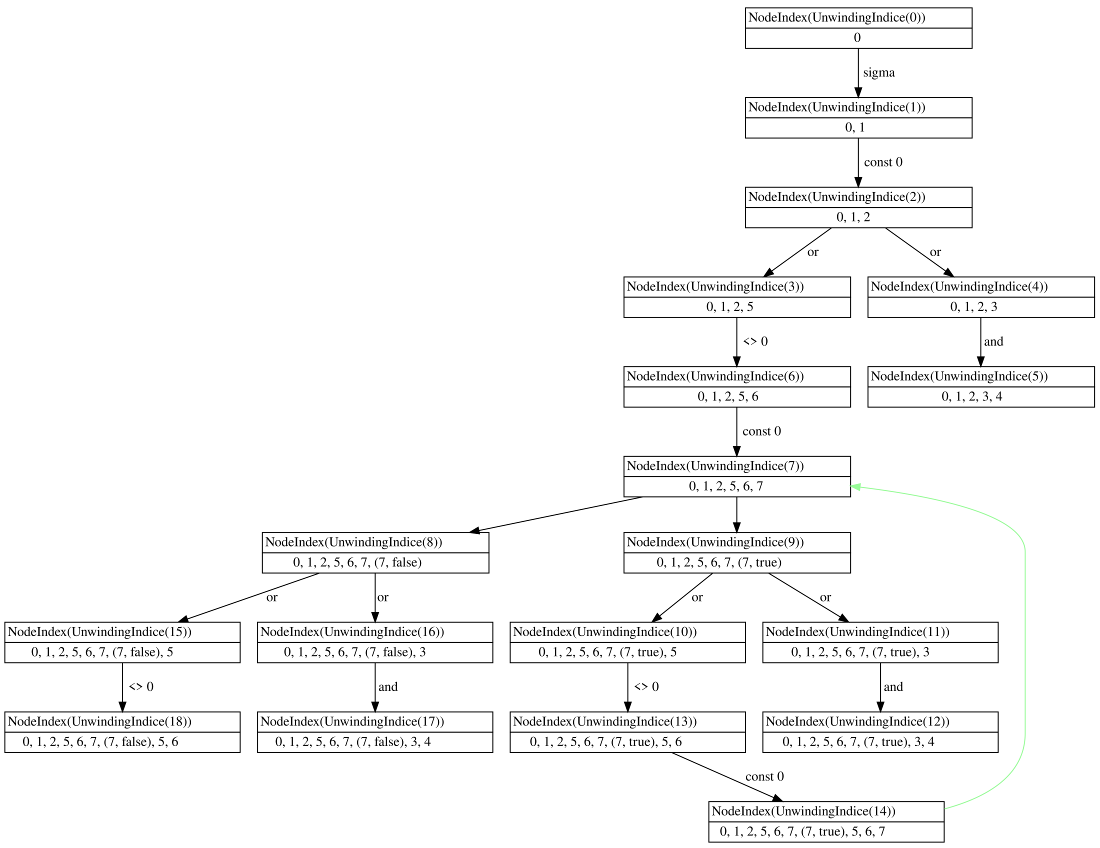
NodeIndex(UnwindingIndice(0))
0
sigma
NodeIndex(UnwindingIndice(1))
0, 1
const 0
NodeIndex(UnwindingIndice(2))
0, 1, 2
or
or
NodeIndex(UnwindingIndice(3))
0, 1, 2, 5
NodeIndex(UnwindingIndice(4))
0, 1, 2, 3
<> 0
and
NodeIndex(UnwindingIndice(6))
0, 1, 2, 5, 6
NodeIndex(UnwindingIndice(5))
0, 1, 2, 3, 4
const 0
NodeIndex(UnwindingIndice(7))
0, 1, 2, 5, 6, 7
NodeIndex(UnwindingIndice(8))
0, 1, 2, 5, 6, 7, (7, false)
NodeIndex(UnwindingIndice(9))
0, 1, 2, 5, 6, 7, (7, true)
or
or
or
or
NodeIndex(UnwindingIndice(15))
0, 1, 2, 5, 6, 7, (7, false), 5
NodeIndex(UnwindingIndice(16))
0, 1, 2, 5, 6, 7, (7, false), 3
NodeIndex(UnwindingIndice(10))
0, 1, 2, 5, 6, 7, (7, true), 5
NodeIndex(UnwindingIndice(11))
0, 1, 2, 5, 6, 7, (7, true), 3
<> 0
and
<> 0
and
NodeIndex(UnwindingIndice(18))
0, 1, 2, 5, 6, 7, (7, false), 5, 6
NodeIndex(UnwindingIndice(17))
0, 1, 2, 5, 6, 7, (7, false), 3, 4
NodeIndex(UnwindingIndice(13))
0, 1, 2, 5, 6, 7, (7, true), 5, 6
NodeIndex(UnwindingIndice(12))
0, 1, 2, 5, 6, 7, (7, true), 3, 4
const 0
NodeIndex(UnwindingIndice(14))
0, 1, 2, 5, 6, 7, (7, true), 5, 6, 7

Figure 4.3: Unwinding for φ_0

Ord$(7) = \{v\}$ by definition of Ord (2.21). Therefore the second condition has been met.

- Unwinding node 14 is a type 2 final node (2.24). The matching prefix is $0, 1, 2, 5, 6, 7, (7, 1)$. There are three longer prefixes that end in $5, 6, 7$. From above we know Ord$(5) =$ Ord$(6) =$ Ord$(7) = \{v\}$, which satisfies the condition for a type 2 final node.

4.4 Assignment

The correctness of the assignment should be clear, apart from perhaps node 14. To see how this can easily be proven note that the formula takes the form $ff \vee \langle a \rangle \alpha \vdash \mu X.\langle a \rangle X \vee \alpha$. The following is proof that a formula of that form is unsatisfiable:

$$
\dfrac{\dfrac{}{ff \vdash \dots} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\alpha \vdash \langle a\rangle(\mu X.\langle a\rangle X \lor \alpha), \alpha}}{\alpha \vdash \langle a\rangle(\mu X.\langle a\rangle X \lor \alpha) \lor \alpha}\,(\lor_r)}{\alpha \vdash \mu X.\langle a\rangle X \lor \alpha}\,(\mu)}{\langle a\rangle\alpha \vdash \langle a\rangle(\mu X.\langle a\rangle X \lor \alpha), \alpha}\,(\langle\rangle)}{\langle a\rangle\alpha \vdash \langle a\rangle(\mu X.\langle a\rangle X \lor \alpha) \lor \alpha}\,(\lor_r)}{\langle a\rangle\alpha \vdash \mu X.\langle a\rangle X \lor \alpha}\,(\mu)}{ff \lor \langle a\rangle\alpha \vdash \mu X.\langle a\rangle X \lor \alpha}\,(\lor_l)
$$

4.5 Further Examples

Further examples of all four stages can be found on my website[17]. This includes:

1. $\mu X_0.(p_1) \land (\langle a_0\rangle((\langle a1\rangle(((X_0) \lor (X_0)) \lor (p_0))) \land (\langle a_0\rangle(X_0))))$
2. $\nu X_0.\mu X_1.\nu X_2.\nu X_3.((\langle a1\rangle(\langle a_1\rangle(X_1))) \land (((\langle a1\rangle([a_1](((p_1)\langle((p_1) \lor ((X_1) \lor (X_2)))) \land (p_0)))) \lor (((\langle a0\rangle(X_0)) \land ((p_1) \land (\langle a0\rangle(X_3)))))$
3. $\mu X_0.\mu X_1.\nu X_2.\langle a_0\rangle(([a_1](((X_1) \land (X_0)) \land (X_1))) \land (([a_1](X_2)) \land (\langle a_1\rangle(X_0))))$

These cannot be included here, since they are far to large; the sequent assignment for the second formula has a width of 209 130 pixels. Attempting to run the program on larger formulas will produce images so large that it crashed all image viewers I tried. These graphs are primarly for debugging, a better output is discussed in section 5.1.1.

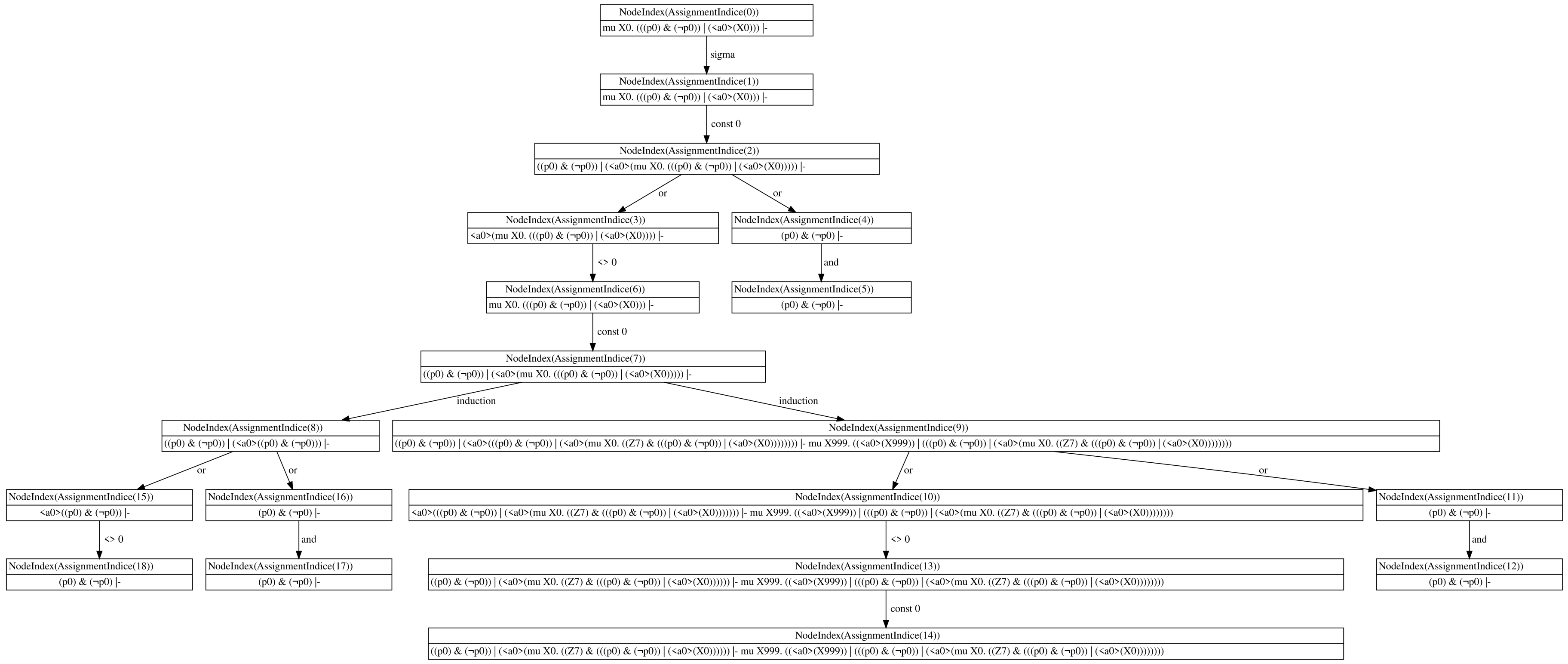
NodeIndex(AssignmentIndice(0))
mu X0. (((p0) & (¬p0)) | (<a0>(X0))) |-
sigma
NodeIndex(AssignmentIndice(1))
mu X0. (((p0) & (¬p0)) | (<a0>(X0))) |-
const 0
NodeIndex(AssignmentIndice(2))
((p0) & (¬p0)) | (<a0>(mu X0. (((p0) & (¬p0)) | (<a0>(X0))))) |-
or
or
NodeIndex(AssignmentIndice(3))
<a0>(mu X0. (((p0) & (¬p0)) | (<a0>(X0)))) |-
NodeIndex(AssignmentIndice(4))
(p0) & (¬p0) |-
<> 0
and
NodeIndex(AssignmentIndice(6))
mu X0. (((p0) & (¬p0)) | (<a0>(X0))) |-
NodeIndex(AssignmentIndice(5))
(p0) & (¬p0) |-
const 0
NodeIndex(AssignmentIndice(7))
((p0) & (¬p0)) | (<a0>(mu X0. (((p0) & (¬p0)) | (<a0>(X0))))) |-
induction
induction
NodeIndex(AssignmentIndice(8))
((p0) & (¬p0)) | (<a0>((p0) & (¬p0))) |-
NodeIndex(AssignmentIndice(9))
((p0) & (¬p0)) | (<a0>(((p0) & (¬p0)) | (<a0>(mu X0. ((Z7) & (((p0) & (¬p0)) | (<a0>(X0)))))))) |- mu X999. ((<a0>(X999)) | (((p0) & (¬p0)) | (<a0>(mu X0. ((Z7) & (((p0) & (¬p0)) | (<a0>(X0))))))))
or
or
or
or
NodeIndex(AssignmentIndice(15))
<a0>((p0) & (¬p0)) |-
NodeIndex(AssignmentIndice(16))
(p0) & (¬p0) |-
NodeIndex(AssignmentIndice(10))
<a0>(((p0) & (¬p0)) | (<a0>(mu X0. ((Z7) & (((p0) & (¬p0)) | (<a0>(X0))))))) |- mu X999. ((<a0>(X999)) | (((p0) & (¬p0)) | (<a0>(mu X0. ((Z7) & (((p0) & (¬p0)) | (<a0>(X0))))))))
NodeIndex(AssignmentIndice(11))
(p0) & (¬p0) |-
<> 0
and
<> 0
and
NodeIndex(AssignmentIndice(18))
(p0) & (¬p0) |-
NodeIndex(AssignmentIndice(17))
(p0) & (¬p0) |-
NodeIndex(AssignmentIndice(13))
((p0) & (¬p0)) | (<a0>(mu X0. ((Z7) & (((p0) & (¬p0)) | (<a0>(X0)))))) |- mu X999. ((<a0>(X999)) | (((p0) & (¬p0)) | (<a0>(mu X0. ((Z7) & (((p0) & (¬p0)) | (<a0>(X0))))))))
NodeIndex(AssignmentIndice(12))
(p0) & (¬p0) |-
const 0
NodeIndex(AssignmentIndice(14))
((p0) & (¬p0)) | (<a0>(mu X0. ((Z7) & (((p0) & (¬p0)) | (<a0>(X0)))))) |- mu X999. ((<a0>(X999)) | (((p0) & (¬p0)) | (<a0>(mu X0. ((Z7) & (((p0) & (¬p0)) | (<a0>(X0))))))))

Figure 4.4: Assignment for φ_0

Chapter 5

Conclusion

5.1 Limitations

5.1.1 Outputting the proof tree

After assigning sequents to the unwinding, we obtain a tree with two properties:

1. Leaves are assigned provable sequents
2. Internal nodes are provable from their children
3. The root is labelled with $\neg\varphi$

However, this isn't quite the definition of proof system we described in section 2.1.2. The transformation is relatively simple but the resulting tree is extremely large.

Attempting to output a tree, such as the one in figure 4.4, for a more complex example is challenging. Using pdfTeX, a widely used LaTeX typesetting program, causes "dimension to large" errors stemming from pdfTeX's unconfigurable dimension constraints.

It's possible that luaTeX, a project based on pdfTeX with embedded Lua scripting, could work but I encountered layout issues that will take some time to solve. It may also be worth considering additional output formats. For example, we could emit the proof in Agda (or some other theorem prover) with a preamble postulating the axioms and inference rules. This has the additional benefit of being able to verify the proof with an external program.

5.1.2 Verification

A fully formally verified completeness procedure was obviously untenable; a formalisation of the LTL model checking algorithm[16] (a much simpler problem) took around 4 person months. However, there are perhaps parts of the system that could be extracted and verified. Theorem provers such as Agda and Isabelle can emit code that could be integrated into the main program.

A proof verifier, that checks the proof tree output, could also increase confidence in the program correctness. This is dependent on outputting the full proof tree as discussed in the section above.

5.1.3 Axioms

As noted in section 1.3, the original goal was to implement Walukiewicz second paper with Kozen's axioms. This is likely to be tackled in part 2 of this project.

5.2 Future work

In addition to the improvements described in the section above, there are a few extensions that can be considered.

Kim G. Larsen, Radu Mardare, and Bingtian Xue introduced an extension of μ-calculus built on top of probabilistic modal logic (PMC) and an axiomatization for the alternation-free fragment[13]. Perhaps the techniques used here could prove the completeness for the full PMC.

Amina Doumane presented a constructive completeness result for linear time μ-calculus[11] which is a restriction of μ-calculus to linear time. Doumane's system lends itself to an implementation much more so than Walukiewicz's procedure.

5.3 Conclusion

We have presented an implementation of Walukiewicz completeness result for the μ-calculus. Each stage, from tableau to automaton to unwinding and assignment, a graph/tree is produced. The result is a proof that a given expression as unsatisfiable.

There is a more direct way of showing satisfiability. μ-calculus formulas have a one-to-one correspondence with alternating parity automata over infinite trees. To show φ is unsatisfiable, it suffices to transform it into an automaton $\mathcal{A}_\varphi$ and show $\mathcal{L}(\mathcal{A}_\varphi) = \emptyset$ (the language emptiness problem). The advantage of the approach described here is that the program emits a proof which can be considered as a certificate that can be verified by a third party.

Bibliography

[1] Adding dependent types to haskell. "https://gitlab.haskell.org/ghc/ghc/-/wikis/dependent-haskell".

[2] Agda. "https://wiki.portal.chalmers.se/agda/pmwiki.php".

[3] Algorithmic Game Theory and its Applications Course (DRPS). "http://www.drps.ed.ac.uk/19-20/dpt/cxinfr11020.htm".

[4] The Coq Proof Assistant. "https://coq.inria.fr/".

[5] Formal Verification Course (DRPS). "http://www.drps.ed.ac.uk/19-20/dpt/cxinfr11129.htm".

[6] Graphviz. "https://graphviz.org".

[7] Isabelle. "http://isabelle.in.tum.de/".

[8] Mu-calculus. "https://github.com/cipib/Mu-calculus".

[9] petgraph. "https://crates.io/crates/petgraph".

[10] Geoffroy Couprie. Nom. "https://crates.io/crates/nom".

[11] Amina Doumane. Constructive completeness for the linear-time μ-calculus. pages 1–12, 06 2017.

[12] Richard Eisenberg and Stephanie Weirich. Dependently typed programming with singletons. *ACM SIGPLAN Notices*, 47, 01 2012.

[13] Kim G. Larsen, Radu Mardare, and Bingtian Xue. Probabilistic Mu-Calculus: Decidability and Complete Axiomatization. In Akash Lal, S. Akshay, Saket Saurabh, and Sandeep Sen, editors, *36th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2016)*, volume 65 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 25:1–25:18, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[14] Andrey Mokhov. algebraic-graphs. "https://hackage.haskell.org/package/algebraic-graphs".

[15] Andrey Mokhov. Algebraic graphs with class (functional pearl). In *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell*, Haskell 2017, page 2–13, New York, NY, USA, 2017. Association for Computing Machinery.

[16] Alexander Schimpf, Stephan Merz, and Jan-Georg Smaus. Construction of büchi automata for ltl model checking verified in isabelle/hol. pages 424–439, 08 2009.

[17] Ben Sheffield. Implementing the completeness proof for the modal μ-calculus. `"https://mrbenshef.co.uk/posts/implementing-the-completeness-proof-for-the-modal-mu-calculus/"`.

[18] I. Walukiewicz. On completeness of the mu-calculus. In *[1993] Proceedings Eighth Annual IEEE Symposium on Logic in Computer Science*, pages 136–146, June 1993.

[19] Igor Walukiewicz. Completeness of kozen's axiomatisation of the propositional μ-calculus. *Information and Computation*, 157(1):142 – 182, 2000.