Linear Temporal Logic: Separation and Translation

Daniel Rebelo de Oliveira

Thesis to obtain the Master of Science Degree in

Mathematics and Applications

Supervisor: Prof. João Filipe Quintas dos Santos Rasga

Examination Committee

Chairperson: Prof. Maria Cristina De Sales Viana Serôdio Sernadas
Supervisor: Prof. João Filipe Quintas dos Santos Rasga
Member of the Committee: Prof. Jaime Arsénio de Brito Ramos

November 2017

Acknowledgments

I would like to thank my advisor, Professor João Rasga, for sticking with me and helping me through this process. And my family, for everything they have done for me.

Resumo

Lógica temporal linear com as modalidades Since e Until tem o mesmo poder expressivo, sobre ordens lineares completas, que um fragmento de primeira-ordem conhecido como FOMLO. Também se sabe que uma lógica temporal linear, assumindo algumas propriedades básicas, tem a mesma expressividade que FOMLO se e só se tem uma propriedade, chamada separação, que qualquer fórmula é equivalente a uma combinação Booleana de fórmulas tal que cada uma delas apenas considera o passado, presente ou futuro.

Neste texto apresenta-se algoritmos simples e as suas implementações para fazer esta separação da lógica temporal linear com Since e Until, sobre ordens discretas e completas, e tradução de FOMLO para esta mesma lógica.

Palavras-chave: Lógica Temporal, LTL, FOMLO, Separação, Tradução, Completude Expressiva

Abstract

Linear temporal logic with Since and Until modalities is expressively equivalent, over the class of complete linear orders, to a fragment of first-order logic known as FOMLO. It turns out that a linear temporal logic, under some basic assumptions, is expressively complete if and only if it has the property, called separation, that every formula is equivalent to a Boolean combination of formulas that each refer only to the past, present or future.

Here we present simple algorithms and their implementations to perform separation of this linear temporal logic with Since and Until, over discrete and complete linear orders, and translation from FOMLO formulas into equivalent temporal logic formulas.

Keywords: Temporal Logic, LTL, FOMLO, Separation, Translation, Expressive Completeness

Contents

Acknowledgments . . . ii
Resumo . . . iii
Abstract . . . iv
List of Figures . . . vii

1 Introduction **1**
1.1 Outline . . . 3

2 Temporal Logic and FOMLO **4**
2.1 Syntax . . . 4
2.2 Semantics . . . 6
2.3 Standard Translation . . . 7

3 Separation **10**
3.1 Algorithm . . . 14
3.1.1 Transformations . . . 17
3.2 Algorithm Correctness . . . 33

4 Translation **54**
4.1 Algorithm . . . 57
4.2 Algorithm Correctness . . . 59
4.3 Extensions . . . 67
4.3.1 LTL . . . 67
4.3.2 Expressively Complete Temporal Logics . . . 68

5 Conclusions **69**

Bibliography **71**

A Code **73**
A.1 Util.hs . . . 73
A.2 BooleanCombination.hs . . . 73
A.3 TL.hs . . . 77
A.4 FOMLO.hs . . . 79

A.5 Separation.hs 81
A.6 Translation.hs 84
A.7 Parse.hs 88
A.8 Pretty.hs 91
A.9 Main.hs 92

List of Figures

3.1 Syntax tree of A . . . 14

3.2 Example diagram . . . 17

3.3 $s \Vdash N$ with $s_{\neg A}$ as witness . . . 18

3.4 $\mathrm{Pred}(s) \Vdash N$ with r as witness . . . 19

3.5 $t_0 \Vdash A\mathcal{S}(B \wedge F\mathcal{U}G)$ with $t_G < t_0$ . . . 20

3.6 $t_0 \Vdash A\mathcal{S}(B \wedge F\mathcal{U}G)$ with $t_G < t_0$, simplified . . . 20

3.7 $t_0 \Vdash A\mathcal{S}(B \wedge F\mathcal{U}G)$ with $t_G = t_0$ . . . 20

3.8 $t_0 \Vdash A\mathcal{S}(B \wedge F\mathcal{U}G)$ with $t_G > t_0$ . . . 21

3.9 $t_0 \Vdash A\mathcal{S}(B \wedge F\mathcal{U}G)$ with $t_G > t_0$, simplified . . . 21

3.10 $t_0 \Vdash (A \vee F\mathcal{U}G)\mathcal{S}(B \wedge F\mathcal{U}G)$ with $t_G < t_0$ . . . 22

3.11 $t_0 \Vdash (A \vee F\mathcal{U}G)\mathcal{S}(B \wedge F\mathcal{U}G)$ with $t_G < t_0$, simplified . . . 22

3.12 Consequences of $s \Vdash N$ . . . 23

3.13 $t_0 \Vdash H_<$ . . . 23

3.14 $\mathrm{Pred}(s) \Vdash N$ with r as witness . . . 23

3.15 $t_0 \Vdash (A \vee F\mathcal{U}G)\mathcal{S}(B \wedge F\mathcal{U}G)$ with $t_G = t_0$ . . . 24

3.16 $t_0 \Vdash (A \vee F\mathcal{U}G)\mathcal{S}(B \wedge F\mathcal{U}G)$ with $t_G > t_0$ . . . 24

3.17 s is the greatest point before t that satisfies B . . . 25

3.18 $t_0 \Vdash A\mathcal{S}(B \wedge \Box G)$ with t_B as witness . . . 26

3.19 t_0 satisfies (1) with $t_T < t_0$ . . . 28

3.20 t_0 satisfies (1) with $t_T < t_0$, updated . . . 28

3.21 t_0 satisfies (1) with $t_T = t_0$ . . . 29

3.22 t_0 satisfies (1) with $t_T > t_0$ . . . 29

3.23 t_0 satisfies (2) . . . 30

3.24 $t_0 \Vdash (A \vee \neg(F\mathcal{U}G))\mathcal{S}(B \wedge F\mathcal{U}G)$ with $t_G < t_0$ . . . 31

3.25 $t_0 \Vdash (A \vee \neg(F\mathcal{U}G))\mathcal{S}(B \wedge F\mathcal{U}G)$ with $t_G = t_0$ . . . 32

3.26 $t_0 \Vdash (A \vee \neg(F\mathcal{U}G))\mathcal{S}(B \wedge F\mathcal{U}G)$ with $t_G > t_0$ . . . 32

3.27 Syntax tree of A . . . 34

3.28 Syntax tree of A with $\mathcal{SPC}$ circled . . . 35

3.29 Syntax tree of A with leaders circled . . . 35

Chapter 1

Introduction

In 1957 in [1], Arthur Prior introduced Tense Logic, by extending propositional logic with two operators P and F, with the intended semantics being such that $P\varphi$ is true if φ was true at some time in the past, and F meaning the same but in the future direction.

In [2], Amir Pnueli proposed the use of this logic in the analysis of properties of computer programs, and while Prior's Tense Logic is strong enough to express many useful properties, there are still properties that are not expressible. Examples of properties of practical interest that cannot be expressed can be found in [3].

The question of how expressive these logics are naturally arises. A way to measure their expressiveness is by considering a fragment of first-order logic which seems to appropriately describe everything that we ought to be able to say, and then checking if all these properties can also be expressed in temporal logic. Or, more precisely, a temporal logic is *expressively complete* if for every first-order formula in this fragment, there is a temporal logic formula that has exactly the same models (and vice-versa from temporal logic to the first-order fragment).

In 1968 in [4], Hans Kamp introduced two temporal operators called Since and Until. With semantics such that $A\,\text{Until}\,B$ means that in the future B will be true and until then A is always true, and similarly for Since but in the direction of the past. He also defined a fragment of first-order now known as FOMLO (First-Order Monadic Logic of Order) which appears appropriate in the sense described above, and showed that if we consider a Dedekind-complete linear model of time, then a temporal logic with just these two operators is expressively complete. A shorter proof of this result can be found in [5].

Still in [3], it is shown that a logic with only the temporal operator Until is expressively complete over the naturals, when considering satisfaction at the initial point of time. Or alternatively over complete and discrete linear orders when considering a fragment of FOMLO that restricts quantifiers such that they can only do bounded quantification and never into the past. This result also implies that, at the initial point, having the additional Since operator does not add expressive power. Although it does not add expressive power, it still gives us something, namely succinctness. In [6] it is shown that there are formulas in the language with both operators such that an equivalent formula using only Until is necessarily at least exponentially larger. Even so, nowadays, this language only with Until is quite popular and has found

many practical applications, for example in the analysis of the correctness of computer programs and systems with concurrent processes. This language is now known as LTL (Linear Temporal Logic).

Dov Gabbay showed that this logic with both Since and Until has the property, called separation, that for every formula there is an equivalent formula consisting of a Boolean combination of formulas that each talk only about the past, present or future. This is mentioned already in [3] and a proof of this over the integers can be found in [7], it was later extended to Dedekind-complete linear time in [8]. This property not only leads to proof a Kamp's theorem, but it turns out that, over any class of linear orders, a temporal logic has the separation property if and only if it is expressively complete, provided that the temporal logic can at least express Prior's P and F operators. This is shown in [8] and [9].

In this text we provide simple algorithms to perform separation and translation. We show that they are sound and complete in the sense given any formula of the temporal language with Since and Until, the separation algorithm constructs a separated formula equivalent to the first over complete and discrete linear time, and given any FOMLO formula, the translation algorithm constructs an equivalent temporal logic formula. In particular we show that these algorithms always terminate.

The translation algorithm is a simplified version of the algorithm that can be extracted from theorem 2.3 in [7]. The same translation algorithm can in fact be used to translate from FOMLO to any expressively complete temporal logic, when provided with a separation algorithm for the logic. We also give an algorithm to translate FOMLO to LTL, which only requires a small addition to the main translation algorithm.

The algorithm that can be extracted from Gabbay's proof of separation over integer time is somewhat complicated. It works by using a family of algorithms that can separate a restricted class of formulas, with the last one being able to perform separation on every formula. Each algorithm considers the whole formula, identifying the most "unseparated" subformulas and substituing certain subformulas with atoms so that the formula as a whole is of a form that can be processed with the previous algorithm. Then it substitutes the previously introduced atoms back with the subformulas they had replaced, and continues recursively. We instead opt for a simple unified recursive algorithm to perform separation, and offload most of the complexity to the proof of correctness. We also simplify some of basic steps in the separation process and provide detailed proofs after introducing some concepts and notation that make it easier to do so.

All of the algorithms we describe have been implemented in the Haskell programming language, and can be found in appendix A.

1.1 Outline

Chapter 2 is an introduction to temporal logic and FOMLO. Defining their syntax and semantics, some notation we will be using and how to translate from temporal logic to FOMLO.

In chapter 3 we concern ourselves with separation of temporal logic. It begins by proving some results about temporal logic, then in section 3.1 the algorithm is fully defined, with section 3.2 being devoted to proving its correctness.

In chapter 4 we turn to the problem of translating from FOMLO to temporal logic. We start with an intuitive explanation of the main difficulty in performing this conversion, define the algorithm in 4.1 and prove its correctness in 4.2. Section 4.3 shows how to use the algorithm to perform the translation for other temporal logics.

Chapter 2

Temporal Logic and FOMLO

In this chapter we introduce the logics, some notation, and show how to translate from temporal logic to FOMLO. For the remainder of this text, unless explicitly mentioned, when we say “temporal logic” or “TL”, we mean the temporal logic defined in this chapter.

2.1 Syntax

Fix a countable set Pred that will serve as both the propositional variables in temporal logic and monadic predicates in FOMLO.

Definition 1. The language of temporal logic $\mathcal{L}_{\text{TL}}$ is defined inductively by:

- $\bot$
- $\top$
- p $\quad (p \in \text{Pred})$
- $\neg A$ $\quad (A \in \mathcal{L}_{TL})$
- $A \vee B$ $\quad (A, B \in \mathcal{L}_{TL})$
- $A \wedge B$ $\quad (A, B \in \mathcal{L}_{TL})$
- $A\mathcal{S}B$ $\quad (A, B \in \mathcal{L}_{TL})$
- $A\mathcal{U}B$ $\quad (A, B \in \mathcal{L}_{TL})$

Definition 2. The set $\mathcal{L}_{\text{LTL}}$ of LTL formulas is defined just like the one of TL, but omitting the $\mathcal{S}$ case.

Definition 3. Some additional temporal operators:

- $\bullet A := \bot\mathcal{S}A$ (Previous)
- $\circ A := \bot\mathcal{U}A$ (Next)
- $\blacklozenge A := \top\mathcal{S}A$ (Eventually in the past)

- $\Diamond A := \top \mathcal{U} A$ (Eventually)
- $\blacksquare A := \neg \blacklozenge \neg A$ (Forever in the past)
- $\Box A := \neg \Diamond \neg A$ (Forever)

Definition 4. We say a formula A is *simple* if it has no outer Boolean structure, that is, if it is of the form p, $B\mathcal{S}C$, or $B\mathcal{U}C$ (for some $p \in$ Pred and $B, C \in \mathcal{L}_{\mathsf{TL}}$).

Definition 5. A formula is called *non-future* if it has no occurrences of $\mathcal{U}$ and *non-past* if it has no occurrences of $\mathcal{S}$.

A *pure past* formula is then a Boolean combination of formulas of the form $A\mathcal{S}B$ where both A and B are non-future and similarly a formula is *pure future* if it is a Boolean combination of formulas the form $A\mathcal{U}B$ with A and B non-past.

A formula is *pure present* if it is a Boolean combination of variables.

Definition 6. A formula is *separated* if it is a Boolean combination of pure formulas.

For FOMLO, we also need a set of variables, we call it Var and assume it is countable and infinite.

Definition 7. The set $\mathcal{L}_{\mathsf{FOMLO}}$ of FOMLO formulas is defined inductively by:

- $\bot$
- $\top$
- $P(x)$ ($P \in$ Pred, $x \in$ Var)
- $(x = y)$ ($x, y \in$ Var)
- $(x < y)$ ($x, y \in$ Var)
- $\neg\alpha$ ($\alpha \in \mathcal{L}_{\mathsf{FOMLO}}$)
- $\alpha \vee \beta$ ($\alpha, \beta \in \mathcal{L}_{\mathsf{FOMLO}}$)
- $\alpha \wedge \beta$ ($\alpha, \beta \in \mathcal{L}_{\mathsf{FOMLO}}$)
- $\exists_x \alpha$ ($x \in$ Var, $\alpha \in \mathcal{L}_{\mathsf{FOMLO}}$)
- $\forall_x \alpha$ ($x \in$ Var, $\alpha \in \mathcal{L}_{\mathsf{FOMLO}}$)

In both TL and FOMLO, we additionally define $\varphi \to \psi$ as an abbreviation for $\neg\varphi \vee \psi$.

We use lower case letters from the latin alphabet to refer to elements of Pred in the context of temporal logic, and upper case latin latters to refer to these same elements in the context of FOMLO. Additionally, each letter refers to the same predicate symbol in both the lower case and upper case versions, for example p and P refer to the same predicate symbol.

We write $\mathrm{Subs}(A)$ to denote the set of subformulas of A.

2.2 Semantics

We will consider interpretation structures over a signature with a binary predicate $<$ and countable unary predicates $P \in$ Pred, with the interpretation of $<$ a *complete* and *discrete* linear order. Where *complete* means that every non-empty bounded above subset has a supremum and every non-empty bounded below subset has an infimum, and *discrete* means that every non-maximal element has a successor and every non-minimal element has a predecessor. These conditions also imply that every infimum is in fact a minimum element of the subset, as if the infimum was not a minimum element, then its successor would be a larger lower bound for the subset. A similar reasoning also allows us to conclude that every non-empty bounded above subset has a greatest element.

Definition 8. An interpretation structure I is a tuple $I = \left\langle D, <^I, \{P^I\}_{P \in \mathsf{Pred}} \right\rangle$ where D is a non-empty set called the domain of I, $<^I$ is a complete and discrete linear order over D, and each $P^I \subseteq D$.

We write $\mathrm{Domain}(I)$ to refer to this D.

In temporal logic, we use the notation $I, (t, s) \Vdash A$ to mean that all the points in the interval (t, s) satisfy A. That is, for all $r^{>t}_{<s}$: $I, r \Vdash A$. And similarly for $[t, s)$, $(t, s]$ and $[t, s]$, which mean $r^{\geq t}_{<s}$, $r^{>t}_{\leq s}$ and $r^{\geq t}_{\leq s}$, respectively. When we use $\nVdash$ with this interval notation, we mean that *no* point in the interval satisfies the formula, it does not mean that some point in the interval fails to satisfy.

This use of $<$ should in fact have been $<^I$. We will continue to write $<$ to refer to $<^I$ when there is no risk of confusion. We might also write $t \in I$ to mean $t \in \mathrm{Domain}(I)$.

Definition 9. Let I be an interpretation structure and $t \in I$. Then we define satisfaction of a temporal logic formula by the structure I at the point t by:

- $I, t \nVdash_{TL} \bot$
- $I, t \Vdash_{TL} \top$
- $I, t \Vdash_{TL} p$ if and only if $t \in P^I$
- $I, t \Vdash_{TL} \neg A$ if and only if $I, t \nVdash_{TL} A$
- $I, t \Vdash_{TL} A \vee B$ if and only if $I, t \Vdash_{TL} A$ or $I, t \Vdash_{TL} B$
- $I, t \Vdash_{TL} A \wedge B$ if and only if $I, t \Vdash_{TL} A$ and $I, t \Vdash_{TL} B$
- $I, t \Vdash_{TL} A\mathcal{S}B$ if and only if there is an $s <^I t$ such that: $I, s \Vdash_{TL} B$ and $I, r \Vdash_{TL} A$ for all $r \in (s, t)$
- $I, t \Vdash_{TL} A\mathcal{U}B$ if and only if there is an $s >^I t$ such that: $I, s \Vdash_{TL} B$ and $I, r \Vdash_{TL} A$ for all $r \in (t, s)$

Naturally, in the last two cases, s and r are over the domain of I.

We call the s in the definition of $\mathcal{S}$ ($\mathcal{U}$) satisfaction a *witness* for $A\mathcal{S}B$ ($A\mathcal{U}B$) at t.

Satisfaction for FOMLO is defined in the usual way.

Definition 10. An assignment ρ into an interpretation structure I is a function $\rho : \mathsf{Var} \to \mathrm{Domain}(I)$.

We use the notation $[x \mapsto x_0, y \mapsto y_0]$ to denote an assignment ρ such that $\rho(x) = x_0$ and $\rho(y) = y_0$. And the notation $\rho[x \mapsto x_0, y \mapsto y_0]$ for an assignment identical to ρ except possibly at x and y, to which it assigns x_0 and y_0, respectively.

Definition 11. Let I be an interpretation structure and ρ an assignment into I, then:

- $I, \rho \nVdash_{FOMLO} \bot$
- $I, \rho \Vdash_{FOMLO} \top$
- $I, \rho \Vdash_{FOMLO} P(x)$ if and only if $\rho(x) \in P^I$
- $I, \rho \Vdash_{FOMLO} (x = y)$ if and only if $\rho(x) = \rho(y)$
- $I, \rho \Vdash_{FOMLO} (x < y)$ if and only if $\rho(x) <^I \rho(y)$
- $I, \rho \Vdash_{FOMLO} \neg\alpha$ if and only if $I, \rho \nVdash_{FOMLO} \alpha$
- $I, \rho \Vdash_{FOMLO} \alpha \vee \beta$ if and only if $I, \rho \Vdash_{FOMLO} \alpha$ or $I, \rho \Vdash_{FOMLO} \beta$
- $I, \rho \Vdash_{FOMLO} \alpha \wedge \beta$ if and only if $I, \rho \Vdash_{FOMLO} \alpha$ and $I, \rho \Vdash_{FOMLO} \beta$
- $I, \rho \Vdash_{FOMLO} \exists_x \alpha$ iff there is an assignment ρ' identical to ρ except possibly at x such that $I, \rho' \Vdash_{FOMLO} \alpha$
- $I, \rho \Vdash_{FOMLO} \forall_x \alpha$ iff for all assignments ρ' identical to ρ except possibly at x: $I, \rho' \Vdash_{FOMLO} \alpha$

In both TL and FOMLO, we may simply write $\Vdash$ when it is clear which logic we mean. And in TL we may write $t \Vdash A$ instead of $I, t \Vdash A$ when it is clear from the context which interpretation we are considering.

We use $\equiv$ to denote semantic equivalence of formulas. That is, in the case of temporal logic, $A \equiv B$ means that for all interpretations I and points t of I: $I, t \Vdash A$ if and only if $I, t \Vdash B$. And in the case of FOMLO, $\varphi \equiv \psi$ means that for all interpretations I and assignments ρ: $I, \rho \Vdash \varphi$ if and only if $I, \rho \Vdash \psi$.

2.3 Standard Translation

Translating from temporal logic to FOMLO is quite easy, all we have to do is write the definition of satisfaction of each connective in FOMLO. This is known as the *standard translation*.

Definition 12. $\mathrm{ST} : \mathsf{Var} \to \mathcal{L}_{\mathsf{TL}} \to \mathcal{L}_{\mathsf{FOMLO}}$ is defined by:

$$\mathrm{ST}_t(\bot) := \bot$$

$$\mathrm{ST}_t(\top) := \top$$

$$\mathrm{ST}_t(p) := P(t)$$

$$\mathrm{ST}_t(\neg A) := \neg\,\mathrm{ST}_t(A)$$

$$\mathrm{ST}_t(A \vee B) := \mathrm{ST}_t(A) \vee \mathrm{ST}_t(B)$$

$$\mathrm{ST}_t(A \wedge B) := \mathrm{ST}_t(A) \wedge \mathrm{ST}_t(B)$$

$$\mathrm{ST}_t(A\mathcal{S}B) := \exists_s\,(s < t \wedge \mathrm{ST}_s(B) \wedge \forall_r\,(s < r \wedge r < t \;\rightarrow\; \mathrm{ST}_r(A)))$$

$$\mathrm{ST}_t(A\mathcal{U}B) := \exists_s\,(t < s \wedge \mathrm{ST}_s(B) \wedge \forall_r\,(t < r \wedge r < s \;\rightarrow\; \mathrm{ST}_r(A)))$$

The standard translation produces a FOMLO formula that is equivalent to the temporal formula in the following sense:

Proposition 13. *Let I be an interpretation structure, t_0 a point of I, $A \in \mathcal{L}_{TL}$ and $t \in$ Var. Then:*

$$I, t_0 \Vdash_{TL} A \ \text{ if and only if } \ I, [t \mapsto t_0] \Vdash_{FOMLO} \mathrm{ST}_t(A)$$

Proof. The proof is by induction the structure of A.

The first three cases are immediate.

The negation, disjunction and conjunction cases are a direct use of the induction hypothesis.

For the $\mathcal{S}$ case, $I, t_0 \Vdash A = B\mathcal{S}C$ if and only if there is some s_0 such that $s_0 < t_0$, $I, s_0 \Vdash C$ and $I, (s_0, t_0) \Vdash B$.

The first condition is equivalent to $I, [t \mapsto t_0, s \mapsto s_0] \Vdash (s < t)$.

The second one, $I, s_0 \Vdash C$, is equivalent to $I, [t \mapsto t_0, s \mapsto s_0] \Vdash \mathrm{ST}_s(C)$ by the induction hypothesis.

The third one, $I, (s_0, t_0) \Vdash B$, is the same as saying that for all r_0 such that $s_0 < r_0$ and $r_0 < t_0$ we have $I, r_0 \Vdash B$. By the induction hypothesis, $I, r_0 \Vdash B$ if and only if $I, [t \mapsto t_0, s \mapsto s_0, r \mapsto r_0] \Vdash \mathrm{ST}_r(B)$, and then we get $I, [t \mapsto t_0, s \mapsto s_0] \Vdash \forall_r\, (t < r \wedge r < s \rightarrow \mathrm{ST}_r(B))$.

Joining these three together, we get $I, [t \mapsto t_0, s \mapsto s_0] \Vdash s < t \wedge \mathrm{ST}_s(C) \wedge \forall_r\, (t < r \wedge r < s \rightarrow \mathrm{ST}_r(B))$, which is equivalent to $I, [t \mapsto t_0] \Vdash \exists_s\, (s < t \wedge \mathrm{ST}_s(C) \wedge \forall_r\, (t < r \wedge r < s \rightarrow \mathrm{ST}_r(B)))$ by definition of satisfaction.

The $\mathcal{U}$ case is similar to the $\mathcal{S}$ one. $\square$

Chapter 3

Separation

In this chapter we define and prove the correctness of an algorithm that separates a temporal logic formula. But before we show the algorithm, we prove some crucial properties of temporal logic.

Definition 14. The *dual* of a formula is obtained by replacing every $\mathcal{S}$ with an $\mathcal{U}$, and vice-versa. We write $\mathrm{Dual}(A)$ for the dual of A.

We now show that this Dual has some expected properties, which allows us to only worry about $\mathcal{U}$ inside $\mathcal{S}$ and get the cases where $\mathcal{S}$ is inside $\mathcal{U}$ "for free".

Definition 15. Given an interpretation structure I, I^{op} is called the opposite structure. It is defined just like I but where the interpretation of $<$ is precisely the opposite order of the one of I. That is:

$$\mathrm{Domain}(I^{\mathsf{op}}) = \mathrm{Domain}(I)$$

$$P^{I^{\mathsf{op}}} = P^{I} \quad \text{(for every } P \in \mathsf{Pred}\text{)}$$

$$<^{I^{\mathsf{op}}} = \left\{ \langle y, x \rangle \mid \langle x, y \rangle \in <^{I} \right\}$$

Proposition 16. *For every formula* A*:* $\mathrm{Dual}(\mathrm{Dual}(A)) = A$.

Proof. It's clear that swapping $\mathcal{S}$ and $\mathcal{U}$ twice results in the same formula. $\square$

Proposition 17. *Let I be an interpretation structure, t a point of I, and A a formula. Then*

$$I, t \Vdash A \text{ if and only if } I^{\mathsf{op}}, t \Vdash \mathrm{Dual}(A)$$

Proof. We prove this by induction on the structure of A.

If $A = \bot$ or $A = \top$ then it's trivial.

If $A = p$: $I, t \Vdash p$ iff $t \in P^I = P^{I^{\mathsf{op}}}$ iff $I^{\mathsf{op}}, t \Vdash p = \mathrm{Dual}(p)$.

If $A = \neg B$ or $A = B \vee C$ or $A = B \wedge C$ (we examine the $\vee$ case, the others being very similar):

	$I, t \Vdash B \vee C$	
iff	$I, t \Vdash B$ or $I, t \Vdash C$	(by definition of $\Vdash$)
iff	$I^{\mathsf{op}}, t \Vdash \mathrm{Dual}(B)$ or $I^{\mathsf{op}}, t \Vdash \mathrm{Dual}(C)$	(induction hypothesis)
iff	$I^{\mathsf{op}}, t \Vdash \mathrm{Dual}(B) \vee \mathrm{Dual}(C) = \mathrm{Dual}(B \vee C)$	(by definition of $\Vdash$)

If $A = B\mathcal{S}C$: $I, t \Vdash B\mathcal{S}C$ is by definition equivalent to the existence of some $s \in \mathrm{Domain}(I) = \mathrm{Domain}(I^{\mathsf{op}})$ such that $s <^I t$, $I, s \Vdash C$, and for all $r \in I$, if $s <^I r <^I t$ then $I, t \Vdash B$. We then have:

	$s <^I t$ $I, s \Vdash C$ for all $r \in I$, if $s <^I r <^I t$ then $I, t \Vdash B$	
iff	$s >^{I^{\mathsf{op}}} t$ $I, s \Vdash C$ for all $r \in I^{\mathsf{op}}$, if $s >^{I^{\mathsf{op}}} r >^{I^{\mathsf{op}}} t$ then $I, t \Vdash B$	(by definition of $_^{\mathsf{op}}$)
iff	$s >^{I^{\mathsf{op}}} t$ $I^{\mathsf{op}}, s \Vdash \mathrm{Dual}(C)$ for all $r \in I^{\mathsf{op}}$, if $s >^{I^{\mathsf{op}}} r >^{I^{\mathsf{op}}} t$ then $I^{\mathsf{op}}, t \Vdash \mathrm{Dual}(B)$	(induction hypothesis)

Which is the definition of $I^{\mathsf{op}}, t \Vdash \mathrm{Dual}(B)\,\mathcal{U}\,\mathrm{Dual}(C) = \mathrm{Dual}(B\mathcal{S}C)$.

If $A = B\mathcal{U}C$:

	$I, t \Vdash B\mathcal{U}C$	
iff	$(I^{\mathsf{op}})^{\mathsf{op}}, t \Vdash \mathrm{Dual}(\mathrm{Dual}(B\mathcal{U}C))$	($I = (I^{\mathsf{op}})^{\mathsf{op}}$ and proposition 16)
iff	$(I^{\mathsf{op}})^{\mathsf{op}}, t \Vdash \mathrm{Dual}(\mathrm{Dual}(B)\mathcal{S}\,\mathrm{Dual}(C))$	($\mathrm{Dual}(B\mathcal{U}C) = \mathrm{Dual}(B)\mathcal{S}\,\mathrm{Dual}(C)$)
iff	$I^{\mathsf{op}}, t \Vdash \mathrm{Dual}(B)\mathcal{S}\,\mathrm{Dual}(C) = \mathrm{Dual}(B\mathcal{U}C)$	(by the $\mathcal{S}$ case just above)

□

Proposition 18. *For all $A, B \in \mathcal{L}_{TL}$: $A \equiv B$ if and only if $\mathrm{Dual}(A) \equiv \mathrm{Dual}(B)$*

Proof. We first show that $A \equiv B$ implies $\mathrm{Dual}(A) \equiv \mathrm{Dual}(B)$:

	$I, t \Vdash \mathrm{Dual}(A)$	
iff	$I^{\mathsf{op}}, t \Vdash \mathrm{Dual}(\mathrm{Dual}(A)) = A$	(propositions 17 and 16)
iff	$I^{\mathsf{op}}, t \Vdash B$	(by assumption)
iff	$(I^{\mathsf{op}})^{\mathsf{op}} = I, t \Vdash \mathrm{Dual}(B)$	(proposition 17)

For the other direction, using the result just above and proposition 16:

$$\mathrm{Dual}(A) \equiv \mathrm{Dual}(B) \implies \mathrm{Dual}(\mathrm{Dual}(A)) \equiv \mathrm{Dual}(\mathrm{Dual}(B)) \iff A \equiv B$$

□

Thus far we have seen that dual formulas really are dual in the expected sense.

We now prove two important distribution results, namely that $\mathcal{S}$ distributes over $\wedge$ on the right and over $\vee$ on the left. Naturally, by duality, $\mathcal{U}$ also distributes over $\wedge$ on the right and $\vee$ on the left.

Proposition 19. *For all* $A, B, C \in \mathcal{L}_{TL}$*:* $(A \wedge B)\mathcal{S}C \equiv A\mathcal{S}C \wedge B\mathcal{S}C$

Proof.

($\Rightarrow$) Let $t \Vdash (A \wedge B)\mathcal{S}C$.

Then there is a $t_C < t$ such that $t_C \Vdash C$, $(t_C, t) \Vdash A$ and $(t_C, t) \Vdash B$.

This same t_C is also a witness for both $A\mathcal{S}C$ and $B\mathcal{S}C$ at t.

($\Leftarrow$) Let $t \Vdash A\mathcal{S}C \wedge B\mathcal{S}C$

Then there are $t_C, s_C < t$ that satisfy C and $(t_C, t) \Vdash A$ and $(s_C, t) \Vdash B$.

Let r be the greatest of t_C and s_C.

Then $r \Vdash C$. And $(r, t) \subseteq (t_C, t) \cap (s_C, t)$, since $t_C, s_C \leq r$, so $(r, t) \Vdash A \wedge B$.

This means that r is a witness for $(A \wedge B)\mathcal{S}C$ at t. □

Corollary 20 ($\mathcal{S}$ right-distributes over $\wedge$)**.** *For all* $A, B \in \mathcal{L}_{TL}$*:*

$$\left(\bigwedge_{i=1}^{n} A_i\right)\mathcal{S}B \equiv \bigwedge_{i=1}^{n} (A_i \mathcal{S} B)$$

Proof. Induction on n using proposition 19. □

Corollary 21 ($\mathcal{U}$ right-distributes over $\wedge$)**.** *For all* $A, B \in \mathcal{L}_{TL}$*:*

$$\left(\bigwedge_{i=1}^{n} A_i\right)\mathcal{U}B \equiv \bigwedge_{i=1}^{n} (A_i \mathcal{U} B)$$

Proof.

$$
\begin{aligned}
\left(\bigwedge_{i=1}^{n} A_i\right)\mathcal{U}B &= \mathrm{Dual}(\mathrm{Dual}(\left(\bigwedge_{i=1}^{n} A_i\right)\mathcal{U}B)) && \text{(proposition 16)}\\
&= \mathrm{Dual}(\left(\bigwedge_{i=1}^{n} \mathrm{Dual}(A_i)\right)\mathcal{S}\,\mathrm{Dual}(B)) && \text{(by definition of Dual)}\\
&\equiv \mathrm{Dual}(\bigwedge_{i=1}^{n} (\mathrm{Dual}(A_i)\mathcal{S}\,\mathrm{Dual}(B))) && \text{(by corollary 20 and proposition 18)}\\
&= \mathrm{Dual}(\mathrm{Dual}(\bigwedge_{i=1}^{n} (A_i\mathcal{U}B))) && \text{(by definition of Dual)}\\
&= \bigwedge_{i=1}^{n} (A_i\mathcal{U}B) && \text{(proposition 16)}
\end{aligned}
$$

□

Proposition 22. *For all* $A, B, C \in \mathcal{L}_{TL}$*:* $A\mathcal{S}(B \vee C) \equiv A\mathcal{S}B \vee A\mathcal{S}C$

Proof.

($\Rightarrow$) Let $t \Vdash A\mathcal{S}(B \vee C)$.

Then there is a $t_T < t$ such that $t_T \Vdash B \vee C$ and $(t_T, t) \Vdash A$.

If t_T satisfies B then $t \Vdash A\mathcal{S}B$, similarly if t_T satisfies C then $t \Vdash A\mathcal{S}C$.

($\Leftarrow$) Let $t \Vdash A\mathcal{S}B \vee A\mathcal{S}C$.

There's either a t_B such that $t_B \Vdash B$ and $(t_B, t) \Vdash A$, or a t_C such that $t_C \Vdash C$ and $(t_C, t) \Vdash A$.

Either way, both t_B and t_C satisfy $B \vee C$ as well and are witnesses for $A\mathcal{S}(B \vee C)$ at t. □

Corollary 23 ($\mathcal{S}$ left-distributes over $\vee$)**.** *For all* $A, B \in \mathcal{L}_{TL}$*:*

$$A\mathcal{S}\left(\bigvee_{i=1}^{n} B_i\right) \equiv \bigvee_{i=1}^{n} (A\mathcal{S}B_i)$$

Proof. Induction on n using proposition 22. □

Corollary 24 ($\mathcal{U}$ left-distributes over $\vee$)**.** *For all* $A, B \in \mathcal{L}_{TL}$*:*

$$A\mathcal{U}\left(\bigvee_{i=1}^{n} B_i\right) \equiv \bigvee_{i=1}^{n} (A\mathcal{U}B_i)$$

Proof. Similar to the proof of corollary 21, using corollary 23 and proposition 18. □

3.1 Algorithm

The algorithm works by recursively separating the immediate subformulas, which is all that is required for a Boolean operator. Instead of worrying about both $\mathcal{S}$ and $\mathcal{U}$, the $\mathcal{U}$ case is handled by duality and the bulk of the algorithm is on how to separate a $\mathcal{S}$.

When trying to separate a $\mathcal{S}$, the distribution results we proved earlier help in the following way: Consider a formula $A\mathcal{S}B$ where A and B are already separated. A and B can be an arbitrary Boolean combination of simple pure formulas. If we convert the outer Boolean structure of A to conjunctive normal form and the one of B to disjunctive normal form, we can then use distribution to "split" $A\mathcal{S}B$ into a Boolean combination of formulas of the form $C\mathcal{S}D$, where the outer Boolean structures are much simpler, namely C is a disjunctive clause of simple pure formulas and D is a conjunctive clause of simple pure formulas. We then only have to worry about "pulling" the simple pure $\mathcal{U}$s from inside the $\mathcal{S}$ in this narrower case where the left side is a disjunctive clause and the right side is a conjunctive clause. We then eliminate one $\mathcal{U}$ from inside the $\mathcal{S}$, removing it from both sides at once. There are eight possibilities for each, it can occur only on the left, or only on the right, or in both, and it can appear negated or not. Having pulled out one of the $\mathcal{U}$s, we continue separating the result recursively.

Before we show the algorithm, we introduce the concept of path and paths of a formula. This is necessary later for the proof of correctness, but also allows us to define a bit more conveniently a minor thing needed for the algorithm, so we introduce it now.

Definition 25. A *path* is a finite sequence over $\{\mathcal{S},\mathcal{U}\}$. We use $\langle\rangle$ to denote the empty path.

To avoid confusion, we call a path (in the usual sense of graph theory) on the syntax tree of a formula, a *syntax path*. And we call the path corresponding to a syntax path the sequence obtained by considering the labels on the syntax path and then deleting the labels outside of $\{\mathcal{S},\mathcal{U}\}$.

For example, take $A = p\mathcal{S}q \wedge \top\mathcal{U}(\bot\mathcal{S}r)$. It has the following syntax tree:

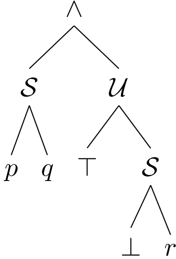

Diagram 3.1: Syntax tree of A

In this example, the path corresponding to the syntax path that starts from the root and goes left twice is $\mathcal{S}$, and the one from the root to the right twice and then left is $\mathcal{U}\mathcal{S}$.

This leads to the definition of the paths of a formula.

Definition 26. Given a formula A, $\mathrm{Paths}(A)$ is the set of paths corresponding with the syntax paths from the root of the syntax tree of A to a leaf. More precisely:

$$\mathrm{Paths}(\bot) := \emptyset$$

$$\mathrm{Paths}(\top) := \emptyset$$

$$\mathrm{Paths}(p) := \emptyset$$

$$\mathrm{Paths}(\neg B) := \mathrm{Paths}(B)$$

$$\mathrm{Paths}(B \vee C) := \mathrm{Paths}(B) \cup \mathrm{Paths}(C)$$

$$\mathrm{Paths}(B \wedge C) := \mathrm{Paths}(B) \cup \mathrm{Paths}(C)$$

$$\mathrm{Paths}(B\mathcal{S}C) := \{\mathcal{S}\pi \mid \pi \in \mathrm{Paths}(B) \cup \mathrm{Paths}(C)\}$$

$$\mathrm{Paths}(B\mathcal{U}C) := \{\mathcal{U}\pi \mid \pi \in \mathrm{Paths}(B) \cup \mathrm{Paths}(C)\}$$

The following definition, used in the algorithm, is why we needed to define Paths beforehand.

Definition 27. The *temporal depth* of a formula A is the maximum length of a path of A.

We can finally present the algorithm, the eight transformations ($\mathcal{T}_1$, $\mathcal{T}_2$, etc) are defined afterwards.

When we write $(\neg?)$ in the algorithm, we mean to allow the possibility of a negation occurring there.

The normal form conversions are only over the outer Boolean structure of a formula, treating all simple formulas as atoms. And we assume that a normal form has no repeated literals and no complementary literals.

procedure Sep(X)

^{0}**if** X is separated **then return** X

^{1}**else if** $X = \neg A$ **then return** $\neg$Sep(A)

^{2}**else if** $X = A \vee B$ **then return** Sep(A) $\vee$ Sep(B)

^{3}**else if** $X = A \wedge B$ **then return** Sep(A) $\wedge$ Sep(B)

^{4}**else if** $X = A\mathcal{S}B$ with A a separated disjunctive clause, and B a separated conjunctive clause

let $\bigvee_{i=1}^{m} (\neg?)A_i := A$

let $\bigwedge_{j=1}^{n} (\neg?)B_j := B$

let $\mathbf{C} := \{(\neg?)A_i \mid A_i$ is a pure future formula$, i \in [1,m]\}$

let $\mathbf{A} := \{(\neg?)A_i \mid i \in [1,m]\} - \mathbf{C}$

let $\mathbf{D} := \{(\neg?)B_j \mid B_j$ is a pure future formula$, j \in [1,n]\}$

let $\mathbf{B} := \{(\neg?)B_j \mid j \in [1,n]\} - \mathbf{D}$ $\qquad (X \equiv (\bigvee \mathbf{A} \vee \bigvee \mathbf{C})\mathcal{S}(\bigwedge \mathbf{B} \wedge \bigwedge \mathbf{D}))$

let $(\neg?)(F\mathcal{U}G) \in \mathbf{C} \cup \mathbf{D}$ with maximal temporal depth in $\mathbf{C} \cup \mathbf{D}$

let $A' := \bigvee((\mathbf{A} \cup \mathbf{C}) - \{F\mathcal{U}G, \neg(F\mathcal{U}G)\})$

let $B' := \bigwedge((\mathbf{B} \cup \mathbf{D}) - \{F\mathcal{U}G, \neg(F\mathcal{U}G)\})$

$^{4.1}$**if** $F\mathcal{U}G \in \mathbf{C}$ and $F\mathcal{U}G, \neg(F\mathcal{U}G) \notin \mathbf{D}$ $\qquad (X \equiv (A' \vee F\mathcal{U}G)\mathcal{S}B')$

return Sep($\mathcal{T}_1\,(A', B', F, G)$)

[4.2]**else if** $F\mathcal{U}G, \neg\,(F\mathcal{U}G) \notin \mathbf{C}$ and $F\mathcal{U}G \in \mathbf{D}$ $\qquad (X \equiv A'\mathcal{S}(B' \wedge F\mathcal{U}G))$

return Sep($\mathcal{T}_2\,(A', B', F, G,)$)

[4.3]**else if** $F\mathcal{U}G \in \mathbf{C}$ and $F\mathcal{U}G \in \mathbf{D}$ $\qquad (X \equiv (A' \vee F\mathcal{U}G)\mathcal{S}(B' \wedge F\mathcal{U}G))$

return Sep($\mathcal{T}_3\,(A', B', F, G)$)

[4.4]**else if** $\neg\,(F\mathcal{U}G) \in \mathbf{C}$ and $F\mathcal{U}G, \neg\,(F\mathcal{U}G) \notin \mathbf{D}$ $\qquad (X \equiv (A' \vee \neg\,(F\mathcal{U}G))\mathcal{S}B')$

return Sep($\mathcal{T}_4\,(A', B', F, G)$)

[4.5]**else if** $F\mathcal{U}G, \neg\,(F\mathcal{U}G) \notin \mathbf{C}$ and $\neg\,(F\mathcal{U}G) \in \mathbf{D}$ $\qquad (X \equiv A'\mathcal{S}(B' \wedge \neg\,(F\mathcal{U}G)))$

return Sep($\mathcal{T}_5\,(A', B', F, G,)$)

[4.6]**else if** $F\mathcal{U}G \in \mathbf{C}$ and $\neg\,(F\mathcal{U}G) \in \mathbf{D}$ $\qquad (X \equiv (A' \vee F\mathcal{U}G)\mathcal{S}(B' \wedge \neg\,(F\mathcal{U}G)))$

return Sep($\mathcal{T}_6\,(A', B', F, G,)$)

[4.7]**else if** $\neg\,(F\mathcal{U}G) \in \mathbf{C}$ and $\neg\,(F\mathcal{U}G) \in \mathbf{D}$ $\qquad (X \equiv (A' \vee \neg\,(F\mathcal{U}G))\mathcal{S}(B' \wedge \neg\,(F\mathcal{U}G)))$

return Sep($\mathcal{T}_7\,(A', B', F, G,)$)

[4.8]**else if** $\neg\,(F\mathcal{U}G) \in \mathbf{C}$ and $F\mathcal{U}G \in \mathbf{D}$ $\qquad (X \equiv (A' \vee \neg\,(F\mathcal{U}G))\mathcal{S}(B' \wedge F\mathcal{U}G))$

return Sep($\mathcal{T}_8\,(A', B', F, G,)$)

[5]**else if** $X = A\mathcal{S}B$ with A and B separated

let $\bigwedge_{i=1}^{m} A_i$ be a CNF of A

let $\bigvee_{j=1}^{n} B_j$ be a DNF of B

if $m = 0$ **then** set m to 1 and A_1 to $\top$

if $n = 0$ **then** set n to 1 and B_1 to $\bot$

return $\bigwedge_{i=1}^{m} \bigvee_{j=1}^{n}$ Sep($A_i\mathcal{S}B_j$)

[6]**else if** $X = A\mathcal{S}B$ **then return** Sep(Sep(A)$\mathcal{S}$Sep(B))

[7]**else if** $X = A\mathcal{U}B$ **then return** $\mathrm{Dual}($Sep$(\mathrm{Dual}(X)))$

In case 4 what we are doing is essentially picking one of the “hardest” cases and choosing to eliminate that one first.

3.1.1 Transformations

To make things easier to follow, we will be using diagrams such as the following:

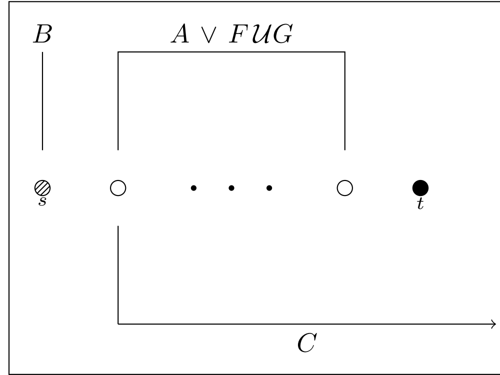

Diagram 3.2: Example diagram

Where ▨, ○ and ● represent points in the domain of the interpretation. With ● referring to a specific point, ▨ to the existence of such a point and ○ to a point that may or may not exist.

The order of the points in the diagram, from left to right, reflects the intended relationship between them in the interpretation's order. And if a point is next to another, it means there is no other point between them. To represent an arbitrary number of points, we use • • •.

Diagram 3.2 would mean that there is a point $s < t$, s satisfies B, (s, t) satisfies $A \vee F\mathcal{U}G$ and all points greater than s satisfy C.

Proposition 28. *Let $A, B \in \mathcal{L}_{TL}$, t a point and $t \Vdash A\mathcal{S}B$.*

Then there is a witness t_B for $A\mathcal{S}B$ at t such that $(t_B, t) \Vdash \neg B$, and we call such a t_B the nearest witness*.*

Proof. Let $W = \{s < t \mid s \Vdash B \text{ and } (s, t) \Vdash A\}$ be the set of witnesses.
W is non-empty because $t \Vdash A\mathcal{S}B$, and is bounded above by t.
So let t_B be the greatest element of W.

If any $s' \in (t_B, t)$ satisfied B, then s' would be a witness for $A\mathcal{S}B$ at t greater than t_B, but this contradicts the fact that t_B is the greatest element of W, so $(t_B, t) \Vdash \neg B$. □

Proposition 29. *For all $A, B \in \mathcal{L}_{TL}$: $A\mathcal{U}B \equiv \bigcirc(B \vee (A \wedge A\mathcal{U}B))$*

Proof. (⇒) Let $t \Vdash A\mathcal{U}B$ with t_B as witness.
If $t_B = \mathrm{Suc}(t)$ then $\mathrm{Suc}(t) \Vdash B$ which is equivalent to $t \Vdash \bigcirc B$.
Otherwise $\mathrm{Suc}(t) \in (t, t_B)$, and so $\mathrm{Suc}(t) \Vdash A$ and also $\mathrm{Suc}(t) \Vdash A\mathcal{U}B$ (with t_B as witness).

(⇐) If $\mathrm{Suc}(t) \Vdash B$ then $\mathrm{Suc}(t)$ is a witness for $t \Vdash A\mathcal{U}B$.
If $\mathrm{Suc}(t) \Vdash A$ and $\mathrm{Suc}(t) \Vdash A\mathcal{U}B$ with t_B as witness, then t_B is also a witness for $A\mathcal{U}B$ at t, as $(t, t_B) = [\mathrm{Suc}(t), t_B) \Vdash A$. □

Proposition 30. *Let* $\mathcal{T}_1(A, B, F, G) := P \wedge P'\mathcal{S}B$
where

$$N := (\neg G \wedge \neg B)\mathcal{S}(\neg A \wedge \neg B)$$

$$P' := N \rightarrow G \vee F$$

$$P := N \rightarrow G \vee (F \wedge F\mathcal{U}G)$$

Then $(A \vee F\mathcal{U}G)\mathcal{S}B \equiv \mathcal{T}_1(A, B, F, G)$

In this case, if t_0 satisfies $(A \vee F\mathcal{U}G)\mathcal{S}B$, then there is some $t_B < t_0$ with $t_B \Vdash B$ and $(t_B, t_0) \Vdash A \vee F\mathcal{U}G$. The difficulty here is finding a way to ensure all the points in (t_B, t_0) satisfy $F\mathcal{U}G$ if they don't happen to satisfy A, but without explicitly using $\mathcal{U}$.

For some point to satisfy $F\mathcal{U}G$, there needs to be an uninterrupted chain of Fs followed by a G. When a point $s \in (t_B, t_0)$ satisfies $N = (\neg G \wedge \neg B)\mathcal{S}(\neg A \wedge \neg B)$, this intuitively means that there is some point in (t_B, s) that did not satisfy A and so needs to satisfy $F\mathcal{U}G$, but there has been no witness so far since $\neg G$ has been true.

Our point s then needs to ensure that this happens by either being a witness itself or continuing the chain of Fs, this what $P' = N \rightarrow G \vee F$ means. Finally at t_0, if there is still no witness, t_0 itself must either be a witness or ensure that one exists in the future, this is done with $P = N \rightarrow G \vee (F \wedge F\mathcal{U}G)$.

Proof. (⇒) Let t_0 satisfy $(A \vee F\mathcal{U}G)\mathcal{S}B$ with a witness t_B. We show that $t_0 \Vdash P$ and $(t_B, t_0) \Vdash P'$, which implies that t_B is a witness for $P'\mathcal{S}B$ at t_0.

To do this, we first notice that $\vDash P \rightarrow P'$, and prove the stronger result $(t_B, t_0] \Vdash P$. So, for this purpose, let $s \in (t_B, t_0]$ with $s \Vdash N = (\neg G \wedge \neg B)\mathcal{S}(\neg A \wedge \neg B)$, and let $s_{\neg A}$ be a witness for this fact. $s_{\neg A}$ cannot be before t_B, as that would imply that $t_B \in (s_{\neg A}, s)$ and so would have to satisfy $\neg B$, which is a contradiction. And it cannot be equal to t_B, for the same reason. Hence $s_{\neg A}$ occurs between t_B and s. This means that $s_{\neg A}$ must satisfy $F\mathcal{U}G$ as it does not satisfy A. A diagram illustrating the situation:

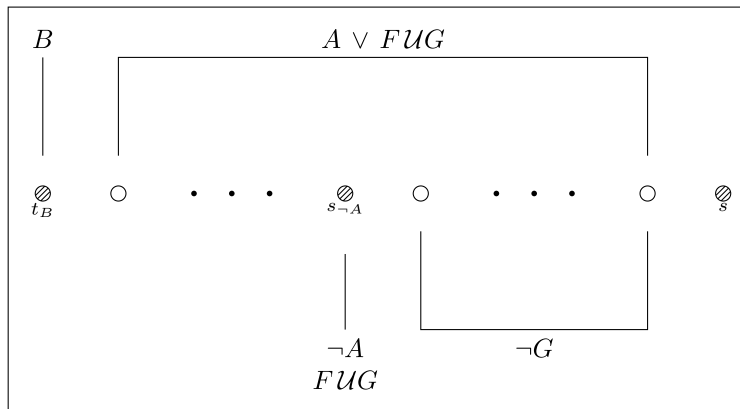

Diagram 3.3: $s \Vdash N$ with $s_{\neg A}$ as witness

Since $s_{\neg A}$ satisfies $F\mathcal{U}G$, there must be a witness $w > s_{\neg A}$ of this fact, and $w \geq s$ since no point between $s_{\neg A}$ and s satisfies G. If $w = s$ then $s \Vdash G$. If $w > s$ then s is between $s_{\neg A}$ and w and so satisfies F, and w is also a witness for $F\mathcal{U}G$ at s.

($\Leftarrow$) Let $t_0 \Vdash P \wedge P'\mathcal{S}B$ with t_B the nearest witness for $P'\mathcal{S}B$. If we show that $(t_B, t_0) \Vdash A \vee F\mathcal{U}G$ then t_B is also a witness for $(A \vee F\mathcal{U}G)\mathcal{S}B$ at t_0.

We now show that P is satisfied in the whole of $(t_B, t_0]$ by induction on the order $t_0 \to \mathrm{Pred}(t_0) \to \mathrm{Pred}^2(t_0) \to \cdots$. The base case is trivial, we already know $t_0 \Vdash P$. For the step, let $\mathrm{Pred}(s) \in (t_B, t_0)$ and assume $\mathrm{Pred}(s) \Vdash N = (\neg G \wedge \neg B)\mathcal{S}(\neg A \wedge \neg B)$ with r as witness. Remember that we have $s \Vdash P$ by the induction hypothesis. Diagram:

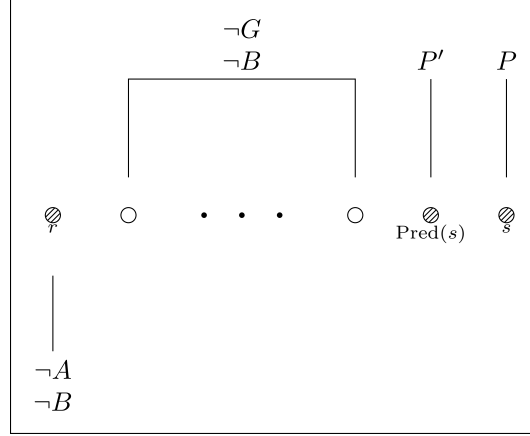

Diagram 3.4: $\mathrm{Pred}(s) \Vdash N$ with r as witness

If $\mathrm{Pred}(s) \Vdash G$ then also $\mathrm{Pred}(s) \Vdash G \vee (F \wedge F\mathcal{U}G)$ and we are done.

If $\mathrm{Pred}(s) \Vdash \neg G$, then we have $\mathrm{Pred}(s) \Vdash F$ via P', and remember that $\mathrm{Pred}(s) \in (t_B, t_0)$, which means that $\mathrm{Pred}(s) \Vdash \neg B$ since t_B is the nearest witness for $P'\mathcal{S}B$ at t_0. This gives us $\mathrm{Pred}(s) \Vdash \neg G \wedge \neg B$ and so r is also a witness for N at s, which gives us $s \Vdash G \vee (F \wedge F\mathcal{U}G)$ via P. Given this, by proposition 29, $\mathrm{Pred}(s) \Vdash F\mathcal{U}G$ as well.

We have thus showed that $(t_B, t_0] \Vdash P$.

Now consider $s_{\neg A} \in (t_B, t_0)$ such that $s_{\neg A} \Vdash \neg A$. Again because t_B is the nearest witness, $s_{\neg A} \Vdash \neg A \wedge \neg B$ and so $s_{\neg A}$ is a witness for N at $\mathrm{Suc}(s_{\neg A}) \in (t_B, t_0]$. Therefore $\mathrm{Suc}(s_{\neg A}) \Vdash G \vee (F \wedge F\mathcal{U}G)$ via P, and $s_{\neg A} \Vdash F\mathcal{U}G$ by proposition 29. $\square$

Proposition 31. *Let $\mathcal{T}_2(A, B, F, G) := H_< \vee H_\geq$*
where

$$H_< := A\mathcal{S}(G \wedge A \wedge (A \wedge F)\mathcal{S}B)$$

$$H_\geq := (A \wedge F)\mathcal{S}B \wedge (G \vee (F \wedge F\mathcal{U}G))$$

Then $A\mathcal{S}(B \wedge F\mathcal{U}G) \equiv \mathcal{T}_2(A, B, F, G)$

Proof. The satisfaction of $A\mathcal{S}(B \wedge F\mathcal{U}G)$ at t_0 is equivalent to the existence of a witness $t_B < t_0$ for $A\mathcal{S}(B \wedge F\mathcal{U}G)$ at t_0 and a witness $t_G > t_B$ for $F\mathcal{U}G$ at t_B.

Since we are working over linear orders, we can split this into a disjunction of three cases, depending on whether $t_G < t_0$, $t_G = t_0$ or $t_G > t_0$.

For the first case, we have the following diagram:

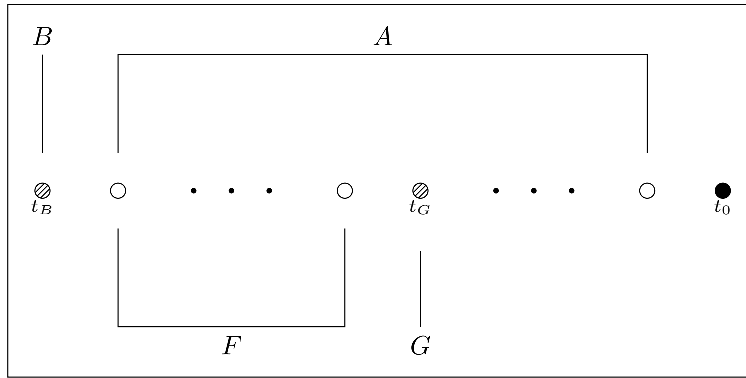

Diagram 3.5: $t_0 \Vdash A\mathcal{S}(B \wedge F\mathcal{U}G)$ with $t_G < t_0$

Adjusting this diagram slightly by "splitting" the A, we obtain:

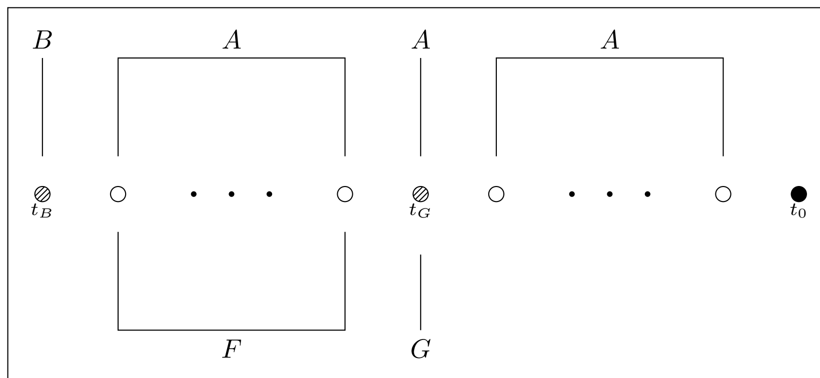

Diagram 3.6: $t_0 \Vdash A\mathcal{S}(B \wedge F\mathcal{U}G)$ with $t_G < t_0$, simplified

It is then clear that this diagram is equivalent to t_B being a witness for $(A \wedge F)\mathcal{S}B$ at t_G, and t_G a witness for $A\mathcal{S}(G \wedge A \wedge (A \wedge F)\mathcal{S}B) = H_<$ at t_0.

The second case ($t_G = t_0$) is equivalent to the diagram:

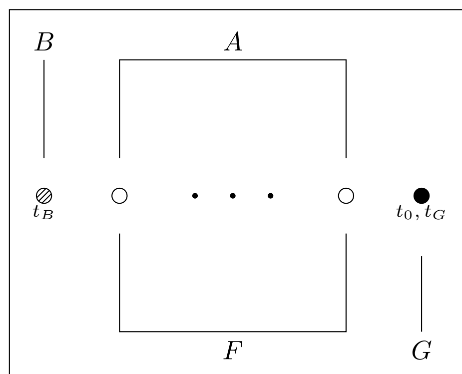

Diagram 3.7: $t_0 \Vdash A\mathcal{S}(B \wedge F\mathcal{U}G)$ with $t_G = t_0$

Which is equivalent to $(A \wedge F)\mathcal{S}B$ and G at t_0.

The third case ($t_G > t_0$):

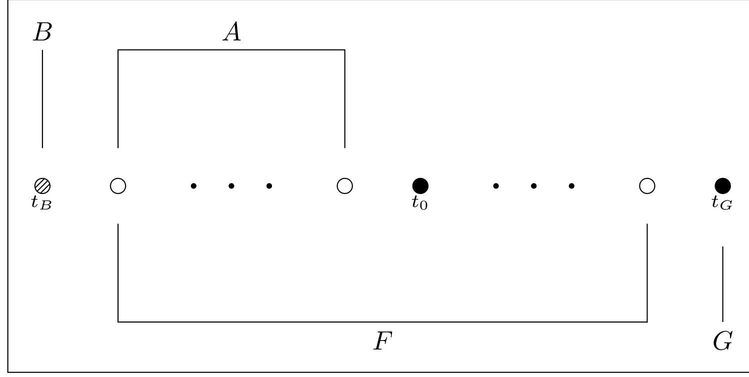

Diagram 3.8: $t_0 \Vdash A\mathcal{S}(B \wedge F\mathcal{U}G)$ with $t_G > t_0$

By splitting the F we get:

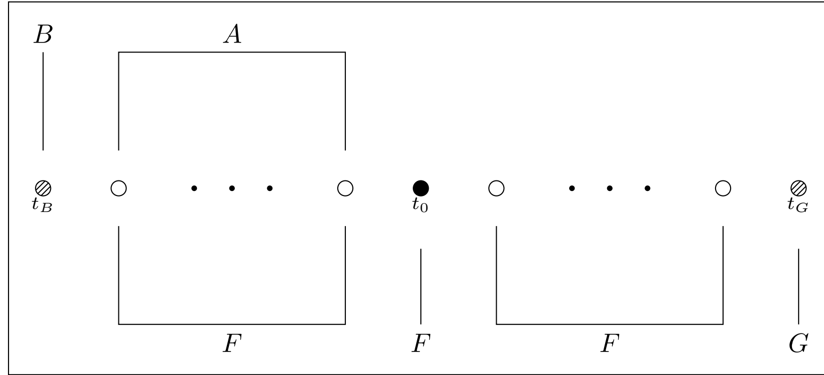

Diagram 3.9: $t_0 \Vdash A\mathcal{S}(B \wedge F\mathcal{U}G)$ with $t_G > t_0$, simplified

And so diagram 3.9 is equivalent to $(A \wedge F)\mathcal{S}B$, F and $F\mathcal{U}G$ at t_0.

Remember that the original formula is equivalent to a disjunction of the three cases, therefore we can combining the second and third cases by distribution to get $(A \wedge F)\mathcal{S}B \wedge (G \vee (F \wedge F\mathcal{U}G)) = H_{\geq}$. $\square$

Proposition 32. *Let* $\mathcal{T}_3(A, B, F, G) := H_< \vee H_{\geq}$
where

$$N := \neg G\mathcal{S}\neg A$$

$$P' := N \rightarrow G \vee F$$

$$P := N \rightarrow G \vee (F \wedge F\mathcal{U}G)$$

$$H_< := P \wedge P'\mathcal{S}(G \wedge F\mathcal{S}B)$$

$$H_{\geq} := F\mathcal{S}B \wedge (G \vee (F \wedge F\mathcal{U}G))$$

Then $(A \vee F\mathcal{U}G)\mathcal{S}(B \wedge F\mathcal{U}G) \equiv \mathcal{T}_3(A, B, F, G)$

Proof. Let t_0 be a point. Then t_0 satisfies $(A \vee F\mathcal{U}G)\mathcal{S}(B \wedge F\mathcal{U}G)$ if and only if there is a $t_B < t_0$ such that $t_B \Vdash B \wedge F\mathcal{U}G$ and $(t_B, t_0) \Vdash A \vee F\mathcal{U}G$. The satisfaction of $F\mathcal{U}G$ at t_B is equivalent to the

existence of a witness $t_G > t_B$. We can split this into two cases, $t_G \in (t_B, t_0)$ or $t_G \geq t_0$. We show that these cases are equivalent to $H_<$ and $H_\geq$, respectively.

The first case, where t_G occurs between t_B and t_0 is equivalent to the following diagram:

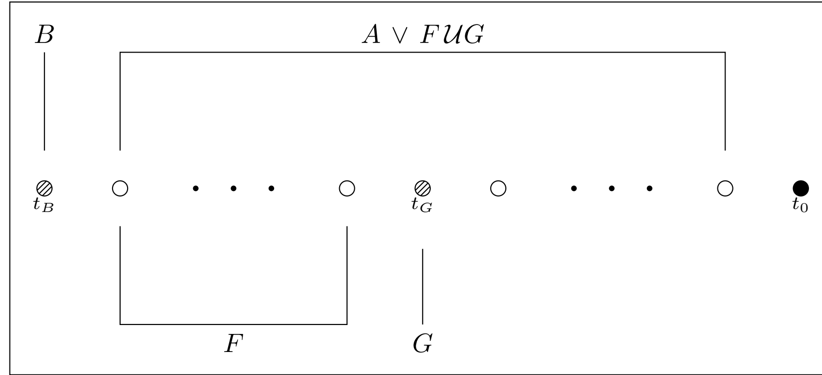

Diagram 3.10: $t_0 \Vdash (A \vee F\mathcal{U}G)\mathcal{S}(B \wedge F\mathcal{U}G)$ with $t_G < t_0$

Notice that in this case $F\mathcal{U}G$ is also satisfied in (t_B, t_G) with t_G as witness and so we can simplify the diagram to:

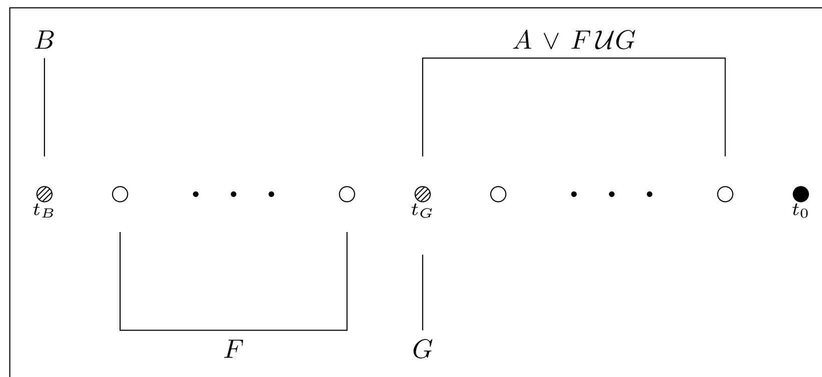

Diagram 3.11: $t_0 \Vdash (A \vee F\mathcal{U}G)\mathcal{S}(B \wedge F\mathcal{U}G)$ with $t_G < t_0$, simplified

We also have $t_G \Vdash G \wedge F\mathcal{S}B$, this last one with t_B as witness. Given that $\vDash P \rightarrow P'$, we will show that P is satisfied in $(t_G, t_0]$ to obtain the left to right implication, with t_G as a witness for $P'\mathcal{S}(G \wedge F\mathcal{S}B)$ at t_0.

Given a point $s \in (t_G, t_0]$, assume $s \Vdash N = \neg G\mathcal{S}\neg A$, we have to show that s satisfies either G or $F \wedge F\mathcal{U}G$. This means there is a witness $s_{\neg A} < s$, which cannot occur before t_G, as that would contradict the fact that $t_G \Vdash G$, and therefore occurs at t_G or between t_G and s, which means that $s_{\neg A} \Vdash F\mathcal{U}G$. In a (partial) diagram:

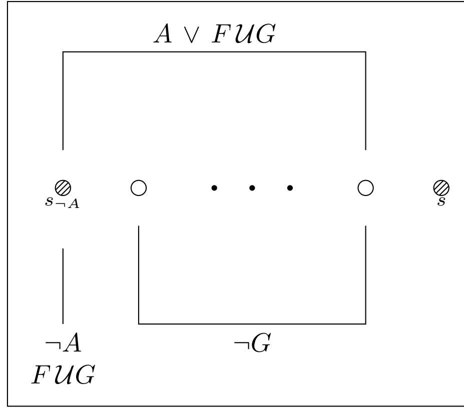

Diagram 3.12: Consequences of $s \Vdash N$

To satisfy $F\mathcal{U}G$ at $s_{\neg A}$ there must be a witness at s or after s, since no point between $s_{\neg A}$ and s can be a witness. If $s \Vdash G$ then we are done. If there is some witness after s, then s must satisfy F and the same witness provides $s \Vdash F\mathcal{U}G$.

Now, still in the first case, we worry about the other direction. Assume $t_0 \Vdash H_< = P \wedge P'\mathcal{S}(G \wedge F\mathcal{S}B)$, with t_G a witness for $P'\mathcal{S}(G \wedge F\mathcal{S}B)$ at t_0 and t_B a witness for $F\mathcal{S}B$ at t_G. Diagram:

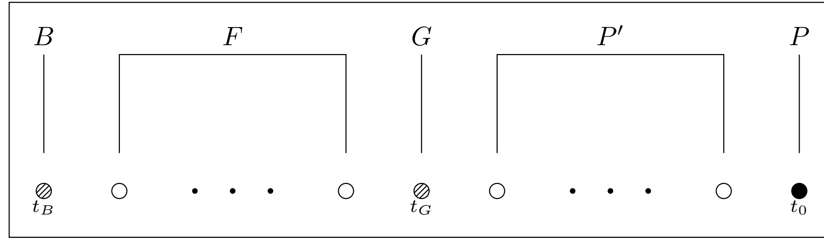

Diagram 3.13: $t_0 \Vdash H_<$

We will show that $[t_G, t_0) \Vdash A \vee F\mathcal{U}G$, which gives us the situation in diagram 3.11.

Similarly to the $(\Leftarrow)$ case of $\mathcal{T}_1$, we first show $(t_G, t_0] \Vdash P$ by induction on the order $t_0 \to \mathrm{Pred}(t_0) \to \mathrm{Pred}^2(t_0) \to \cdots$.

The base case is again trivial since we have $t_0 \Vdash P$. So let $\mathrm{Pred}(s) \in (t_G, t_0)$ and assume $\mathrm{Pred}(s) \Vdash N = \neg G\mathcal{S}\neg A$ with r as witness. Diagram:

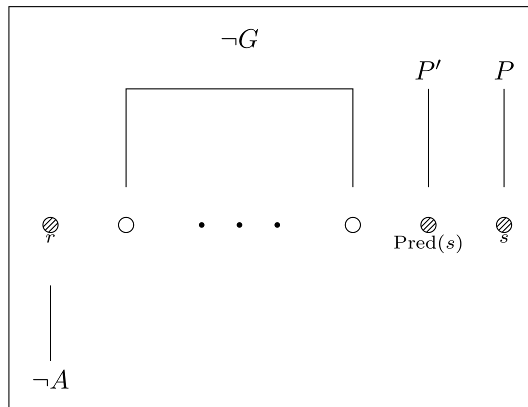

Diagram 3.14: $\mathrm{Pred}(s) \Vdash N$ with r as witness

If $\mathrm{Pred}(s) \Vdash G$ then $\mathrm{Pred}(s) \Vdash G \vee (F \wedge F\mathcal{U}G)$.

If $\mathrm{Pred}(s) \Vdash \neg G$ then $\mathrm{Pred}(s) \Vdash F$ via P', and r is also a witness for N at s which gives us $s \Vdash G \vee (F \wedge F\mathcal{U}G)$ via P and then $\mathrm{Pred}(s) \Vdash F\mathcal{U}G$ by proposition 29.

Given this, any $s_{\neg A} \in [t_G, t_0)$ that satisfies $\neg A$ is a witness for N at $\mathrm{Suc}(s_{\neg A}) \in (t_G, t_0]$, which gives us $\mathrm{Suc}(s_{\neg A}) \Vdash G \vee (F \wedge F\mathcal{U}G)$ via P and $s_{\neg A} \Vdash F\mathcal{U}G$ by proposition 29.

This concludes the first case.

For the second case, where $t_G \geq t_0$, we have the following diagrams for the cases where $t_G = t_0$ and $t_G > t_0$, respectively:

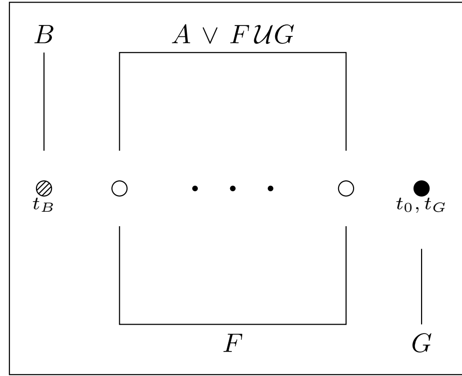

Diagram 3.15: $t_0 \Vdash (A \vee F\mathcal{U}G)\mathcal{S}(B \wedge F\mathcal{U}G)$ with $t_G = t_0$

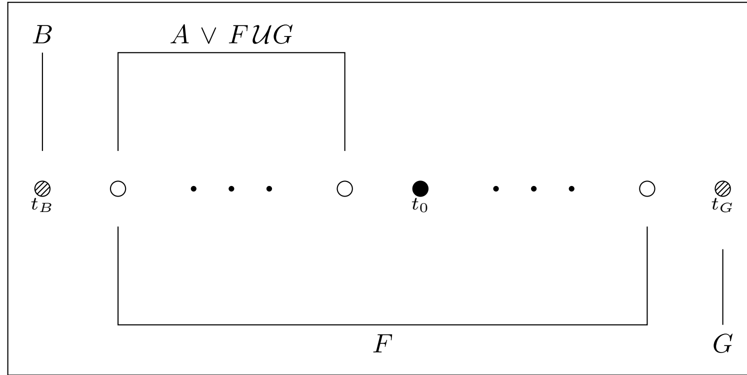

Diagram 3.16: $t_0 \Vdash (A \vee F\mathcal{U}G)\mathcal{S}(B \wedge F\mathcal{U}G)$ with $t_G > t_0$

In either case, t_G is a witness for $F\mathcal{U}G$ everywhere in (t_B, t_0), and so the diagrams without $A \vee F\mathcal{U}G$ in (t_B, t_0) are equivalent.

The first diagram is then equivalent to $t_0 \Vdash F\mathcal{S}B \wedge G$ and the second to $t_0 \Vdash F\mathcal{S}B \wedge F \wedge F\mathcal{U}G$, which combined by distribution give us $F\mathcal{S}B \wedge (G \vee (F \wedge F\mathcal{U}G)) = H_{\geq}$. ☐

Proposition 33. *For all $A \in \mathcal{L}_{TL}$, $\neg\blacksquare\neg A \equiv \blacklozenge A$.*

Proof. Recall that $\blacksquare A$ is defined as $\neg\blacklozenge\neg A$, then: $\neg\blacksquare\neg A = \neg\neg\blacklozenge\neg\neg A \equiv \blacklozenge A$. ☐

Proposition 34. *For all $A, B \in \mathcal{L}_{TL}$: $\neg(A\mathcal{S}B) \equiv \neg B\mathcal{S}(\neg A \wedge \neg B) \vee \blacksquare\neg B$*

Proof. (⇒) Let $t \Vdash \neg(A\mathcal{S}B)$ and consider the set $S = \{x < t \mid x \Vdash B\}$.

If S is empty then all points before t satisfy $\neg B$ and so $t \Vdash \blacksquare\neg B$.

Otherwise, let s be the greatest point in S. There must be a point in (s,t) that does not satisfy A, as otherwise we would have $t \Vdash A\mathcal{S}B$, call such point r. We are now in the situation described by the following diagram:

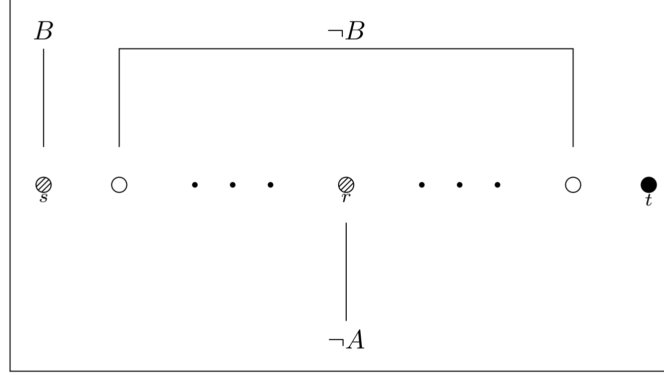

Diagram 3.17: s is the greatest point before t that satisfies B

Which means that r is a witness for $\neg B\mathcal{S}(\neg A \wedge \neg B)$ at t.

(⇐) If $t \Vdash \blacksquare\neg B$ then immediately $t \Vdash \neg(A\mathcal{S}B)$.

If $t \Vdash \neg B\mathcal{S}(\neg A \wedge \neg B)$ then let r be a witness. For every point s before t that satisfies B we must have $s < r$ since $[r,t) \Vdash \neg B$, but in that case there is at least one point in (s,t) that does not satisfy A, namely r. □

Corollary 35. *For all* $A, B \in \mathcal{L}_{TL}$*:* $\neg(A\mathcal{U}B) \equiv \neg B\mathcal{U}(\neg A \wedge \neg B) \vee \Box\neg B$

Proof. By definition of Dual:

$$\mathrm{Dual}(\neg(A\mathcal{U}B)) = \neg(\mathrm{Dual}(A)\mathcal{S}\,\mathrm{Dual}(B))$$

$$\mathrm{Dual}(\neg B\mathcal{U}(\neg A \wedge \neg B) \vee \Box\neg B) = \neg\,\mathrm{Dual}(B)\mathcal{S}(\neg\,\mathrm{Dual}(A) \wedge \neg\,\mathrm{Dual}(B)) \vee \blacksquare\neg\,\mathrm{Dual}(B)$$

By proposition 34, we have

$$\neg(\mathrm{Dual}(A)\mathcal{S}\,\mathrm{Dual}(B)) \equiv \neg\,\mathrm{Dual}(B)\mathcal{S}(\neg\,\mathrm{Dual}(A) \wedge \neg\,\mathrm{Dual}(B)) \vee \blacksquare\neg\,\mathrm{Dual}(B)$$

which means that the duals are also equivalent, by proposition 18. □

Proposition 36. *Let* $\mathcal{T}_4(A,B,F,G) := \neg\mathcal{T}_2(\neg B, \neg A \wedge \neg B, F, G) \vee \blacklozenge B$
Then $(A \vee \neg(F\mathcal{U}G))\mathcal{S}B \equiv \mathcal{T}_4(A,B,F,G)$

Proof. We have $(A \vee \neg(F\mathcal{U}G))\mathcal{S}B \equiv \neg\neg((A \vee \neg(F\mathcal{U}G))\mathcal{S}B)$. By using proposition 34 with the inner negation and the $\mathcal{S}$ we obtain

$$\neg\Big[\neg B\mathcal{S}(\neg(A \vee \neg(F\mathcal{U}G)) \wedge \neg B) \vee \blacksquare\neg B\Big]$$

Applying a Morgan law to $\neg(A \vee \neg(F\mathcal{U}G))$ and then again on the outer negation and disjunction:

$$\neg(\neg B\mathcal{S}(\neg A \wedge F\mathcal{U}G \wedge \neg B)) \wedge \neg\blacksquare\neg B$$

Rearranging the right side of the $\mathcal{S}$ and using proposition 33 on $\neg\blacksquare\neg B$:

$$\neg(\neg B\mathcal{S}(\neg A \wedge \neg B \wedge F\mathcal{U}G)) \wedge \blacklozenge B$$

The left disjunct is now in the form of $\mathcal{T}_2$ (proposition 31), which gives us the final result

$$\neg\mathcal{T}_2(\neg B, \neg A \wedge \neg B, F, G) \wedge \blacklozenge B$$

$\square$

Lemma 37. $A\mathcal{S}(B \wedge \square G) \equiv (A \wedge G)\mathcal{S}B \wedge G \wedge \square G$

Proof. The left side of the equivalence is equivalent to the existence of the diagram:

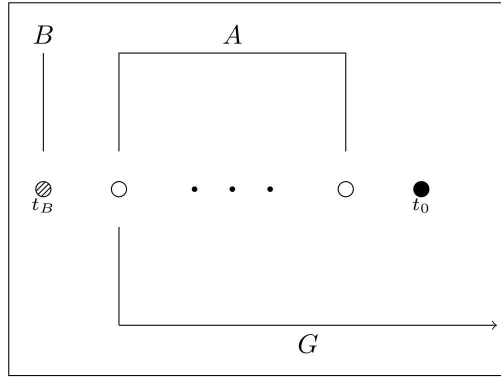

Diagram 3.18: $t_0 \Vdash A\mathcal{S}(B \wedge \square G)$ with t_B as witness

By splitting the G we see that it's also equivalent to the right side. $\square$

Proposition 38. *Let* $\mathcal{T}_5(A, B, F, G) := \quad A\mathcal{S}(\neg F \wedge \neg G \wedge A \wedge (A \wedge \neg G)\mathcal{S}B)$
$\vee \quad (A \wedge \neg G)\mathcal{S}B \wedge \neg G \wedge (\neg F \vee \neg(F\mathcal{U}G))$
Then $A\mathcal{S}(B \wedge \neg(F\mathcal{U}G)) \equiv \mathcal{T}_5(A, B, F, G)$

Proof. We start with $A\mathcal{S}(B \wedge \neg(F\mathcal{U}G))$. By corollary 35, we can split the $\neg(F\mathcal{U}G)$ to get

$$A\mathcal{S}(B \wedge (\neg G\mathcal{U}(\neg F \wedge \neg G) \vee \square\neg G))$$

Performing distribution with the newly introduced disjunction, this is equivalent to the disjunction of the following two formulas

$$A\mathcal{S}(B \wedge \neg G\mathcal{U}(\neg F \wedge \neg G))$$

$$A\mathcal{S}(B \wedge \square\neg G)$$

The first one is in the form of $\mathcal{T}_2$ (proposition 31), and we use lemma 37 on the second, obtaining

$$\mathcal{T}_2\,(A, B, \neg G, \neg F \wedge \neg G)$$

$$(A \wedge \neg G)\mathcal{S}B \wedge \neg G \wedge \Box\neg G$$

Expanding $\mathcal{T}_2$ in the first line:

$$A\mathcal{S}(\neg F \wedge \neg G \wedge A \wedge (A \wedge \neg G)\mathcal{S}B)$$

$$(A \wedge \neg G)\mathcal{S}B \wedge ((\neg F \wedge \neg G) \vee (\neg G \wedge \neg G\,\mathcal{U}\,(\neg F \wedge \neg G)))$$

$$(A \wedge \neg G)\mathcal{S}B \wedge \neg G \wedge \Box\neg G$$

Distribution on the second formula with $\neg G$:

$$A\mathcal{S}(\neg F \wedge \neg G \wedge A \wedge (A \wedge \neg G)\mathcal{S}B)$$

$$(A \wedge \neg G)\mathcal{S}B \wedge \neg G \wedge (\neg F \vee \neg G\,\mathcal{U}\,(\neg F \wedge \neg G))$$

$$(A \wedge \neg G)\mathcal{S}B \wedge \neg G \wedge \Box\neg G$$

Distribution again, the second and third formulas have a common prefix

$$A\mathcal{S}(\neg F \wedge \neg G \wedge A \wedge (A \wedge \neg G)\mathcal{S}B)$$

$$(A \wedge \neg G)\mathcal{S}B \wedge \neg G \wedge (\neg F \vee \neg G\,\mathcal{U}\,(\neg F \wedge \neg G) \vee \Box\neg G)$$

Finally, using corollary 35 again, we can get back the $\neg\,(F\mathcal{U}G)$

$$A\mathcal{S}(\neg F \wedge \neg G \wedge A \wedge (A \wedge \neg G)\mathcal{S}B)$$

$$(A \wedge \neg G)\mathcal{S}B \wedge \neg G \wedge (\neg F \vee \neg\,(F\mathcal{U}G))$$

□

Proposition 39. *Let* $\mathcal{T}_6\,(A, B, F, G) := \quad \mathcal{T}_1\,(A, (A \wedge \neg G)\mathcal{S}B \wedge \neg G \wedge \neg F \wedge A, F, G)$
$\quad\quad \vee \quad \mathcal{T}_3\,(A, (A \wedge \neg G)\mathcal{S}B \wedge \neg G \wedge \neg F, F, G)$
$\quad\quad \vee \quad (A \wedge \neg G)\mathcal{S}B \wedge \neg G \wedge (\neg F \vee \neg\,(F\mathcal{U}G))$
Then $(A \vee F\mathcal{U}G)\mathcal{S}(B \wedge \neg\,(F\mathcal{U}G)) \equiv \mathcal{T}_6\,(A, B, F, G)$

Proof. By corollary 35, $(A \vee F\mathcal{U}G)\mathcal{S}(B \wedge \neg\,(F\mathcal{U}G))$ is equivalent to

$$\Big(A \vee F\mathcal{U}G\Big)\mathcal{S}\Big(B \wedge (\neg G\,\mathcal{U}\,(\neg F \wedge \neg G) \vee \Box\neg G)\Big)$$

Using distribution, this can split into the disjunction of

$$(A \vee F\mathcal{U}G)\mathcal{S}(B \wedge \neg G\mathcal{U}(\neg F \wedge \neg G)) \quad (1)$$

$$(A \vee F\mathcal{U}G)\mathcal{S}(B \wedge \Box\neg G) \quad (2)$$

Looking at (1) first, we can split this into a disjunction three cases depending on where the witness for $\neg G\mathcal{U}(\neg F \wedge \neg G)$ occurs, before t_0, at t_0, or after t_0. Call this witness t_T.

The first case leads to the diagram

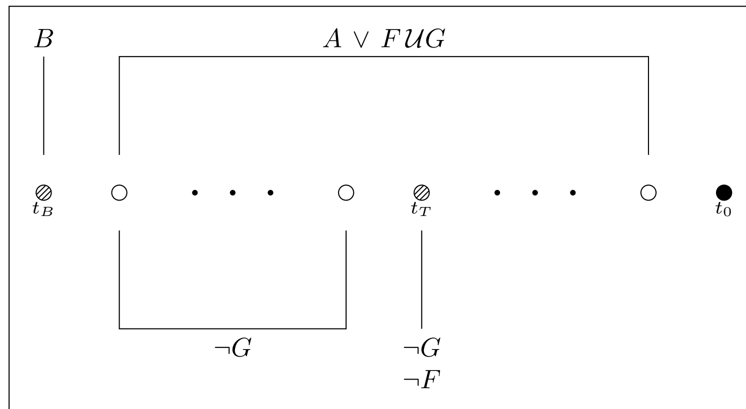

Diagram 3.19: t_0 satisfies (1) with $t_T < t_0$

Notice that $\neg G\mathcal{U}(\neg F \wedge \neg G)$ is true in (t_B, t_T), which implies $\neg(F\mathcal{U}G)$ by corollary 35, and so this is equivalent to A being true in (t_B, t_T). Updated diagram:

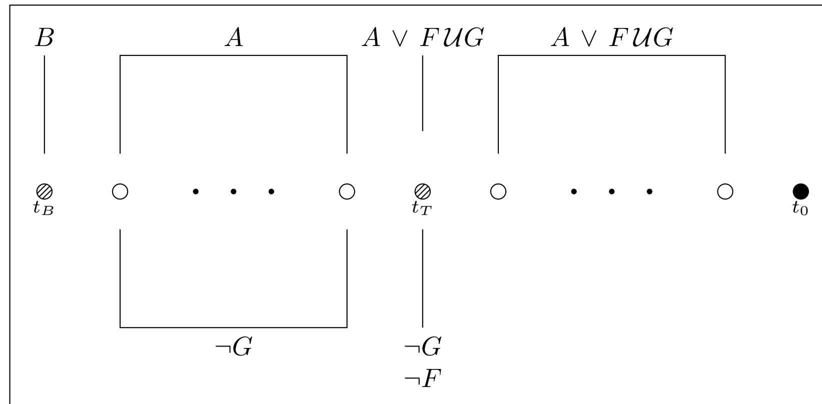

Diagram 3.20: t_0 satisfies (1) with $t_T < t_0$, updated

This last diagram is then equivalent to

$$(A \vee F\mathcal{U}G)\mathcal{S}((A \wedge \neg G)\mathcal{S}B \wedge \neg G \wedge \neg F \wedge (A \vee F\mathcal{U}G)) \quad (1.<)$$

Again, using distribution on $(A \vee F\mathcal{U}G)$, in the right side of the outer $\mathcal{S}$, we get a disjunction of

$$(A \vee F\mathcal{U}G)\mathcal{S}((A \wedge \neg G)\mathcal{S}B \wedge \neg G \wedge \neg F \wedge A) \quad (1.<.1)$$

$$(A \lor F\mathcal{U}G)\mathcal{S}((A \land \neg G)\mathcal{S}B \land \neg G \land \neg F \land F\mathcal{U}G) \tag{1.<.2}$$

By proposition 30, (1.<.1) is equivalent to $\mathcal{T}_1(A, (A \land \neg G)\mathcal{S}B \land \neg G \land \neg F \land A, F, G)$. And by proposition 32, (1.<.2) is equivalent to $\mathcal{T}_3(A, (A \land \neg G)\mathcal{S}B \land \neg G \land \neg F, F, G)$.

We have now obtained the first and second disjuncts of the definition of $\mathcal{T}_6$, we will see that the disjunction of the two remaining cases and (2) is equivalent to the third disjunct.

Continuing with the other cases of (1). The $t_T = t_0$ case leads to the following diagram:

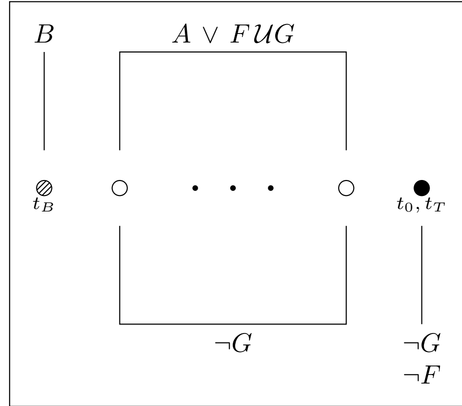

Diagram 3.21: t_0 satisfies (1) with $t_T = t_0$

For the same reason as previously, this is equivalent to having the stronger A instead of $A \lor F\mathcal{U}G$ in (t_B, t_0), and the diagram is equivalent to the formula

$$(A \land \neg G)\mathcal{S}B \land \neg G \land \neg F \tag{1.=}$$

We continue by looking at the final case of (1), where $t_T > t_0$, before coming back to (1.=). Diagram:

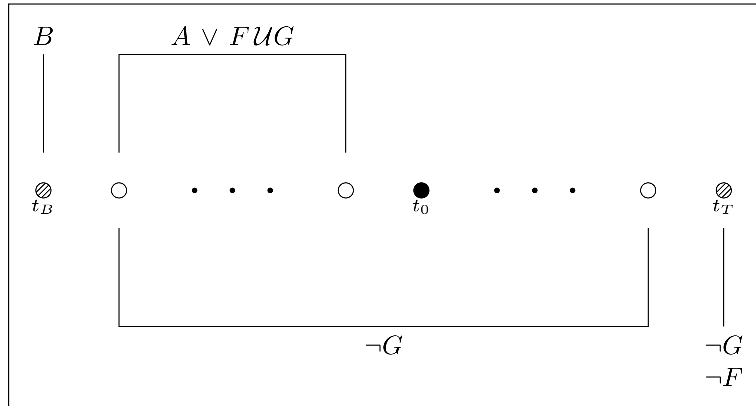

Diagram 3.22: t_0 satisfies (1) with $t_T > t_0$

Similarly, $A \lor F\mathcal{U}G$ can be replaced with A and we get the formula

$$(A \land \neg G)\mathcal{S}B \land \neg G \land \neg G\mathcal{U}(\neg F \land \neg G) \tag{1.>}$$

We still have the formulas (1.=) and (1.>) to deal with, but we explore (2) before we do so.

A diagram equivalent to (2) is as follows:

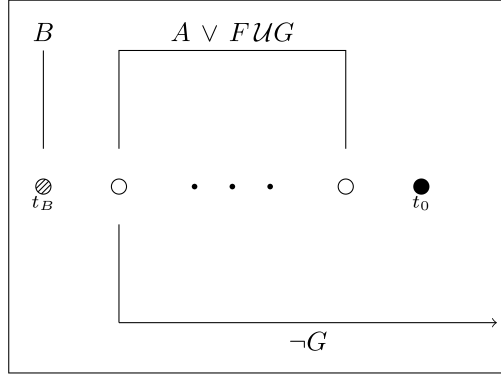

Diagram 3.23: t_0 satisfies (2)

Again by corollary 35, $\neg(F\mathcal{U}G)$ is satisfied in (t_B, t_0) and so $A \vee F\mathcal{U}G$ in (t_B, t_0) is equivalent to A in (t_B, t_0). Which means (2) is equivalent to $A\mathcal{S}(B \wedge \Box\neg G)$. Using lemma 37, we get

$$(A \wedge \neg G)\mathcal{S}B \wedge \neg G \wedge \Box\neg G \tag{2'}$$

We now combine (1.$>$) and (2$'$) by distribution (notice the common prefix) to get

$$(A \wedge \neg G)\mathcal{S}B \wedge \neg G \wedge (\neg G\mathcal{U}(\neg F \wedge \neg G) \vee \Box\neg G) \tag{3}$$

which, by corollary 35, is equivalent to

$$(A \wedge \neg G)\mathcal{S}B \wedge \neg G \wedge \neg(F\mathcal{U}G) \tag{3'}$$

Combining (1.$=$) and (3$'$) by distribution we obtain the third disjunct. ☐

Proposition 40. *Let* $\mathcal{T}_7(A, B, F, G) := \neg(\mathcal{T}_3(\neg B, \neg A, F, G)) \wedge \mathcal{T}_5(\top, B, F, G)$
Then $(A \vee \neg(F\mathcal{U}G))\mathcal{S}(B \wedge \neg(F\mathcal{U}G)) \equiv \mathcal{T}_7(A, B, F, G)$

Proof. For clarity, let $X = A \vee \neg(F\mathcal{U}G)$ and $Y = B \wedge \neg(F\mathcal{U}G)$ and perform these substituions in the original formula.

This first part is quite similar to the one of proposition 36. We have $X\mathcal{S}Y \equiv \neg\neg(X\mathcal{S}Y)$. Using proposition 34 we get

$$\neg\Big[\neg Y\mathcal{S}(\neg X \wedge \neg Y) \vee \blacksquare\neg Y\Big]$$

Rewriting using a Morgan law with the outer negation and disjunction, and then using proposition 33 on the right conjunct:

$$\neg(\neg Y\mathcal{S}(\neg X \wedge \neg Y)) \wedge \blacklozenge Y$$

Consider $\blacklozenge Y$ first. By substituting back Y we get

$$\blacklozenge(B \wedge \neg(F\mathcal{U}G)) = \top\mathcal{S}(B \wedge \neg(F\mathcal{U}G)) \equiv \mathcal{T}_5(\top, D, B, C)$$

Now consider $\neg Y\mathcal{S}(\neg X \wedge \neg Y)$, by substituting back X and Y we get

$$\neg (B \wedge \neg (F\mathcal{U}G))\,\mathcal{S}(\neg (A \vee \neg (F\mathcal{U}G)) \wedge \neg (B \wedge \neg (F\mathcal{U}G)))$$

Using Morgan laws on $\neg (A \vee \neg (F\mathcal{U}G))$ and $\neg (B \wedge \neg (F\mathcal{U}G))$:

$$(\neg B \vee F\mathcal{U}G)\mathcal{S}(\neg A \wedge F\mathcal{U}G \wedge (\neg B \vee F\mathcal{U}G))$$

And then using absorption on the right side of the $\mathcal{S}$:

$$(\neg B \vee F\mathcal{U}G)\mathcal{S}(\neg A \wedge F\mathcal{U}G) \equiv \mathcal{T}_3\,(\neg B, \neg A, B, C)$$

The original formula is then equivalent to $\neg\,(\mathcal{T}_3\,(\neg B, \neg A, B, C)) \wedge \mathcal{T}_5\,(\top, D, B, C)$ ☐

Proposition 41. *Let* $\mathcal{T}_8\,(A, B, F, G) := \mathcal{T}_4\,(A, (A \wedge F)\mathcal{S}B \wedge G \wedge A, F, G)$
$\vee\ \mathcal{T}_7\,(A, (A \wedge F)\mathcal{S}B \wedge G, F, G)$
$\vee\ (A \wedge F)\mathcal{S}B \wedge (G \vee (F \wedge F\mathcal{U}G))$
Then $(A \vee \neg (F\mathcal{U}G))\mathcal{S}(B \wedge F\mathcal{U}G) \equiv \mathcal{T}_8\,(A, B, F, G)$

Proof. This proof follows fairly similarly to the proof of proposition 39.

We begin by considering the three cases depending on where the witness for $F\mathcal{U}G$ in the right side of $\mathcal{S}$ (t_G) occurs, which give rise to the following three diagrams:

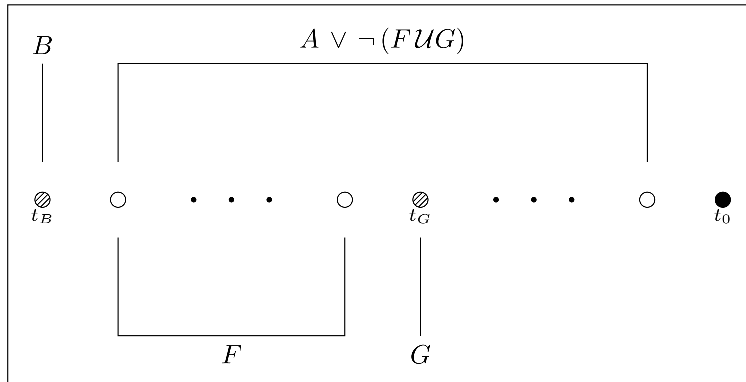

Diagram 3.24: $t_0 \Vdash (A \vee \neg (F\mathcal{U}G))\mathcal{S}(B \wedge F\mathcal{U}G)$ with $t_G < t_0$

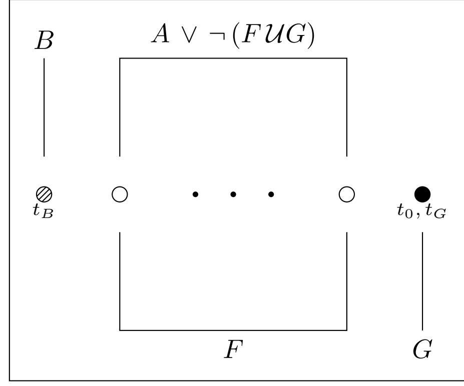

Diagram 3.25: $t_0 \Vdash (A \lor \neg(F\mathcal{U}G))\mathcal{S}(B \land F\mathcal{U}G)$ with $t_G = t_0$

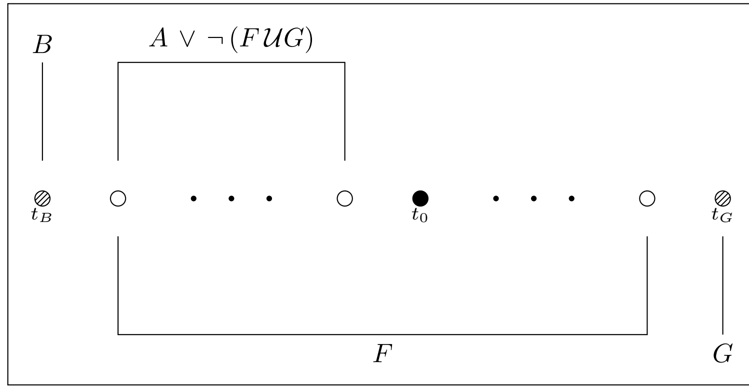

Diagram 3.26: $t_0 \Vdash (A \lor \neg(F\mathcal{U}G))\mathcal{S}(B \land F\mathcal{U}G)$ with $t_G > t_0$

In all three diagrams, t_G is a witness for $F\mathcal{U}G$ in the region (t_B, t_G), which means that $(t_B, t_G) \Vdash A \lor \neg(F\mathcal{U}G)$ is equivalent to $(t_B, t_G) \Vdash A$. This then gives us the formulas (respectively):

$$(A \lor \neg(F\mathcal{U}G))\mathcal{S}((A \land F)\mathcal{S}B \land G \land (A \lor \neg(F\mathcal{U}G))) \tag{1}$$

$$(A \land F)\mathcal{S}B \land G \tag{2}$$

$$(A \land F)\mathcal{S}B \land F \land F\mathcal{U}G \tag{3}$$

We can further split (1) by the $(A \lor \neg(F\mathcal{U}G))$ on right side into

$$(A \lor \neg(F\mathcal{U}G))\mathcal{S}((A \land F)\mathcal{S}B \land G \land A) \tag{1.1}$$

$$(A \lor \neg(F\mathcal{U}G))\mathcal{S}((A \land F)\mathcal{S}B \land G \land \neg(F\mathcal{U}G)) \tag{1.2}$$

(1.1) and (1.2) are equivalent to $\mathcal{T}_4\,(A, (A \land F)\mathcal{S}B \land G \land A, F, G)$ and $\mathcal{T}_7\,(A, (A \land F)\mathcal{S}B \land G, F, G)$, by propositions 36 and 40, respectively.

(2) and (3) combine into the third disjunct of the definition of $\mathcal{T}_8$. □

The algorithm is now completely defined, and the rest the chapter is devoted to showing that it is correct.

3.2 Algorithm Correctness

We have now seen that the transformations produce equivalent formulas. But we still have to show that the algorithm constructs an equivalent separated formula. In particular we have to show that the algorithm does in fact terminate. We do so by induction on a well-founded partial order on formulas.

This order is obtained by combining several relations, each necessary for some cases of the algorithm.

We start by defining some functions on paths that somehow measure how “unseparated” they are.

Definition 42. Given a path π, the *degree* of π, $\mathcal{D}(\pi)$, is the number of adjacent pairs in π with both $\mathcal{S}$ and $\mathcal{U}$. Or, perhaps more intuitively, it is the number of transitions from $\mathcal{S}$ to $\mathcal{U}$ and vice-versa.

Definition 43. Let π be a path. We define $\mathcal{L}_1(\pi)$ to be the length of the last *homogeneous segment*, except in the case where the segment is the whole path, in which case we take $\mathcal{L}_1(\pi)$ to be zero. That is:

$$\mathcal{L}_1(\pi) := \begin{cases} n+1 & \text{if } \pi = \lambda\mathcal{S}\mathcal{U}^{n+1} \text{ or } \pi = \lambda\mathcal{U}\mathcal{S}^{n+1} \\ 0 & \text{otherwise} \end{cases}$$

For convenience, we combine these into one.

Definition 44. Let π be a path. Then:

$$\mathrm{Score}_1(\pi) := \langle \mathcal{D}(\pi), \mathcal{L}_1(\pi) \rangle$$

So far these have been over paths, we now extend the definitions to formulas.

Whenever we have a Cartesian product of orders, the intended order on it is the usual lexicographic order. That is, if $\langle x, y\rangle, \langle x', y'\rangle \in X \times Y$ with $<_X$ an order over X and $<_Y$ an order over Y, $\langle x, y\rangle < \langle x', y'\rangle$ if and only if $x <_X x'$ or ($x = x'$ and $y <_Y y'$). If all such orders are well-founded then so is the lexicographic order, and if they are total then so is the lexicographic order.

Definition 45. Let $A \in \mathcal{L}_{\mathsf{TL}}$. Then:

$$\mathrm{Score}_1(A) := \begin{cases} \langle 0, 0\rangle & \text{if } \mathrm{Paths}(A) = \emptyset \\ \max \mathrm{Score}_1[\mathrm{Paths}(A)] & \text{otherwise} \end{cases}$$

To obtain something that is reduced by every transformation, we need an additional thing, which we call leader count. And whose definition requires the idea of contexted formula.

Definition 46. A *contexted formula* is a pair $\langle \pi, A\rangle$ where π is a path and A is a formula.

Definition 47. Let A be a formula. Then $\mathcal{C}(A)$ is the set of subformulas of A together with their context (the path corresponding to the syntax path that leads to their subtree). More precisely:

$$\mathcal{C}(\bot) := \{\langle \langle\rangle, \bot\rangle\}$$

$$\mathcal{C}(\top) := \{\langle\langle\rangle, \top\rangle\}$$

$$\mathcal{C}(p) := \{\langle\langle\rangle, p\rangle\}$$

$$\mathcal{C}(\neg A) := \{\langle\langle\rangle, \neg A\rangle\} \cup \mathcal{C}(A)$$

$$\mathcal{C}(A \vee B) := \{\langle\langle\rangle, A \vee B\rangle\} \cup \mathcal{C}(A) \cup \mathcal{C}(B)$$

$$\mathcal{C}(A \wedge B) := \{\langle\langle\rangle, A \wedge B\rangle\} \cup \mathcal{C}(A) \cup \mathcal{C}(B)$$

$$\mathcal{C}(A\mathcal{S}B) := \{\langle\langle\rangle, A\mathcal{S}B\rangle\} \cup \{\langle\mathcal{S}\pi, C\rangle \mid \langle\pi, C\rangle \in \mathcal{C}(A) \cup \mathcal{C}(B)\}$$

$$\mathcal{C}(A\mathcal{U}B) := \{\langle\langle\rangle, A\mathcal{U}B\rangle\} \cup \{\langle\mathcal{U}\pi, C\rangle \mid \langle\pi, C\rangle \in \mathcal{C}(A) \cup \mathcal{C}(B)\}$$

As an example of a contexted subformula, consider the following formula

$$A = (q\mathcal{U}(\top\mathcal{U}p) \vee p\mathcal{U}q)\mathcal{S}(p\mathcal{S}(r\mathcal{U}(\top\mathcal{U}p)))$$

The formula A has the following syntax tree:

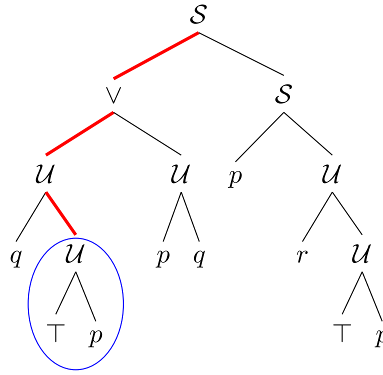

Diagram 3.27: Syntax tree of A

The circled subformula is $\top\mathcal{U}p$. The syntax path that goes from the root of A to the root of this subformula is highlighted in red, and is associated with the path $\mathcal{SU}$. This then represents the contexted subformula $\langle\mathcal{SU}, \top\mathcal{U}p\rangle$ of A.

For the purposes of our algorithm, it suffices to look at the subset of these contexted subformulas where the subformula is simple and pure, which leads to the following definition.

Definition 48. Let $A \in \mathcal{L}_{\mathsf{TL}}$.

$$\mathcal{SPC}(A) := \{\langle\pi, B\rangle \in \mathcal{C}(A) \mid B \text{ is simple and pure}\}$$

We now extend the definition of Paths and Score_1 to contexted formulas in a natural way.

Definition 49. Let $\langle\pi, A\rangle$ be a contexted formula. Then:

$$\mathrm{Paths}(\langle\pi, A\rangle) := \{\pi\lambda \mid \lambda \in \mathrm{Paths}(A)\}$$

$$\text{Score}_1(\langle \pi, A\rangle) := \begin{cases} \langle 0, 0\rangle & \text{if } \text{Paths}(\langle \pi, A\rangle) = \emptyset \\ \max \text{Score}_1[\text{Paths}(\langle \pi, A\rangle)] & \text{otherwise} \end{cases}$$

With this, we can define the concepts of leader and leader count.

Definition 50. Let $A \in \mathcal{L}_{\mathsf{TL}}$ and $\langle \pi, B\rangle \in \mathcal{SPC}(A)$.

Then we say $\langle \pi, B\rangle$ is a *leader* of A if it has maximal Score_1 in $\mathcal{SPC}(A)$ and call $\text{Leaders}(A)$ the set of leaders of A.

Definition 51. Given a formula A, $\mathcal{N}(A)$ is the number of leaders of A with distinct formulas, that is:

$$\mathcal{N}(A) := \#\{B \mid \langle \pi, B\rangle \in \text{Leaders}(A)\}$$

Coming back to the previous example, here we have the syntax tree of A with the pure simple formulas circled:

Diagram 3.28: Syntax tree of A with $\mathcal{SPC}$ circled

Now consider only the ones with maximal Score_1, these are the leaders of A, circled in following tree diagram:

Diagram 3.29: Syntax tree of A with leaders circled

We have $\mathcal{N}(A) = 4$ and not 6, as the subformulas $\top \mathcal{U} p$ and p are repeated in the leaders, even though they appear with different contexts.

So far we have defined Score_1 and $\mathcal{N}$ on formulas, and indeed these together are enough to obtain something that is reduced by every transformation when used in the conditions of the algorithm. The idea is that when a transformation "pulls out" one of the $F\mathcal{U}G$ from inside the $\mathcal{S}$, it either reduces $\mathcal{N}$ or it pulled out the last leader and Score_1 decreases.

We now define another pair of functions, quite similar to $\mathcal{L}_1$ and Score_1, which are necessary for case 6 of the algorithm.

Definition 52. Let π be a path. Then $\mathcal{L}_0(\pi)$ is the length of the first homogeneous segment, with the exception of a fully homogeneous path. Score_0 is analogous to Score_1.

$$\mathcal{L}_0(\pi) := \begin{cases} n+1 & \text{if } \pi = \mathcal{S}^{n+1}\mathcal{U}\lambda \text{ or } \pi = \mathcal{U}^{n+1}\mathcal{S}\lambda \\ 0 & \text{otherwise} \end{cases}$$

$$\mathrm{Score}_0(\pi) := \langle \mathcal{D}(\pi)\,, \mathcal{L}_0(\pi) \rangle$$

This leads to a definition of Score_0 on formulas analogous to the definition of Score_1. And for convenience, we bundle Score_1, $\mathcal{N}$ and Score_0 into a single function, called *weight*.

Definition 53. Let A be a formula, then:

$$\mathrm{Score}_0(A) := \begin{cases} \langle 0, 0 \rangle & \text{if } \mathrm{Paths}(A) = \emptyset \\ \max \mathrm{Score}_0[\mathrm{Paths}(A)] & \text{otherwise} \end{cases}$$

$$\mathcal{W}(A) := \langle \mathrm{Score}_1(A), \mathcal{N}(A)\,, \mathrm{Score}_0(A) \rangle$$

The binary relation $<_{\mathcal{W}}$ is defined by

$$A <_{\mathcal{W}} B \iff \mathcal{W}(A) < \mathcal{W}(B)$$

and is well-founded by the result below.

Proposition 54. *Let R be a well-founded binary relation on A, S a binary relation on X and $f : X \to A$ a function such that*

$$xSy \implies f(x)Rf(y)$$

Then S is well-founded.

Proof. Let $Y \subseteq X$ be non-empty and let $f(m) \in f[Y] \subseteq A$ be R-minimal in $f[Y]$. Then for any $x \in Y$, xSm implies $f(x)Rf(m)$, which is a contradiction, hence m is S-minimal in Y. □

We still need a few more relations.

Definition 55. Let A and B be formulas, then $A \prec B$ if and only if A is an immediate proper subformula of B.

The relation defined above is of course used in cases 1,2 and 3.

In the rest of this text, if R is a binary relation, we write R^+ to mean the transitive closure of R.

This next relation is used in case 5.

Definition 56. $\lessdot$ is a binary relation on formulas such that $A \lessdot B$ if and only if one of:

- $A = C'\mathcal{S}D$, $B = C\mathcal{S}D$ and $C' \prec^+ C$
- $A = C\mathcal{S}D'$, $B = C\mathcal{S}D$ and $D' \prec^+ D$

The next couple of relations are used to deal with with the normal form conversions.

Definition 57. Let $A, B \in \mathcal{L}_{\mathsf{TL}}$. Then we define the relations $\lhd_l$ and $\lhd_r$:

$$A \lhd_l B \text{ if only if } A = C'\mathcal{S}D,\ B = C\mathcal{S}D,\ C \text{ is not in CNF and } C' \text{ is a CNF of } C$$

$$A \lhd_r B \text{ if only if } A = C\mathcal{S}D',\ B = C\mathcal{S}D,\ D \text{ is not in DNF and } D' \text{ is a DNF of } D$$

And finally something for case 7.

Definition 58. $\lhd_u$ is a binary relation on formulas defined as:

$$A \lhd_u B \text{ if only if } A = C\mathcal{S}D \text{ and } B = \mathrm{Dual}(A)$$

We are now ready to define the final relation that is sufficient to show the correctness of the algorithm, by combining the previously defined relations.

Definition 59. The binary relation $< \subseteq \mathcal{L}_{\mathsf{TL}} \times \mathcal{L}_{\mathsf{TL}}$ is defined as $< := \ (<_{\mathcal{W}} \cup \prec \cup \lessdot \cup \lhd_l \cup \lhd_r \cup \lhd_u)^+$.

The goal is to show that this relation is a well-founded partial order. To do this we first show that $\prec \cup \lessdot \cup \lhd_l \cup \lhd_r \cup \lhd_u$ is well-founded and that each of these relations is compatible with $<_{\mathcal{W}}$ in some suitable sense.

To do the former we define some functions on formulas into a well-founded relation, show that each of these relations necessarily decreases this quantity, which means that so does their union. Proposition 54 then provides the result.

So, for this purpose, let $d, v : \mathcal{L}_{\mathsf{TL}} \to \omega,\ \ c : \mathcal{L}_{\mathsf{TL}} \to \omega + 1,\ \ l, r, u : \mathcal{L}_{\mathsf{TL}} \to 2$ be functions with the following definitions:

$$d(A) \text{ is the temporal depth of } A$$

$$c(A) := \begin{cases} d(B) + d(C) & \text{if } A = B\mathcal{S}C \\ \omega & \text{otherwise} \end{cases}$$

$$l(A) := \begin{cases} 0 & \text{if } A = B\mathcal{S}C \text{ with } B \text{ in CNF} \\ 1 & \text{otherwise} \end{cases}$$

$$r(A) := \begin{cases} 0 & \text{if } A = B\mathcal{S}C \text{ with } C \text{ in DNF} \\ 1 & \text{otherwise} \end{cases}$$

$$u(A) := \begin{cases} 0 & \text{if } A = B\mathcal{S}C \\ 1 & \text{otherwise} \end{cases}$$

$v(A)$ is the number of vertices in the syntax tree of A

Bringing these all together, we define a function f as

$$f(A) := \langle d(A), c(A), l(A), r(A), u(A), v(A) \rangle$$

The codomain of this f is what we use to show that $\prec \cup \lessdot \cup \lhd_l \cup \lhd_r \cup \lhd_u$ is well-founded.

Proposition 60. $A \prec B \implies f(A) < f(B)$

Proof. Assume $A \prec B$.

We cannot have $B = \bot$, $B = \top$ or $B = p$ since these have no proper subformulas.

If B is an $\mathcal{U}$ or $\mathcal{S}$, then $d(A) < d(B)$.

If B is one of the Boolean operators, $d(A)$ is not bigger than $d(B)$ since $\mathrm{Paths}(A) \subseteq \mathrm{Paths}(B)$. And $c(B) = \omega$, $l(B) = r(B) = u(B) = 1$, all of these are the greatest element in their respective orders, so they cannot be bigger for A. But the syntax tree of A is a proper subtree of the one of B, so $v(A) < v(B)$. □

Proposition 61. $A \lessdot B \implies f(A) < f(B)$

Proof. By definition of $\lessdot$, both A and B must have $\mathcal{S}$ as their outer connective. We show only the case where the left side of the $\mathcal{S}$ is different. The right-side case is very similar, with the appropriate substitutions (e.g. DNF instead of CNF and r instead of l). So let $A = C'\mathcal{S}D$ and $B = C\mathcal{S}D$.

First of all, $d(C') \leq d(C)$ as C' is a subformula of C, and so $d(A) = 1 + \max\{d(C'), d(D)\} \leq 1 + \max\{d(C), d(D)\} = d(B)$.

Now we separate into two cases, depending on whether $l(A) > l(B)$ or $l(A) \leq l(B)$.

If $l(A) > l(B)$, then $l(A) = 1$ and $l(B) = 0$, which means C is in CNF but C' is not. Notice that any subformula of a formula in CNF that does not "go into" a temporal operator is also in CNF. C' has lost the normal form, and so we can deduce that at least one level of temporal depth has been lost. Hence $d(C') < d(C)$. So $c(A) = d(C') + d(D) < d(C) + d(D) = c(B)$.

If $l(A) \leq l(B)$, we have $c(A) = d(C') + d(D) \leq d(C) + d(D) = c(B)$ since $d(C') \leq d(C)$, we also have $l(A) \leq l(B)$ by assumption, $r(A) = r(B)$ as they have the same right side, and $u(A) = u(B) = 1$ as neither A nor B are an $\mathcal{U}$. But $v(A) < v(B)$, as the syntax tree of C' is a proper subtree of the one of C. □

Before we go into the proofs for $\lhd_l$ and $\lhd_r$, we are going to present some helpful results.

Proposition 62. *For all formulas A and B:*

- *If A is a CNF of B then* $\mathrm{Paths}(A) \subseteq \mathrm{Paths}(B)$ *and* $\mathcal{SPC}(A) \subseteq \mathcal{SPC}(B)$
- *If A is a DNF of B then* $\mathrm{Paths}(A) \subseteq \mathrm{Paths}(B)$ *and* $\mathcal{SPC}(A) \subseteq \mathcal{SPC}(B)$

Proof.

Conversion to normal form works only on the outer Boolean structure, and so it cannot interfere with the simple subformulas or add new ones, it can at most delete them. □

Proposition 63. *For all formulas A and B:*

- $A \lhd_l B \implies \mathrm{Paths}(A) \subseteq \mathrm{Paths}(B)$
- $A \lhd_r B \implies \mathrm{Paths}(A) \subseteq \mathrm{Paths}(B)$

Proof. We consider only the first case, the second case is very similar.

Let $A = C'\mathcal{S}D$ and $B = C\mathcal{S}D$. By proposition 62, every path of C' is also a path of C, which means every path of A is also a path of B. □

Given these, the proofs for $\lhd_l$ and $\lhd_r$ are now straightforward.

Proposition 64. $A \lhd_l B \implies f(A) < f(B)$

Proof. Let $A = C'\mathcal{S}D$ and $B = C\mathcal{S}D$. By proposition 63, $\mathrm{Paths}(A) \subseteq \mathrm{Paths}(B)$ and so $d(A) \leq d(B)$. And by proposition 62, $\mathrm{Paths}(C') \subseteq \mathrm{Paths}(C)$, which means $d(C') \leq d(C)$ and $c(A) = d(C') + d(D) \leq d(C) + d(D) = c(B)$. But C' is in CNF while C is not, so l is reduced. □

Proposition 65. $A \lhd_r B \implies f(A) < f(B)$

Proof. Let $A = C\mathcal{S}D'$ and $B = C\mathcal{S}D$. Similarly to proposition 64, $d(A) \leq d(B)$ and $c(A) \leq c(B)$. We have $l(A) = l(B)$ since they have the same left side, but $r(A) = 0$ and $r(B) = 1$. □

The Dual of a path is defined in the obvious way, by swapping $\mathcal{S}$ and $\mathcal{U}$.

Proposition 66. *For every formula A:*

$$\mathrm{Paths}(A) = \mathrm{Dual}[\mathrm{Paths}(\mathrm{Dual}(A))]$$

Proof. Consider the syntax trees of A and $\mathrm{Dual}(A)$. These are identical except for the labels of the temporal operators, which are dual. Given a path of one of the formulas, the corresponding syntax path also corresponds to the dual of the path in the other tree. □

Proposition 67. $A \lhd_u B \implies f(A) < f(B)$

Proof. By proposition 66, the paths of A are the duals of the paths of B, and so temporal depth is preserved everywhere, giving $d(A) = d(B)$ and $c(A) = c(B)$. Since B is an $\mathcal{U}$, $l(B) = r(B) = 1$, which is the maximum in those orders, so $l(A) \leq l(B)$ and $r(A) \leq r(B)$. But we have $u(A) < u(B)$, as A is a $\mathcal{S}$ and B is an $\mathcal{U}$, by definition of $\lhd_u$. □

Proposition 68. $(\prec \cup \lessdot \cup \lhd_l \cup \lhd_r \cup \lhd_u)$ *is well-founded.*

Proof. Propositions 60, 61, 64, 65, 67, and 54. □

We still need to show that these are compatible with $<_{\mathcal{W}}$. The following definitions and proposition make clear how this will be done.

Definition 69. Let R be a binary relation over X.

Then $\sim_R$ is the equivalence relation over X defined by:

$$x \sim_R y \iff \text{for all } z \in X\text{:} \begin{array}{c} zRx \iff zRy \\ xRz \iff yRz \end{array}$$

And we say that x and y are R-*indistinguishable* if $x \sim_R y$.

Definition 70. Let R, S be binary relations over X.

Then we say that S *extends* R if (for all $x, y \in X$):

$$xSy \implies xRy \text{ or } x \sim_R y$$

Proposition 71. *Let R and S be well-founded binary relations over X such that S extends R.*

Then $R \cup S$ is well-founded.

Proof. Let $Y \subseteq X$ be non-empty.
Let $Y_R = \{x \in X \mid x \text{ is } R\text{-minimal in } Y\}$, this is non-empty because R is well-founded.
Let $m \in \{x \in Y_R \mid x \text{ is } S\text{-minimal in } Y_R\}$, this is non-empty because S is well-founded.
We will show that m is $(R \cup S)$-minimal in Y.

Let $x \in Y$, then we have two possibilities, $x \in Y_R$ or $x \notin Y_R$.

If $x \in Y_R$ then $\langle x, m\rangle \notin R$ since m is R-minimal in Y and $\langle x, m\rangle \notin S$ since m is S-minimal in Y_R, hence $\langle x, m\rangle \notin R \cup S$.

If $x \notin Y_R$, assume $\langle x, m\rangle \in R \cup S$ for contradiction. Then either xRm or xSm. The first possibility contradicts the fact that m is R-minimal in Y.

In the second possibility, since S extends R, we also have either xRm or x is R-indistinguishable from m. The first case again contradicts the fact that m is R-minimal in Y, and the second case contradicts $x \notin Y_R$, as if x is R-indistinguishable from m, then it should also be R-minimal in Y and be in Y_R. □

Now we have to show that each of these relations extends $<_{\mathcal{W}}$. To do this we first need to prove some results related to $\mathcal{W}$.

Proposition 72. *For any paths π,λ,μ we have:*

(i) $\mathrm{Score}_1(\mathrm{Dual}(\pi)) = \mathrm{Score}_1(\pi)$
$\mathrm{Score}_0(\mathrm{Dual}(\pi)) = \mathrm{Score}_0(\pi)$

(ii) $\mathcal{D}(\pi\mu) \leq \mathcal{D}(\pi\lambda\mu)$

(iii) $\mathrm{Score}_1(\pi\mu) \leq \mathrm{Score}_1(\pi\lambda\mu)$
$\mathrm{Score}_0(\pi\mu) \leq \mathrm{Score}_0(\pi\lambda\mu)$

(iv) $\mathrm{Score}_1(\mathcal{S}^n\pi) \leq \mathrm{Score}_1(\mathcal{S}\pi)$
$\mathrm{Score}_1(\mathcal{U}^n\pi) \leq \mathrm{Score}_1(\mathcal{U}\pi)$

Proof.

(i) Swapping $\mathcal{S}$ with $\mathcal{U}$ does not affect the number of transitions, therefore the degree is identical.

It's also easy to see that this does not affect the length of the initial or final homogeneous segments.

(ii) We first consider the case where $\lambda = \mathcal{S}$.

Inserting a single $\mathcal{S}$ at the start of a path either increases the number of transitions, if the initial symbol is $\mathcal{U}$, or does not change it, if the initial symbol is $\mathcal{S}$ or if the path is empty.

Similarly, inserting a $\mathcal{S}$ at the end also cannot decrease the number of transitions.

If a $\mathcal{S}$ is inserted in between two symbols, the number of transitions increases if both symbols are $\mathcal{U}$, and remains the same in the other three cases.

If $\lambda = \mathcal{U}$:

$$\begin{aligned}
\mathcal{D}(\pi\mu) &= \mathcal{D}(\mathrm{Dual}(\pi\mu)) && \text{(by (i))}\\
&= \mathcal{D}(\mathrm{Dual}(\pi)\,\mathrm{Dual}(\mu)) && \text{(by definition of Dual)}\\
&\leq \mathcal{D}(\mathrm{Dual}(\pi)\,\mathcal{S}\,\mathrm{Dual}(\mu)) && \text{(by the } \lambda = \mathcal{S} \text{ case above)}\\
&= \mathcal{D}(\mathrm{Dual}(\pi\mathcal{U}\mu)) && \text{(by definition of Dual)}\\
&= \mathcal{D}(\pi\mathcal{U}\mu) && \text{(by (i))}
\end{aligned}$$

The general case can then be obtained by induction on the length of λ.

(iii) By (ii) we have $\mathcal{D}(\pi\mu) \leq \mathcal{D}(\pi\lambda\mu)$.

Assume that $\mathcal{D}(\pi\mu) = \mathcal{D}(\pi\lambda\mu)$, $\mathcal{L}_1(\pi\mu) > \mathcal{L}_1(\pi\lambda\mu)$ and $\lambda \neq \langle\rangle$, we show that this is a contradiction.

By assumption, $\mathcal{L}_1(\pi\mu) > 0$ as 0 is a minimum. So let us assume that $\pi\mu = \tau\mathcal{U}^{n+1}$, with $\mathcal{U}^{n+1}$ the final homogeneous segment. The proof for the other case being obtained by swapping $\mathcal{S}$ with $\mathcal{U}$.

If λ is inserted before the final homogeneous segment, then it cannot reduce the length of such segment, hence we can assume that λ was inserted inside the final segment.

If $\lambda = \mathcal{U}^{m+1}$ (for some m), then the insertion inside the final segment only increases its length, this implies that λ must have at least one $\mathcal{S}$. But this means that $\mathcal{D}$ has increased, a contradiction.

The proof for Score_0 goes along by very similar argumentation, considering the initial homogeneous segment instead of the final one.

(iv) We show only the $\mathcal{S}$ case.

If $n = 0$ we can use (iii) directly. Assume then that $n > 0$.

There are no transitions in $\mathcal{S}^{n-1}$, or between $\mathcal{S}^{n-1}$ and $\mathcal{S}$, so deleting $\mathcal{S}^{n-1}$ does not affect degree.

If the final homogeneous segment is the whole of $\mathcal{S}^n\pi$, then $\mathcal{L}_1(\mathcal{S}^n\pi) = \mathcal{L}_1(\mathcal{S}\pi) = 0$. Otherwise, the deletion of the $\mathcal{S}^{n-1}$ prefix does not affect the final homogeneous segment, and so $\mathcal{L}_1(\mathcal{S}^n\pi) = \mathcal{L}_1(\mathcal{S}\pi)$ again.

□

Item (iv) of the previous proposition is the reason why we defined $\mathcal{L}_1$ to be zero in case the path has degree zero. This is technically not required for our proof of correctness, but it makes things easier. In the case of $\mathcal{L}_0$, we do so purely for symmetry with $\mathcal{L}_1$.

Proposition 73. *Let $A \in \mathcal{L}_{TL}$ and $x \in \mathcal{C}(A)$. Then* $\mathrm{Paths}(x) \subseteq \mathrm{Paths}(A)$.

Proof. Let $x = \langle \pi, B \rangle$ and let $\lambda \in \mathrm{Paths}(x)$. We then have $\lambda = \pi\lambda'$ where λ' is a path of B.

The syntax tree of B is a subtree of the one of A, and there's a syntax path from the root of A to the root of B, corresponding to π. And another syntax path from the root of B to a leaf, corresponding to λ'.

The concatenation of these syntax paths is a syntax path of A and corresponds to $\pi\lambda' = \lambda$. □

The next proposition shows that the Score_1 of a formula is precisely the Score_1 of its leaders.

Proposition 74. *Let $A \in \mathcal{L}_{TL}$ and $x \in \mathcal{SPC}(A)$. Then:*

$$\mathrm{Score}_1(x) = \mathrm{Score}_1(A) \iff x \in \mathrm{Leaders}(A)$$

Proof. ($\Leftarrow$) Let $x \in \mathrm{Leaders}(A)$.

By proposition 73, any path of x is also one of A, in particular the maximum Score_1 one, hence $\mathrm{Score}_1(x) \leq \mathrm{Score}_1(A)$. Now we need to show that $\mathrm{Score}_1(A) \leq \mathrm{Score}_1(x)$ as well.

If $\mathrm{Paths}(A) = \emptyset$ then $\mathrm{Score}_1(A) = \langle 0, 0 \rangle$ which is the minimum so $\mathrm{Score}_1(x)$ can't be any lower.

From now on we assume that $\mathrm{Paths}(A) \neq \emptyset$ and let π be a maximum Score_1 path of A.

Since we are assuming that x is a leader, it is enough to find some $y \in \mathcal{SPC}(A)$ such that $\mathrm{Score}_1(\pi) \leq \mathrm{Score}_1(y)$, since we also have $\mathrm{Score}_1(y) = \mathrm{Score}_1(x)$ by definition of leader.

If the syntax path corresponding to π ends in p then $\langle \pi, p \rangle \in \mathcal{SPC}(A)$ and $\mathrm{Score}_1(\pi) = \mathrm{Score}_1(\langle \pi, p \rangle)$.

If the syntax path ends in a leaf with $\bot$ or $\top$, then notice that by the definition of Paths no formula has empty paths. The path π is then not empty and so we can go up the syntax tree from this leaf until we reach a $\mathcal{S}$ or an $\mathcal{U}$. Call B the simple subformula of A obtained in this way and λ its context. Notice that we have $\pi = \lambda\mathcal{S}$ or $\pi = \lambda\mathcal{U}$, depending on which temporal operator is B's outer operator. For convenience, assume that we are in the $\mathcal{S}$ case, that is $B = C\mathcal{S}D$ (for some formulas C and D) and $\pi = \lambda\mathcal{S}$. The $\mathcal{U}$ case is very similar.

If B is pure, then $\langle \lambda, B \rangle \in \mathcal{SPC}(A)$, and there is some μ such that $\mathcal{S}\mu$ is a path of B, hence $\lambda\mathcal{S}\mu$ is a path of $\langle \lambda, B \rangle$. By proposition 72, $\mathrm{Score}_1(\lambda\mathcal{S}) \leq \mathrm{Score}_1(\lambda\mathcal{S}\mu)$.

If B is not pure, then there must be an $\mathcal{U}$ in either C or D, and so there is some path μ that contains an $\mathcal{U}$ such that $\lambda\mathcal{S}\mu$ is a path of $\langle \lambda, B \rangle$, and thus also a path of A by proposition 73. This path $\lambda\mathcal{S}\mu$ has at least one additional transition from $\mathcal{S}$ to $\mathcal{U}$ and so its degree is greater than the one of π, which contradicts the assumption that π is a maximum Score_1 path of A.

($\Rightarrow$) Assume $\mathrm{Score}_1(x) = \mathrm{Score}_1(A)$ and let $y \in \mathrm{Leaders}(A)$.

By the ($\Leftarrow$) case of this proposition, $\mathrm{Score}_1(y) = \mathrm{Score}_1(A)$. Hence $\mathrm{Score}_1(x) = \mathrm{Score}_1(y)$, which means that x also has maximum Score_1 in $\mathcal{SPC}(A)$ and is a leader. □

Corollary 75. *Let $A \in \mathcal{L}_{TL}$ and $x \in \mathcal{SPC}(A)$. Then $\mathrm{Score}_1(x) \leq \mathrm{Score}_1(A)$.*

Proof. Let y be a leader of A. By proposition 74 and the definition of leader, $\mathrm{Score}_1(x) \leq \mathrm{Score}_1(y) = \mathrm{Score}_1(A)$. □

Proposition 76. *For all $A, B \in \mathcal{L}_{TL}$:*

$$\mathcal{SPC}(A) \subseteq \mathcal{SPC}(B) \implies \mathcal{W}(A) \leq \mathcal{W}(B)$$

Proof. Let $x \in \mathrm{Leaders}(A) \subseteq \mathcal{SPC}(A) \subseteq \mathcal{SPC}(B)$. Then by proposition 74 and corollary 75, $\mathrm{Score}_1(A) = \mathrm{Score}_1(x) \leq \mathrm{Score}_1(B)$.

To see that $\mathrm{Score}_0(A) \leq \mathrm{Score}_0(B)$, let $\pi \in \mathrm{Paths}(A)$ have maximal Score_0 (in $\mathrm{Paths}(A)$). The argument is very similar to the one used in the proof of the ($\Leftarrow$) case in proposition 74.

If the syntax path corresponding to π ends in p then $\langle \pi, p\rangle \in \mathcal{SPC}(A) \subseteq \mathcal{SPC}(B)$ and so π is also a path of B by proposition 73.

If the syntax path ends in $\perp$ or $\top$, then go up the tree until we get our $\langle \mu, C\rangle$ with C simple and $\pi = \mu\mathcal{S}$ or $\pi = \mu\mathcal{U}$. C must be pure as if C is not pure then it must contain one type of temporal operator inside the other, which means that A has a path with degree greater than π, a contradiction.

Since C is pure, $\langle \mu, C\rangle \in \mathcal{SPC}(A) \subseteq \mathcal{SPC}(B)$. We know that $\langle \mu, C\rangle$ has paths of the form $\pi\tau$, which have Score_0 not smaller than π, by proposition 72. By proposition 73, every such $\pi\tau$ is also a path of B, hence the Score_0 of B is not lower.

Now assume $\mathrm{Score}_1(A) = \mathrm{Score}_1(B)$ and let $x \in \mathrm{Leaders}(A)$. By proposition 74, $\mathrm{Score}_1(x) = \mathrm{Score}_1(A) = \mathrm{Score}_1(B)$, which means x is a leader of B, again by proposition 74. This shows that if the Score_1 of B is not higher, then it has at least as many leaders as A. □

Proposition 77. *For all paths π,λ,μ and formulas A,B:*

(i) $\mathrm{Score}_1(\langle \pi\mu, A\rangle) \leq \mathrm{Score}_1(\langle \pi\lambda\mu, A\rangle)$

(ii) $\mathrm{Score}_1(\langle \mathcal{S}^n\pi, A\rangle) \leq \mathrm{Score}_1(\langle \mathcal{S}\pi, A\rangle)$

(iii) If B is a subformula of A then $\mathrm{Score}_1(\langle \pi, B\rangle) \leq \mathrm{Score}_1(\langle \pi, A\rangle)$

Proof.

(i)

$$\begin{aligned}
\mathrm{Score}_1(\langle \pi\mu, A\rangle) &= \max\{\mathrm{Score}_1(\pi\mu\tau) \mid \tau \in \mathrm{Paths}(A)\} && \text{(by definition of } \mathrm{Score}_1\text{)} \\
&\leq \max\{\mathrm{Score}_1(\pi\lambda\mu\tau) \mid \tau \in \mathrm{Paths}(A)\} && \text{((iii) of proposition 72)} \\
&= \mathrm{Score}_1(\langle \pi\lambda\mu, A\rangle) && \text{(by definition of } \mathrm{Score}_1\text{)}
\end{aligned}$$

(ii)

$$\begin{aligned}
\mathrm{Score}_1(\langle \mathcal{S}^n\pi, A\rangle) &= \max\{\mathrm{Score}_1(\mathcal{S}^n\pi\lambda) \mid \lambda \in \mathrm{Paths}(A)\} && \text{(by definition of } \mathrm{Score}_1\text{)}\\
&\leq \max\{\mathrm{Score}_1(\mathcal{S}\pi\lambda) \mid \lambda \in \mathrm{Paths}(A)\} && \text{((iv) of proposition 72)}\\
&= \mathrm{Score}_1(\langle \mathcal{S}\pi, A\rangle) && \text{(by definition of } \mathrm{Score}_1\text{)}
\end{aligned}$$

(iii) Given any path $\lambda \in \mathrm{Paths}(\langle \pi, B\rangle)$, we have $\lambda = \pi\lambda'$ for some $\lambda' \in \mathrm{Paths}(B)$.

Since B is a subformula of A, there is a syntax path from the root of A to the root of B, call the corresponding path μ. Then $\mu\lambda' \in \mathrm{Paths}(A)$, which means $\pi\mu\lambda' \in \mathrm{Paths}(\langle \pi, A\rangle)$.

By proposition 72, $\mathrm{Score}_1(\pi\lambda') \leq \mathrm{Score}_1(\pi\mu\lambda')$.

□

We can now show that taking a subformula does not increase $\mathcal{W}$.

Proposition 78. *For all formulas A and B: $A \prec B \implies \mathcal{W}(A) \leq \mathcal{W}(B)$*

Proof. Assume $A \prec B$.

If B is one of the Boolean connectives then $\mathcal{SPC}(A) \subseteq \mathcal{SPC}(B)$, which implies $\mathcal{W}(A) \leq \mathcal{W}(B)$ by proposition 76.

Assume then that B is a $\mathcal{S}$ (the $\mathcal{U}$ case is very similar).

Let $\pi \in \mathrm{Paths}(A)$, then $\mathcal{S}\pi \in \mathrm{Paths}(B)$. By proposition 72, $\mathrm{Score}_1(\pi) \leq \mathrm{Score}_1(\mathcal{S}\pi)$ and $\mathrm{Score}_0(\pi) \leq \mathrm{Score}_0(\mathcal{S}\pi)$. This shows $\mathrm{Score}_1(A) \leq \mathrm{Score}_1(B)$ and $\mathrm{Score}_0(A) \leq \mathrm{Score}_0(B)$. For the rest of the proof, assume $\mathrm{Score}_1(A) = \mathrm{Score}_1(B)$.

Let $\langle \pi, C\rangle \in \mathrm{Leaders}(A)$. Then $\langle \mathcal{S}\pi, C\rangle \in \mathcal{SPC}(B)$.

By propositions 74, 77, and corollary 75, $\mathrm{Score}_1(A) = \mathrm{Score}_1(\langle \pi, C\rangle) \leq \mathrm{Score}_1(\langle \mathcal{S}\pi, C\rangle) \leq \mathrm{Score}_1(B)$. But by assumption $\mathrm{Score}_1(A) = \mathrm{Score}_1(B)$, so $\mathrm{Score}_1(\langle \mathcal{S}\pi, C\rangle) = \mathrm{Score}_1(B)$ and $\langle \mathcal{S}\pi, C\rangle$ is also a leader of B. This shows $\mathcal{N}(A) \leq \mathcal{N}(B)$. □

Proposition 79. *For all formulas A and B: $A \lessdot B \implies \mathcal{W}(A) \leq \mathcal{W}(B)$*

Proof. Assume $A \lessdot B$.

We look at the case where $A = C'\mathcal{S}D$ and $B = C\mathcal{S}D$ with $C' \prec^+ C$. The other case is very similar.

We first show that $\mathrm{Score}_1(A) \leq \mathrm{Score}_1(B)$ and $\mathrm{Score}_0(A) \leq \mathrm{Score}_0(B)$ by showing that for every path of A, there's a path of B with a larger or equal Score_1 and Score_0.

Let $\pi \in \mathrm{Paths}(A)$. Then $\pi = \mathcal{S}\pi'$, with $\pi' \in \mathrm{Paths}(C')$ or $\pi' \in \mathrm{Paths}(D)$.

If $\pi' \in \mathrm{Paths}(D)$ then $\pi \in \mathrm{Paths}(B)$ as well.

If $\pi' \in \mathrm{Paths}(C')$ then, since C' is a subformula of C, there is a λ such that $\lambda\pi' \in \mathrm{Paths}(C)$, which means $\mathcal{S}\lambda\pi' \in \mathrm{Paths}(B)$. By proposition 72, $\mathrm{Score}_1(\mathcal{S}\pi') \leq \mathrm{Score}_1(\mathcal{S}\lambda\pi')$ and $\mathrm{Score}_0(\mathcal{S}\pi') \leq \mathrm{Score}_0(\mathcal{S}\lambda\pi')$.

Now assume $\mathrm{Score}_1(A) = \mathrm{Score}_1(B)$ and let us show that in that case $\mathcal{N}(A) \leq \mathcal{N}(B)$.

Let $\langle \pi, E \rangle \in \mathrm{Leaders}(A)$.

In case $E = A$ then A is pure and we have $\mathrm{Score}_1(A) = \langle 0, 0 \rangle$. Since we are assuming that the Score_1 of B is identical to the one of A, B is also pure and $\langle \langle \rangle, B \rangle$ is a leader of B.

Let us assume then that $E \neq A$. Then $\pi = \mathcal{S}\pi'$ and either $\langle \pi', E \rangle \in \mathcal{SPC}(C')$ or $\langle \pi', E \rangle \in \mathcal{SPC}(D)$.

If $\langle \pi', E \rangle \in \mathcal{SPC}(D)$ then $\langle \pi, E \rangle \in \mathcal{SPC}(B)$, and also $\langle \pi, E \rangle \in \mathrm{Leaders}(B)$ since $\mathrm{Score}_1(\langle \pi, E \rangle) = \mathrm{Score}_1(A) = \mathrm{Score}_1(B)$, by proposition 74 and assumption.

If $\langle \pi', E \rangle \in \mathcal{SPC}(C')$, then since C' is a subformula of C, there is some syntax path from the root of C to the root of C', corresponding to a path λ, which means $\langle \lambda\pi', E \rangle \in \mathcal{SPC}(C)$. Hence $\langle \mathcal{S}\lambda\pi', E \rangle \in \mathcal{SPC}(B)$. By proposition 77, $\mathrm{Score}_1(\langle \mathcal{S}\pi', E \rangle) \leq \mathrm{Score}_1(\langle \mathcal{S}\lambda\pi', E \rangle)$. This again means that $\langle \mathcal{S}\lambda\pi', E \rangle \in \mathrm{Leaders}(B)$ since $\langle \mathcal{S}\pi', E \rangle$ is a leader of A and $\mathrm{Score}_1(A) = \mathrm{Score}_1(B)$. □

Proposition 80. *For all formulas A and B:*

$$A \lhd_l B \implies \mathcal{W}(A) \leq \mathcal{W}(B)$$

$$A \lhd_r B \implies \mathcal{W}(A) \leq \mathcal{W}(B)$$

Proof. By proposition 63, $\mathrm{Paths}(A) \subseteq \mathrm{Paths}(B)$, and so $\mathrm{Score}_1(A) \leq \mathrm{Score}_1(B)$ and $\mathrm{Score}_0(A) \leq \mathrm{Score}_0(B)$. Assume $\mathrm{Score}_1(A) = \mathrm{Score}_1(B)$. We consider only one of the cases, so let $A = C'\mathcal{S}D$ and $B = C\mathcal{S}D$ with C' a CNF of C. We have to verify that B has at least as many leaders as A.

By proposition 62, $\mathcal{SPC}(C') \subseteq \mathcal{SPC}(C)$. The simple pure contexted subformulas of A are the ones of C' and D, prefixed with $\mathcal{S}$, and possibly also A itself. In the first two cases, they are also clearly simple pure contexted subformulas of B. So far this means that the leaders of A are also leaders of B, we only have to check that the case of A also being pure is not problematic.

If A is pure, then $\mathcal{D}(A) = 0$, and so $\mathcal{D}(B) = 0$, which means B is also pure. Therefore $\langle \langle \rangle, B \rangle \in \mathcal{SPC}(B)$. □

We can extend Dual to contexted formulas in the obvious way, $\mathrm{Dual}(\langle \pi, A \rangle) = \langle \mathrm{Dual}(\pi), \mathrm{Dual}(A) \rangle$. It has the following properties:

Proposition 81. *Let $A \in \mathcal{L}_{TL}$ and let π be a path, then:*

(i) $\mathrm{Score}_1(\mathrm{Dual}(A)) = \mathrm{Score}_1(A)$
$\mathrm{Score}_0(\mathrm{Dual}(A)) = \mathrm{Score}_0(A)$

(ii) $\mathcal{SPC}(A) = \mathrm{Dual}[\mathcal{SPC}(\mathrm{Dual}(A))]$

(iii) $\mathrm{Paths}(\langle \pi, A \rangle) = \mathrm{Dual}[\mathrm{Paths}(\mathrm{Dual}(\langle \pi, A \rangle))]$

(iv) $\mathrm{Score}_1(\mathrm{Dual}(\langle \pi, A \rangle)) = \mathrm{Score}_1(\langle \pi, A \rangle)$
$\mathrm{Score}_0(\mathrm{Dual}(\langle \pi, A \rangle)) = \mathrm{Score}_0(\langle \pi, A \rangle)$

(v) $\mathcal{N}(\mathrm{Dual}(A)) = \mathcal{N}(A)$

(vi) $\mathcal{W}(\mathrm{Dual}(A)) = \mathcal{W}(A)$

Proof.

(i) Proposition 66 tells us that A and $\mathrm{Dual}(A)$ have dual paths, which have identical Score_1 and Score_0 by proposition 72.

(ii) If we consider the syntax trees of both A and $\mathrm{Dual}(A)$, they are identical except for the labels of the temporal operators, which are dual. Any contexted subformula has a corresponding syntax path for the context, and a subtree for the subformula. These syntax path corresponds to the dual context in the other tree, and the dual subtree corresponds to the dual subformula.

(iii) ($\subseteq$) Let $\lambda \in \mathrm{Paths}(\langle \pi, A\rangle)$, then $\lambda = \pi\lambda'$ with $\lambda' \in \mathrm{Paths}(A)$. By proposition 66, $\mathrm{Dual}(\lambda') \in \mathrm{Paths}(\mathrm{Dual}(A))$, and so $\mathrm{Dual}(\lambda) = \mathrm{Dual}(\pi)\,\mathrm{Dual}(\lambda') \in \mathrm{Paths}(\langle \mathrm{Dual}(\pi), \mathrm{Dual}(A)\rangle) = \mathrm{Paths}(\mathrm{Dual}(\langle \pi, A\rangle))$.

($\supseteq$) Let $\lambda \in \mathrm{Dual}[\mathrm{Paths}(\mathrm{Dual}(\langle \pi, A\rangle))] = \mathrm{Dual}[\mathrm{Paths}(\langle \mathrm{Dual}(\pi), \mathrm{Dual}(A)\rangle)]$. Then $\lambda = \mathrm{Dual}(\mathrm{Dual}(\pi)\mu) = \pi\,\mathrm{Dual}(\mu)$ with $\mu \in \mathrm{Paths}(\mathrm{Dual}(A))$. By proposition 66, $\mathrm{Dual}(\mu) \in \mathrm{Paths}(A)$, so $\pi\,\mathrm{Dual}(\mu) \in \mathrm{Paths}(\langle \pi, A\rangle)$.

(iv) By (iii), the paths of $\mathrm{Dual}(\langle \pi, A\rangle)$ are the duals of the paths of $\langle \pi, A\rangle$, and by proposition 72 dual paths have identical Score_1 and Score_0.

(v) The $\mathcal{SPC}$ of A and B are duals by (ii), and the Score_1 is identical by (i).

Given a leader of one of them, the dual of this leader will be a leader of the dual formula since dual contexted formulas have identical Score_1 by (iv).

(vi) (i) and (v).

$\square$

Corollary 82. *For all* $A, B \in \mathcal{L}_{TL}$*:* $A \lhd_u B \implies \mathcal{W}(A) = \mathcal{W}(B)$

Proof. By proposition 81, $\mathcal{W}(A) = \mathcal{W}(\mathrm{Dual}(A))$, and by the definition of $\lhd_u$, $\mathrm{Dual}(A) = B$. $\square$

We can finally show that our main relation is a well-founded partial order.

Proposition 83. *The relation over formulas $<$, defined in 59, is a well-founded partial order.*

Proof. All of $<_{\mathcal{W}}$, $\prec$, $\lessdot$, $\lhd_l$, $\lhd_r$, and $\lhd_u$ are irreflexive, therefore so is their union. And it is transitive by construction. Therefore it is a partial order.

By propositions 78, 79, 80 and corollary 82, $(\prec \cup \lessdot \cup \lhd_l \cup \lhd_r \cup \lhd_u)$ extends $<_{\mathcal{W}}$. This together with proposition 71 tell us that $(<_{\mathcal{W}} \cup \prec \cup \lessdot \cup \lhd_l \cup \lhd_r \cup \lhd_u)$ is well-founded. Finally, the transitive closure of a well-founded relation is also well-founded. $\square$

Next we show that transformations do not increase Score_1, proving first some results to make this easier.

Proposition 84. *For every transformation, let $X(A, B, F, G)$ be the starting formula and X' the resulting formula.*

Then X' is constructed using only A,B,F,G,$F\mathcal{U}G$,$\top$,$\neg$,$\vee$,$\wedge$ and $\mathcal{S}$.

Moreover, $F\mathcal{U}G$ is never inside a $\mathcal{S}$ in this construction.

Proof. Every explicit use of $\mathcal{U}$ in any transformation is in the form $F\mathcal{U}G$ and never inside any other temporal operator. This is true even in the transformations that use other transformations, as that is always done with $F := F$ and $G := G$. □

Definition 85. Let $A \in \mathcal{L}_{\mathsf{TL}}$ and $S \subseteq \mathcal{C}(X)$. We say S is *complete* (for A) if $\bigcup \mathrm{Paths}[S] = \mathrm{Paths}(A)$.

Proposition 86. *Let $A \in \mathcal{L}_{\mathit{TL}}$ and let $S \subseteq \mathcal{C}(A)$ be non-empty and complete.*
Then $\mathrm{Score}_1(A) = \max \mathrm{Score}_1[S]$.

Proof. The maximum Score_1 of S is the maximum Score_1 of a path of an element of S, which by definition of complete considers exactly the paths of A. □

This notion of completeness is too strong for our purposes, we can define a weaker notion of completeness that is still sufficient by allowing deletion of contexted formulas when they also appear with a stronger context and the same formula.

Definition 87. Let $A \in \mathcal{L}_{\mathsf{TL}}$ and $S \subseteq \mathcal{C}(A)$. We say S is *quasi-complete* (for A) if there is an $S' \subseteq \mathcal{C}(A)$ such that:

- $S \subseteq S'$
- S' is complete
- For all $\langle \pi, C \rangle \in S'$ there is a path λ obtained from π by performing a finite number of insertions and $\langle \lambda, C \rangle \in S$

We call S' a *completion* of S.

Proposition 88. *Let $A \in \mathcal{L}_{\mathit{TL}}$ and let $S \subseteq \mathcal{C}(A)$ be non-empty and quasi-complete.*
Then $\mathrm{Score}_1(A) = \max \mathrm{Score}_1[S]$.

Proof. Let S' be a completion of S. We prove $\max \mathrm{Score}_1[S] = \max \mathrm{Score}_1[S']$ and then use proposition 86.

We have $S \subseteq S'$, therefore $\mathrm{Score}_1[S] \subseteq \mathrm{Score}_1[S']$.

For the other direction, let $\langle \pi, C \rangle \in S'$. Then, since S' is a completion of S, there is a $\langle \lambda, C \rangle \in S$ and λ is obtained by inserting a finite number of elements into π, which means that $\mathrm{Score}_1(\langle \pi, C \rangle) \leq \mathrm{Score}_1(\langle \lambda, C \rangle)$ by proposition 77. □

Proposition 89. *For every transformation, let $X(A, B, F, G)$ be the starting formula and X' the resulting formula. Then:*

- $S = \{\langle \mathcal{S}, A \rangle, \langle \mathcal{S}, B \rangle, \langle \mathcal{S}, F\mathcal{U}G \rangle\}$ *is a complete set for X.*

- *There are* $n_A, n_B, n_F, n_G, n_\top \in \mathbb{N}$ *such that some non-empty subset of*

$$S' = \{\langle \mathcal{S}^{n_A}, A\rangle, \langle \mathcal{S}^{n_B}, B\rangle, \langle \mathcal{S}^{n_F}, F\rangle, \langle \mathcal{S}^{n_G}, G\rangle, \langle\langle\rangle, F\,\mathcal{U}G\rangle, \langle \mathcal{S}^{n_\top}, \top\rangle\}$$

is a quasi-complete set for X'*.*

Proof. It's easy to see that S is complete for X.

For the second claim, given proposition 84, a path of X' is a path of $F\,\mathcal{U}G$, A, B, F, G or $\top$, possibly prefixed by some number of $\mathcal{S}$s in the case of the last five. □

Proposition 90. *For every transformation, let* $X(A, B, F, G)$ *be the starting formula and* X' *the resulting formula. Then* $\mathrm{Score}_1(X') \leq \mathrm{Score}_1(X)$*.*

Proof. Let S and S' be quasi-complete sets for X and X' (respectively) as given by proposition 89. Using proposition 77 we get:

- $\mathrm{Score}_1(\langle \mathcal{S}^{n_A}, A\rangle) \leq \mathrm{Score}_1(\langle \mathcal{S}, A\rangle)$
- $\mathrm{Score}_1(\langle \mathcal{S}^{n_B}, B\rangle) \leq \mathrm{Score}_1(\langle \mathcal{S}, B\rangle)$
- $\mathrm{Score}_1(\langle \mathcal{S}^{n_F}, F\rangle) \leq \mathrm{Score}_1(\langle \mathcal{S}, F\,\mathcal{U}G\rangle)$
- $\mathrm{Score}_1(\langle \mathcal{S}^{n_G}, G\rangle) \leq \mathrm{Score}_1(\langle \mathcal{S}, F\,\mathcal{U}G\rangle)$
- $\mathrm{Score}_1(\langle\langle\rangle, F\,\mathcal{U}G\rangle) \leq \mathrm{Score}_1(\langle \mathcal{S}, F\,\mathcal{U}G\rangle)$
- $\mathrm{Score}_1(\langle \mathcal{S}^{n_\top}, \top\rangle) = \langle 0, 0\rangle$ which is the minimum element in this order.

Which means $\max \mathrm{Score}_1[S'] \leq \max \mathrm{Score}_1[S]$ and thus $\mathrm{Score}_1(X') \leq \mathrm{Score}_1(X)$ by propositions 88 and 86. □

The next lemma is useful in case 4. It intuitively says that transformations do not introduce new simple non-past formulas and do not increase the Score_1 of any of them.

Lemma 91. *For every transformation, let* $X(A, B, F, G)$ *be the starting formula and* X' *the resulting formula.*

If $\langle \pi, C\rangle$ *is a simple and non-past contexted subformula of* X' *then there is a* λ *such that* $\langle \lambda, C\rangle \in \mathcal{SPC}(X)$ *and* $\mathrm{Score}_1(\langle \pi, C\rangle) \leq \mathrm{Score}_1(\langle \lambda, C\rangle)$*.*

Proof. By proposition 84, there are no explicit uses of variables in the construction of X' and the only explicit use of $\mathcal{U}$ is in $F\,\mathcal{U}G$, therefore C must be a subformula of A,B,F,G or $F\,\mathcal{U}G$ with some context μ. This means that C appears in X' with context $\mathcal{S}^n\mu$ (for some n).

Looking at X, we can see that A, B and $F\,\mathcal{U}G$ appear in a Boolean combination immediately inside the outer $\mathcal{S}$, therefore if C was taken from one of these it appears in X with a context $\mathcal{S}\mu$. If C comes from F or G, then it appears in X with context $\mathcal{SU}\mu$.

By proposition 77, $\mathrm{Score}_1(\langle \mathcal{S}^n\mu, C\rangle) \leq \mathrm{Score}_1(\langle \mathcal{S}\mu, C\rangle) \leq \mathrm{Score}_1(\langle \mathcal{SU}\mu, C\rangle)$, so in either case we have the desired result. □

The next theorem shows that Sep is correct. Item (ii) is useful in case 6.

Theorem 92. *For every $X \in \mathcal{L}_{TL}$ we have the following:*

(i) Any Sep calls that Sep(X) reduces to are with a smaller argument (i.e. Sep(X) terminates).

(ii) Let A be a simple and non-past subformula of Sep(X). If X has no $\mathcal{S}$ inside an $\mathcal{U}$ then A is a subformula of X.

(iii) $X \equiv$ Sep(X)

(iv) Sep(X) is separated.

Proof. We proceed by induction on $<$.

Notice that Sep exhaustively matches on formulas:

- Case 0 covers $\bot$, $\top$ and p
- Case 1 covers $\neg$
- Case 2 covers $\vee$
- Case 3 covers $\wedge$
- Cases 4,5,6 cover $\mathcal{S}$
- Case 7 covers $\mathcal{U}$

Which means that every X fits into some case. We now look at each case.

Case 0: Trivial.

Cases 1,2,3:

(i) A and B are subformulas of X.

(ii) Assume X has no $\mathcal{S}$ inside an $\mathcal{U}$. Let C be a simple and non-past subformula of Sep(X). For convenience, assume $C \in \mathrm{Subs}($Sep$(A))$, the B case being very similar. Since X has no $\mathcal{S}$ inside an $\mathcal{U}$, neither does A, and so $C \in \mathrm{Subs}(A) \subseteq \mathrm{Subs}(X)$ by the induction hypothesis.

(iii) Substitution by equivalent formula.

(iv) The induction hypothesis tells us that A and B are separated, and a Boolean combination of separated formulas is also separated.

Case 4: In all the subcases, let X' refer to the result of applying the corresponding transformation (e.g. in case 4.1, $X' = \mathcal{T}_1(A', B', F, G)$).

(i) The set $\{\langle \mathcal{S}, E\rangle \mid E \in \mathbf{A} \cup \mathbf{B} \cup \mathbf{C} \cup \mathbf{D} \cup \{F\mathcal{U}G\}\}$ is complete for X. All the E that come from $\mathbf{A}$ and $\mathbf{B}$ are non-future, therefore in this case $\langle \mathcal{S}, E\rangle$ has degree 0 and none can be leaders. All the elements of $\mathbf{C} \cup \mathbf{D} \cup \{F\mathcal{U}G\}$ are pure future and therefore their associated contexted formulas all

have degree 1. But $F\mathcal{U}G$ has maximal temporal depth amongst them, which means that $\langle \mathcal{S}, F\mathcal{U}G\rangle$ has a maximal final homogeneous segment and maximal $\mathcal{L}_1$, and is therefore a leader. This also implies that the degree of X is 1, by proposition 74.

By proposition 90, $\mathrm{Score}_1(X') \leq \mathrm{Score}_1(X)$. So let us assume $\mathrm{Score}_1(X') = \mathrm{Score}_1(X)$.

Since X has degree 1 and the outer connective of X is a $\mathcal{S}$, there is no $\mathcal{S}$ inside an $\mathcal{U}$ in X as this would imply that X has a degree at least 2. Given proposition 84, any $\mathcal{U}$ in X' must already appear in X, therefore X' also has no $\mathcal{S}$ inside an $\mathcal{U}$. Given this and the fact that X' has degree 1, a leader of X' must be of the form $\langle \mathcal{S}^{m+1}\mathcal{U}^n, H\rangle$, with H non-past.

By lemma 91 there is some $\langle \lambda, H\rangle \in \mathcal{SPC}(X)$ with $\mathrm{Score}_1(\langle \mathcal{S}^{m+1}\mathcal{U}^n, H\rangle) \leq \mathrm{Score}_1(\langle \lambda, H\rangle)$. We have:

$$
\begin{aligned}
& \mathrm{Score}_1(X') & \\
= \; & \mathrm{Score}_1(\langle \mathcal{S}^{m+1}\mathcal{U}^n, H\rangle) & \text{(proposition 74)} \\
\leq \; & \mathrm{Score}_1(\langle \lambda, H\rangle) & \\
\leq \; & \mathrm{Score}_1(X) & \text{(proposition 75)} \\
= \; & \mathrm{Score}_1(X') & \text{(assumption)}
\end{aligned}
$$

Therefore $\mathrm{Score}_1(\langle \lambda, H\rangle) = \mathrm{Score}_1(X)$, $\mathrm{Score}_1(\langle \lambda, H\rangle)$ is a leader of X, and $\mathcal{N}(X') \leq \mathcal{N}(X)$.

The formula $F\mathcal{U}G$ cannot appear in X inside one of the non-future literals (i.e. the ones in $\mathbf{A}$ and $\mathbf{B}$) since by assumption every literal is separated. And $F\mathcal{U}G$ cannot appear inside one of the future literals (i.e. the ones in $\mathbf{C}$ and $\mathbf{D}$) since otherwise such literal would have greater temporal depth, which contradicts the fact that $F\mathcal{U}G$ has maximal temporal depth. This together with proposition 84 means that the only possible occurrences of $F\mathcal{U}G$ in X' are the explicit ones, which always have an empty context. Hence $F\mathcal{U}G$ cannot be leader of X' as that would imply that X' has degree 0. This shows that X' has lost a leader.

(ii) Any simple and non-past subformula H of X' appears there with some context. Lemma 91 tells us that H also appears in X.

(iii) The transformations construct equivalent formulas, that is $X \equiv X'$, then the induction hypothesis tells us that $X' \equiv \text{Sep}(X')$. By definition of the algorithm we have $\text{Sep}(X) = \text{Sep}(X')$.

(iv) Direct use of the induction hypothesis.

Case 5:

(i) If we have both $m = 1$ and $n = 1$, then at least one of A or B must not be in normal form, as otherwise we would be in case 4. Therefore $A_1\mathcal{S}B_1 < A\mathcal{S}B$ (via $\lhd_l$ and/or $\lhd_r$).

We now assume that $m > 1$, the proof is analogous for the case where $n > 1$.

For all relevant is and js, A_i is a proper subformula of $\bigwedge_{i=1}^{m} A_i$ and B_j is a subformula of $\bigvee_{j=1}^{n} B_j$.

Hence $A_i \mathcal{S} B_j \lessdot \left(\bigwedge_{i=1}^{m} A_i\right) \mathcal{S} \left(\bigvee_{j=1}^{n} B_j\right)$.

We also have $\left(\bigwedge_{i=1}^{m} A_i\right) \mathcal{S} \left(\bigvee_{j=1}^{n} B_j\right) \leq A \mathcal{S} B$, since the left side is just conversion of the $\mathcal{S}$ operands to normal form, and so each operand is either identical or has been converted to normal form. This is again justified by $\lhd_l$ and/or $\lhd_r$.

(ii) Let C be a simple and non-past subformula of Sep$(A \mathcal{S} B)$. Then $C \in \mathrm{Subs}(\text{Sep}(A_i \mathcal{S} B_j))$ for some i and j since Sep$(A \mathcal{S} B)$ is a Boolean combination of formulas of this form.

Notice that every A_i and B_j is a clause whose literals are each (possibly negated) $\bot$, $\top$ or a simple formula. Each of these simple formulas must be a subformula of either A or B since normal form conversion is only over the outer Boolean structure and does not "make up" simple formulas.

The formulas A and B are both separated, hence have no $\mathcal{S}$ inside an $\mathcal{U}$, which means that neither does any A_i, B_j or $A_i \mathcal{S} B_j$. Therefore, by the induction hypothesis, $C \in \mathrm{Subs}(A_i \mathcal{S} B_j)$.

We know that $C \neq A_i \mathcal{S} B_j$ since the former is non-past. So we must have $C \in \mathrm{Subs}(A_i) \cup \mathrm{Subs}(B_j)$. Remember that C is simple and so must be a subformula of one of the simple formulas that appear as literals in A_i and B_j. Hence, as we have seen before, it is also a subformula of A or B, which implies that is is a subformula of $A \mathcal{S} B$.

(iii)

$$
\begin{aligned}
A \mathcal{S} B &\equiv \left(\bigwedge_{i=1}^{m} A_i\right) \mathcal{S} \left(\bigvee_{j=1}^{n} B_j\right) && \text{(Normal forms are equivalent)} \\
&\equiv \bigwedge_{i=1}^{m} A_i \mathcal{S} \left(\bigvee_{j=1}^{n} B_j\right) && \text{(corollary 20)} \\
&\equiv \bigwedge_{i=1}^{m} \bigvee_{j=1}^{n} A_i \mathcal{S} B_j && \text{(corollary 23)}
\end{aligned}
$$

(iv) By the induction hypothesis, all the Sep$(A_i \mathcal{S} B_j)$ are separated, and a Boolean combination of separated formulas is also separated.

Case 6:

(i) The inner calls, Sep(A) and Sep(B), are with a proper subformula. So let us look at the outer Sep call.

By the induction hypothesis, both Sep(A) and Sep(B) are separated, which implies that they have degree 0. Hence Sep$(A) \mathcal{S}$ Sep(B) has degree at most 1.

Notice that $A\mathcal{S}B$ cannot have degree 0 since otherwise we would be in case 0. If $A\mathcal{S}B$ has degree greater than 1 or if Sep$(A)\mathcal{S}$ Sep(B) has degree 0, we have reduced the degree. Assume then that both $A\mathcal{S}B$ and Sep$(A)\mathcal{S}$ Sep(B) have degree 1.

We first show that Score_1(Sep$(A)\mathcal{S}$ Sep$(B)) \leq \mathrm{Score}_1(A\mathcal{S}B)$.

Let $\langle \mathcal{S}\pi, C\rangle \in \mathrm{Leaders}$(Sep$(A)\mathcal{S}$ Sep(B)) with $\langle \pi, C\rangle \in \mathcal{SPC}$(Sep$(A)$) or $\langle \pi, C\rangle \in \mathcal{SPC}$(Sep$(B)$). Since both Sep$(A)$ and Sep(B) have degree 0, so does $\langle \pi, C\rangle$. This means that if π has any $\mathcal{S}$ then C must be non-future, but that would be mean that $\langle \mathcal{S}\pi, C\rangle$ has degree 0, which is a contradiction since it is the leader of a degree 1 formula. Hence $\pi = \mathcal{U}^n$ for some n.

Given this, we can walk up the syntax tree of Sep(A) or Sep(B) from the root of C until we consume all the n $\mathcal{U}$s, giving us a superformula D of C. Notice that D is simple by definition and must be non-past because otherwise would imply that either Sep(A) or Sep(B) is not separated.

Since we are assuming that $A\mathcal{S}B$ has degree 1, it cannot have a $\mathcal{S}$ inside an $\mathcal{U}$. Therefore we can use (ii) of the induction hypothesis to conclude that D is also a subformula of A or B, appears there with some context λ and so appears in $A\mathcal{S}B$ with a context $\mathcal{S}\lambda$. Remember that $\langle \pi, C\rangle \in \mathcal{SPC}(D)$, which means that $\langle \mathcal{S}\lambda\pi, C\rangle \in \mathcal{SPC}(A\mathcal{S}B)$.

We now have

$$\mathrm{Score}_1(\text{Sep}(A)\mathcal{S}\,\text{Sep}(B)) = \mathrm{Score}_1(\langle \mathcal{S}\pi, C\rangle) \leq \mathrm{Score}_1(\langle \mathcal{S}\lambda\pi, C\rangle) \leq \mathrm{Score}_1(A\mathcal{S}B)$$

Assume Score_1(Sep$(A)\mathcal{S}$ Sep$(B)) = \mathrm{Score}_1(A\mathcal{S}B)$. Then $\mathrm{Score}_1(\langle \mathcal{S}\lambda\pi, C\rangle) = \mathrm{Score}_1(A\mathcal{S}B)$ and so $\langle \mathcal{S}\lambda\pi, C\rangle$ is a leader of $A\mathcal{S}B$, and we get $\mathcal{N}$(Sep$(A)\mathcal{S}$ Sep$(B)) \leq \mathcal{N}(A\mathcal{S}B)$ as well.

We finally show that if neither Score_1 nor $\mathcal{N}$ decrease, then Score_0 must do so.

A maximum degree path of Sep$(A)\mathcal{S}$ Sep(B) must be of the form $\mathcal{S}\mathcal{U}^{m+1}$, since Sep$(A)\mathcal{S}$ Sep(B) has degree 1, Sep(A) and Sep(B) are separated, and the outer connective is $\mathcal{S}$.
Hence Score_0(Sep$(A)\mathcal{S}$ Sep$(B)) = \langle 1, 1\rangle$.

One of A or B must not be separated, otherwise we would be in case 0 instead. Assume that A is not separated, with the other case being analogous.

Then A has some path with degree 1, and it must start with $\mathcal{S}$ as otherwise $A\mathcal{S}B$ would have degree at least 2. This implies that $A\mathcal{S}B$ has a degree 1 path starting with $\mathcal{S}\mathcal{S}$, and so the $\mathcal{L}_0$ of this path is at least 2, and the Score_0 of $A\mathcal{S}B$ is at least $\langle 1, 2\rangle$.

(ii) Let C be a simple and non-past subformula of Sep(Sep$(A)\mathcal{S}$ Sep(B)).
By the induction hypothesis, both Sep(A) and Sep(B) are separated, and so do not have a $\mathcal{S}$ inside an $\mathcal{U}$, hence neither does Sep$(A)\mathcal{S}$ Sep(B). By the induction hypothesis again, $C \in \mathrm{Subs}$(Sep$(A)\mathcal{S}$ Sep(B)).
Additionally, $C \neq$ Sep$(A)\mathcal{S}$ Sep(B) since the former is non-past and so $C \in \mathrm{Subs}$(Sep(A)) or $C \in \mathrm{Subs}$(Sep(B)).

If we then assume that $A\mathcal{S}B$ has no $\mathcal{S}$ inside an $\mathcal{U}$, then neither do A nor B and the induction hypothesis gives us that $C \in \mathrm{Subs}(A)$ or $C\,\mathrm{Subs}(B)$, which means that C is also a subformula of $A\mathcal{S}B$.

(iii) By induction hypothesis $A \equiv \mathsf{Sep}(A)$ and $B \equiv \mathsf{Sep}(B)$, hence $A\mathcal{S}B \equiv \mathsf{Sep}(A)\mathcal{S}\,\mathsf{Sep}(B)$, which is equivalent to $\mathsf{Sep}(\mathsf{Sep}(A)\mathcal{S}\,\mathsf{Sep}(B))$ using the induction hypothesis one more time.

(iv) Immediate by the induction hypothesis.

Case 7:

(i) Immediate by $\lhd_u$.

(ii) Since we are here and not in case 0, $A\mathcal{U}B$ is not separated, and so there is a $\mathcal{S}$ inside the $\mathcal{U}$.

(iii)

$$\begin{aligned} & \mathrm{Dual}(X) \equiv \mathsf{Sep}(\mathrm{Dual}(X)) && \text{(induction hypothesis)} \\ \iff & \mathrm{Dual}(\mathrm{Dual}(X)) \equiv \mathrm{Dual}(\mathsf{Sep}(\mathrm{Dual}(X))) && \text{(proposition 18)} \\ \iff & X \equiv \mathrm{Dual}(\mathsf{Sep}(\mathrm{Dual}(X))) && (X = \mathrm{Dual}(\mathrm{Dual}(X)), \text{ by proposition 16}) \end{aligned}$$

(iv) $\mathsf{Sep}(\mathrm{Dual}(X))$ is separated, by the induction hypothesis. Separation is equivalent to having degree 0, and Dual preserves degree, by proposition 77.

□

Chapter 4

Translation

In this chapter we define an algorithm that given a FOMLO formula with at most one free variable, produces an equivalent temporal formula. This algorithm makes crucial use of the separation algorithm defined previously.

Intuitively, the main difficulty in performing this conversion is that in FOMLO we can always refer to any variable, no matter how many more levels of quantification there have been since the quantifier that introduced a variable, while temporal logic has a sequential nature, only allowing us to relate the "current point" with the next. This is overcome by converting the binary predicates in FOMLO to unary ones, "Before", "Now" and "After", that carry the same information, essentially converting FOMLO into truly monadic first-order logic without equality.

To understand how this works, we will go through an example. Consider the following formula, which intuitively means that when process 1 enters a critical section at time x ($E_1(x)$), it will leave the critical section at some time y after x ($L_1(y)$) and process 2 cannot enter the critical section while process 1 is in there.

$$E_1(x) \;\rightarrow\; \exists_y \,(y > x \;\wedge\; L_1(y) \;\wedge\; \forall_z \,(x \leq z < y \;\rightarrow\; \neg E_2(z)))$$

First, we get rid of binary predicates involving x, the free variable of the formula.

$$E_1(x) \;\rightarrow\; \exists_y \,(\mathrm{After}(y) \;\wedge\; L_1(y) \;\wedge\; \forall_z \,((\mathrm{Now}(z) \;\vee\; \mathrm{After}(z)) \;\wedge\; z < y \;\rightarrow\; \neg E_2(z)))$$

In this case, the unary predicates involving x already occur outside the scope of any quantifier, we would pull them out if they did not. Notice that the formula inside $\exists y$ now has no occurences of x, and we can proceed with translating it. Eliminating the binary predicates involving y we get:

$$E_1(x) \;\rightarrow\; \exists_y \left(\underbrace{\overline{\mathrm{After}}(y) \;\wedge\; L_1(y) \;\wedge\; \forall_z \left(\underbrace{\big(\overline{\mathrm{Now}}(z) \;\vee\; \overline{\mathrm{After}}(z)\big) \;\wedge\; \mathrm{Before}(z) \;\rightarrow\; \neg E_2(z)}_{\alpha}\right)}_{\beta}\right)$$

We have marked the predicates introduced in the previous iteration with a line over them, so that it is

clear that they are a quantification level above. And again in this case, the predicates involving only y already happen to occur outside of any additional quantification.

α is now just ordinary propositional logic, and can be translated as

$$A = \left(\overline{\text{Now}} \vee \overline{\text{After}}\right) \wedge \text{Before} \ \rightarrow \ \neg E_2$$

and $\forall_z \alpha$ is satisfied at any point iff all points z satisfy α, and can be translated by requiring that A be true in the past, present and future, that is $\forall_z \alpha$ is translated as $\blacksquare A \wedge A \wedge \square A$.

Since we now have a translation of $\forall_z \alpha$, β can be translated as

$$B = \overline{\text{After}} \wedge L_1 \wedge \blacksquare A \wedge A \wedge \square A$$

and $\exists_y \beta$ as $\blacklozenge B \vee B \vee \lozenge B$, the translation of the whole formula then being $E_1 \ \rightarrow \ \blacklozenge B \vee B \vee \lozenge B$.

Now we have a temporal logic formula, but it still has those extra predicates, and we need to remove them somehow. In this case, Before intuitively means that we are at a point before the point one level of temporal operator above, similarly $\overline{\text{After}}$ means that we are after the point two levels of temporal operator above.

We remove the new predicates level by level, in the reverse order that we introduced them. So, consider $\blacksquare A \wedge A \wedge \square A$, written in full:

$$\begin{aligned} & \blacksquare \left(\left(\overline{\text{Now}} \vee \overline{\text{After}}\right) \wedge \text{Before} \ \rightarrow \ \neg E_2\right) \\ \wedge \quad & \left(\overline{\text{Now}} \vee \overline{\text{After}}\right) \wedge \text{Before} \ \rightarrow \ \neg E_2 \\ \wedge \quad & \square \left(\left(\overline{\text{Now}} \vee \overline{\text{After}}\right) \wedge \text{Before} \ \rightarrow \ \neg E_2\right) \end{aligned}$$

The Before in the first line appears inside the $\blacksquare$, so we know that we are in the past, and this Before can be replaced with $\top$. Similarly, in the second line the Before appears in the present, and the one in the third line appears in the future, both can be replaced with $\bot$. This renders the second and third lines equivalent to $\top$ and so from $\blacksquare A \wedge A \wedge \square A$ we get

$$\blacksquare \left(\overline{\text{Now}} \vee \overline{\text{After}} \ \rightarrow \ \neg E_2\right)$$

We have removed one level of the new predicates, we now remove the second level, so what follows is $\blacklozenge B \vee B \vee \lozenge B$ written in full, replacing $\blacksquare A \wedge A \wedge \square A$ with the formula we just found and removing the lines over the predicates since that level is what we will now be looking at.

$$\begin{aligned} & \blacklozenge \left(\text{After} \wedge L_1 \wedge \blacksquare \left(\text{Now} \vee \text{After} \ \rightarrow \ \neg E_2\right)\right) \\ \vee \quad & \text{After} \wedge L_1 \wedge \blacksquare \left(\text{Now} \vee \text{After} \ \rightarrow \ \neg E_2\right) \\ \vee \quad & \lozenge \left(\text{After} \wedge L_1 \wedge \blacksquare \left(\text{Now} \vee \text{After} \ \rightarrow \ \neg E_2\right)\right) \end{aligned}$$

In the first line, the first After is immediately inside a past operator, and so can be set to $\bot$ since we are in the past, this makes the whole first line equivalent to $\bot$. In the second line we have a similar situation, the first After is in the present and so is equivalent to $\bot$ and so is the whole line. In the third case, the first After is in the future, and we can write $\top$ there.

This gives us

$$\Diamond\,(L_1 \wedge \blacksquare\,(\mathrm{Now} \vee \mathrm{After} \;\rightarrow\; \neg E_2))$$

Now we have a problem, the Now and After both appear inside a past operator inside a future operator. In that situation it is not necessary for us to be in either the past or the future, we could for example go to the past of the current point and then to the future of that point, while still remaining in the past in relation to the current point.

This is where separation comes in. We can separate the formula into a Boolean combination of pure formulas, and then easily tell if we are in the past, present or future.

Calling Sep on $\Diamond\,(L_1 \wedge \blacksquare\,(\mathrm{Now} \vee \mathrm{After} \;\rightarrow\; \neg E_2))$ we obtain, after minor simplification:

$$\begin{array}{ll} & \blacksquare\,(\mathrm{Now} \vee \mathrm{After} \;\rightarrow\; \neg E_2) \\ \wedge & \mathrm{Now} \vee \mathrm{After} \;\rightarrow\; \neg E_2 \\ \wedge & (\mathrm{Now} \vee \mathrm{After} \;\rightarrow\; \neg E_2)\mathcal{U}L_1 \end{array}$$

It is now easy to remove Now and After. They are both $\bot$ in the first line, $\top$ and $\bot$ in the second, $\bot$ and $\top$ in the third (respectively). The first line then becomes $\blacksquare\top \equiv \top$, the second $\neg E_2$ and the third $\neg E_2\mathcal{U}L_1$.

The translation of the original formula is then

$$E_1 \;\rightarrow\; \neg E_2 \wedge \neg E_2\mathcal{U}L_1$$

We worked with the whole formula in the example, the algorithm itself works recursively.

4.1 Algorithm

Before we present the translation algorithm itself, we need some auxiliary algorithms and definitions.

The idea of the following definition is that in a pulled out formula there are no occurrences of $P(x)$ inside irrelevant quantifiers. In the translation algorithm, we convert a formula into an equivalent pulled out formula before beginning the translation process.

Definition 93. We say that a FOMLO formula φ is *pulled out* if for every subformula of φ of the form $P(x)$, if $P(x)$ occurs inside the scope of a quantifier then the deepest quantifier inside whose scope $P(x)$ occurs binds x.

The following algorithm pulls out a formula. Normal form conversions are again only over the outer Boolean structure.

procedure Pullout(φ)

^{0}**if** $\varphi = \bot$ or $\varphi = \top$ or $\varphi = P(x)$ or $\varphi = (x = y)$ or $\varphi = (x < y)$ **then** φ

^{1}**else if** $\varphi = \neg\alpha$ **then** $\neg$Pullout(α)

^{2}**else if** $\varphi = \alpha \vee \beta$ **then** Pullout(α) $\vee$ Pullout(β)

^{3}**else if** $\varphi = \alpha \wedge \beta$ **then** Pullout(α) $\wedge$ Pullout(β)

^{4}**else if** $\varphi = \exists_x \alpha$

let $\bigvee_{i=1}^{m} \alpha_i$ be a DNF of Pullout(α)

for each $i \in [1, \ldots, m]$

let δ_i be the conjunction of the literals of the α_i clause where x appears free

let γ_i be the conjunction of the other literals (not in δ_i)

return $\bigvee_{i=1}^{m} (\gamma_i \wedge \exists_x \delta_i)$

^{5}**else if** $\varphi = \forall_x \alpha$

let $\bigwedge_{i=1}^{m} \alpha_i$ be a CNF of Pullout(α)

for each $i \in [1, \ldots, m]$

let δ_i be the disjunction of the literals of the α_i clause where x appears free

let γ_i be the disjunction of the other literals (not in δ_i)

return $\bigwedge_{i=1}^{m} (\gamma_i \vee \forall_x \delta_i)$

The following definition makes clear how we are getting rid of the binary predicates.

Definition 94 (Extend)**.** Given a variable t and a formula φ, Extend(t, φ) is a formula over a signature extended with three new unary predicate symbols Before, Now and After, obtained by applying the following substitutions to φ whenever they occur in a context where t is free:

- $(t = t) \mapsto \top$
- $(t < t) \mapsto \bot$
- $(x < t) \mapsto \mathrm{Before}(x)$
- $(x = t) \mapsto \mathrm{Now}(x)$

- $(t = x) \mapsto \text{Now}(x)$
- $(t < x) \mapsto \text{After}(x)$

These predicates are ordinary predicates, all we have to do is add three new symbols to Pred.

And then to remove these extra monadic predicates:

Definition 95 (Unextend)**.** Given a separated temporal formula A over an extended signature, Unextend(A) is the formula over the unextended signature obtained from A by performing the following replacements:

- In every pure past subformula, Before, Now and After are replaced with $\top$, $\bot$ and $\bot$, respectively.
- In every pure present subformula, Before, Now and After are replaced with $\bot$, $\top$ and $\bot$, respectively.
- In every pure future subformula, Before, Now and After are replaced with $\bot$, $\bot$ and $\top$, respectively.

We can finally define the translation algorithm.

procedure Translate(φ)
  return Translate'(Pullout(φ))
procedure Translate'(φ)
  let t be the free variable of φ (or an arbitrary variable if φ is a sentence)
  ^{0}**if** $\varphi = \bot$ **then return** $\bot$
  ^{1}**else if** $\varphi = \top$ **then return** $\top$
  ^{2}**else if** $\varphi = P(t)$ **then return** p
  ^{3}**else if** $\varphi = (t = t)$ **then return** $\top$
  ^{4}**else if** $\varphi = (t < t)$ **then return** $\bot$
  ^{5}**else if** $\varphi = \neg\alpha$ **then return** $\neg$Translate'(α)
  ^{6}**else if** $\varphi = \alpha \vee \beta$ **then return** Translate'(α) $\vee$ Translate'(β)
  ^{7}**else if** $\varphi = \alpha \wedge \beta$ **then return** Translate'(α) $\wedge$ Translate'(β)
  ^{8}**else if** $\varphi = \exists_s\alpha$
    let $A =$ Translate'(Extend(t, α))
    return Unextend(Sep($\blacklozenge A \vee A \vee \Diamond A$))
  ^{9}**else if** $\varphi = \forall_s\alpha$
    let $A =$ Translate'(Extend(t, α))
    return Unextend(Sep($\blacksquare A \wedge A \wedge \Box A$))

4.2 Algorithm Correctness

We being by showing that Pullout is correct.

Proposition 96. *Let $\varphi \in \mathcal{L}_{FOMLO}$ be pulled out. Then every subformula of φ is pulled out.*

Proof. Let ψ be a subformula of φ.

If $P(x)$ occurs inside a quantifier in ψ, then it also occurs in ψ inside a quantifier that binds x, since this quantifier is at least as deep and so is part of the subformula ψ. □

Proposition 97. *Let $\alpha, \beta \in \mathcal{L}_{FOMLO}$ be pulled out. Then:*

- *$\neg\alpha$ is pulled out*
- *$\alpha \vee \beta$ is pulled out*
- *$\alpha \wedge \beta$ is pulled out*

Proof. We show this for $\alpha \vee \beta$, the other cases are very similar.

Let $P(x)$ be a subformula of $\alpha \vee \beta$ that occurs inside a quantifier. Then $P(x) \neq \alpha \vee \beta$. It is then a subformula of either α or β. But α and β are pulled out, and so the deepest quantifier in whose scope $P(x)$ appears binds x. □

Proposition 98. *If φ is pulled out then so is any CNF or DNF of φ.*

Proof. We show this only for DNF, the argument is very similar for CNF.

Assume φ is pulled out and let $\bigvee_{i=1}^{m} \bigwedge_{j=1}^{n} \alpha_{ij}$ be a DNF of φ.

Since DNF conversion is only over the outer Boolean structure, every α_{ij} either already appears in φ or is the negation of a formula that appears in φ. In the first case α_{ij} is pulled out by proposition 96, in the second case, it is pulled out by propositions 96 and 97.

Then by proposition 97, a Boolean combination of pulled out formulas is pulled out. □

Proposition 99. *For every $\varphi \in \mathcal{L}_{FOMLO}$:*

(i) $\varphi \equiv$ *Pullout*(φ)

(ii) Pullout(φ) *is pulled out*

Proof. The recursive calls Pullout makes are always with a subformula, thus it terminates.

By induction on the subformula order.

Case 0 is trivial and cases 1,2,3 are immediate by the induction hypothesis.

Case 4:

(i) We have $\exists_x \alpha \equiv \exists_x$ Pullout$(\alpha) \equiv \exists_x \bigvee_{i=1}^{m} \alpha_i \equiv \bigvee_{i=1}^{m} \exists_x \alpha_i$, the first step by the induction hypothesis and the second because DNF conversion results in an equivalent formula. Then for each conjunctive clause $\alpha_i \equiv \gamma_i \wedge \delta_i$. Since x does not appear free in γ_i, we also have $\exists_x (\gamma_i \wedge \delta_i) \equiv \gamma_i \wedge \exists_x \delta_i$.

(ii) By the induction hypothesis, Pullout(α) is pulled out, and by proposition 98, so is $\bigvee_{i=1}^{m} \alpha_i$.

For every i, the literals in γ_i or δ_i also appear in α_i and are therefore pulled out, by proposition 96. Which means that γ_i and δ_i are also pulled out, by proposition 97,

We are now going to show that $\exists_x \delta_i$ is pulled out as well. Let $P(y)$ be a subformula of $\exists_x \delta_i$ that occurs inside the scope of a quantifier.

Clearly $P(y) \neq \exists_x \delta_i$, and δ_i is a conjunctive clause of literals, therefore $P(y)$ is a subformula of one of these conjuncts, call such conjunct ψ.

ψ must be of the form $(\neg?)\exists_z \chi$ or $(\neg?)\forall_z \chi$, as if ψ was a (possibly negated) first-order atom that has $P(y)$ as a subformula, then it would have to be $(\neg?)P(y)$, which does not have x free, and would therefore be in γ_i and not in δ_i. Therefore $P(y)$ occus in ψ inside a quantifier and so, because ψ is pulled out, as we have seen before, the deepest quantifier that contains $P(y)$ binds y. Hence $\exists_x \delta_i$ is pulled out.

We have shown that both γ_i and $\exists_x \delta_i$ are pulled out, hence by proposition 97 so is $\bigvee_{i=1}^{m} (\gamma_i \wedge \exists_x \delta_i)$.

Case 5 is very similar to case 4. $\square$

We now make precise the sense in which the extra predicates preserve the meaning of a formula.

Definition 100. Let I be an interpretation structure and t_0 a point of I.

Then $(\!|I|\!)_{t_0}$ is a structure over the extended signature, identical to I over the regular signature, and where

$$\mathrm{Before}^{(\!|I|\!)_{t_0}} = \{x \mid x <^I t_0\}$$

$$\mathrm{Now}^{(\!|I|\!)_{t_0}} = \{t_0\}$$

$$\mathrm{After}^{(\!|I|\!)_{t_0}} = \{x \mid t_0 <^I x\}$$

Proposition 101. *Let I be an interpretation structure, ρ an assignment and φ a formula. Then:*

$$I, \rho \Vdash \varphi \ \textit{ if and only if } \ (\!|I|\!)_{\rho(t)}, \rho \Vdash \textit{Extend}(t, \varphi)$$

Proof. Induction on the structure of φ.

For the cases where t does not appear free in φ, we have Extend$(t, \varphi) = \varphi$, and $(\!|I|\!)_{\rho(t)}$ coindices with I over the unextended signature, so the equivalence is immediate.

We now consider the cases where t appears free, in every case assume that $x \neq t$:

If $\varphi = P(t)$: $I, \rho \Vdash P(t)$ is by definition equivalent to $\rho(t) \in P^I = P^{(\!|I|\!)_{\rho(t)}}$, which in turn is equivalent to $(\!|I|\!)_{\rho(t)}, \rho \Vdash P(t) =$ Extend$(t, P(t))$.

If $\varphi = (t = t)$: Then φ is equivalent to $\top =$ Extend$(t, t = t)$.

If $\varphi = (t < t)$: Then φ is equivalent to $\bot =$ Extend$(t, t < t)$.

If $\varphi = (x < t)$: $I, \rho \Vdash (x < t)$ iff $\rho(x) <^I \rho(t)$ iff $\rho(x) \in \mathrm{Before}^{(\!|I|\!)_{\rho(t)}}$ iff $(\!|I|\!)_{\rho(t)}, \rho \Vdash \mathrm{Before}(x) =$ Extend$(t, x < t)$.

If $\varphi = (x = t)$ or $\varphi = (t = x)$ (we show only the first): $I, \rho \Vdash (x = t)$ iff $\rho(x) = \rho(t)$ iff $\rho(x) \in \mathrm{Now}^{(\!|I|\!)_{\rho(t)}}$ iff $(\!|I|\!)_{\rho(t)}, \rho \Vdash \mathrm{Now}(x) = \text{Extend}(t, x = t)$.

If $\varphi = (t < x)$: iff $I, \rho \Vdash (t < x)$ iff $\rho(t) <^I \rho(x)$ iff $\rho(x) \in \mathrm{After}^{(\!|I|\!)_{\rho(t)}}$ iff $(\!|I|\!)_{\rho(t)}, \rho \Vdash \mathrm{After}(x) = \text{Extend}(t, t < x)$

The Boolean operator cases are almost immediate by the induction hypothesis.

If $\varphi = \exists_x \alpha$:

	$I, \rho \Vdash \exists_x \alpha$	
iff	There is some x_0 s.t.: $I, \rho[x \mapsto x_0] \Vdash \alpha$	
iff	There is some x_0 s.t.: $(\!|I|\!)_{\rho(t)}, \rho[x \mapsto x_0] \Vdash \text{Extend}(t, \alpha)$	(induction hypothesis)
iff	$(\!|I|\!)_{\rho(t)}, \rho \Vdash \exists_x \text{Extend}(t, \alpha) = \text{Extend}(t, \exists_x \alpha)$	

The universal quantifier case is very similar to the existential one. □

The following two propositions are necessary for cases 8 and 9 of the translation algorithm.

Proposition 102. *For every interpretation I, point t_0 of I, and formula $A \in \mathcal{L}_{TL}$, the following are equivalent:*

(i) There is some $x_0 \in I$ such that $I, x_0 \Vdash A$

(ii) $I, t_0 \Vdash \blacklozenge A \lor A \lor \lozenge A$

Proof. Since we are working over linear orders, $x_0 < t_0$ or $x_0 = t_0$ or $x_0 > t_0$. These cases are equivalent, respectively, to $I, t_0 \Vdash \blacklozenge A$, $I, t_0 \Vdash A$ and $I, t_0 \Vdash \lozenge A$. □

Proposition 103. *For every interpretation I, point t_0 of I, and formula $A \in \mathcal{L}_{TL}$, the following are equivalent:*

(i) For all $x_0 \in I$: $I, x_0 \Vdash A$

(ii) $I, t_0 \Vdash \blacksquare A \land A \land \square A$

Proof. Similar to proposition 102. □

Now we move on to showing the correctness of Unextend. The following four propositions just make precise the fact that pure formulas are indeed semantically pure. That is, a pure past formula only talks about the past, and so on.

Proposition 104. *Let A be a pure present formula, I and J be interpretation structures, and t_0 a point of both such that*

$$t_0 \in P^I \iff t_0 \in P^J \qquad \textit{(for all } P \in \textit{Pred)}$$

Then:

$$I, t_0 \Vdash A \ \textit{ if and only if } \ J, t_0 \Vdash A$$

Proof. Induction on the structure of pure present formulas. □

In the next proposition, the orders do not have to match after t_0, but these stronger conditions are sufficient for our purposes. A similar thing is true for the two following propositions.

Proposition 105 (The future is irrelevant for a non-future formula)**.** *Let A be a non-future formula, I and J be interpretation structures, and t_0 a point of both, where*

$$\mathrm{Domain}(I) = \mathrm{Domain}(J)$$

$$<^I = <^J$$

$$\textit{For all } s_0 \leq t_0\textit{:} \;\; s_0 \in P^I \iff s_0 \in P^J \qquad \textit{(for all } P \in \textit{Pred)}$$

Then:

$$\textit{For all } s_0 \leq t_0\textit{:} \;\; I, s_0 \Vdash A \;\; \textit{if and only if} \;\; J, s_0 \Vdash A$$

Proof. By induction on the structure of the non-future formula A:

If $A = \bot$ or $A = \top$: Trivial.

If $A = p$: $I, s_0 \Vdash p$ iff $s_0 \in P^I = P^J$ iff $J, s_0 \Vdash p$.

If A is a Boolean operator: Straightforward use of the induction hypothesis.

If $A = B\mathcal{S}C$: $I, s_0 \Vdash B\mathcal{S}C$ if and only if there is some $s_1 \in \mathrm{Domain}(I)$ such that $s_1 <^I s_0$, $I, s_1 \Vdash C$ and all for all $r_0 \in \mathrm{Domain}(I)$, if $s_1 <^I r_0 <^I s_0$ then $I, r_0 \Vdash B$.

The domains and interpreation of the order of I and J are identical, so we can replace $\mathrm{Domain}(I)$ with $\mathrm{Domain}(J)$ and $<^I$ with $<^J$ in the previous statement to obtain something equivalent.

Since A is non-future, so are B and C, and the points s_1 and all r_0 are before s_0, and so also before t_0. Therefore we can use the induction hypothesis to replace $I, s_1 \Vdash C$ with $J, s_1 \Vdash C$ and $I, r_0 \Vdash B$ with $J, r_0 \Vdash B$. After doing this, we indeed obtain the definition of $J, s_0 \Vdash A$. □

Proposition 106 (Only the past is relevant for a pure past formula)**.** *Let A be a pure past formula, I and J be interpretation structures, and t_0 a point of both, where*

$$\mathrm{Domain}(I) = \mathrm{Domain}(J)$$

$$<^I = <^J$$

$$\textit{For all } s_0 < t_0\textit{:} \;\; s_0 \in P^I \iff s_0 \in P^J \qquad \textit{(for all } P \in \textit{Pred)}$$

Then:

$$\textit{For all } s_0 < t_0\textit{:} \;\; I, s_0 \Vdash A \;\; \textit{if and only if} \;\; J, s_0 \Vdash A$$

Proof. The proof is by induction on the structure of A. It is quite similar to the proof of proposition 105, except in the $\mathcal{S}$ case the B and C are not necessarily pure past.

Assume then that we are in the same situation as the $\mathcal{S}$ case of the proof of 105, with $I, s_0 \Vdash B\mathcal{S}C$. We know that all predicates match between the structures at all points strictly before t_0 and that $s_0 < t_0$,

therefore all predicates match at and before s_0, and we can use proposition 105 itself (with $t_0 := s_0$) instead of the induction hypothesis. $\square$

We are in fact only going to make use of proposition 105 in the proof of proposition 106, so we omit the dual of proposition of 105 and show only the dual of proposition 106.

Proposition 107 (Only the future is relevant for a pure future formula)**.** *Let A be a pure future formula, I and J be interpretation structures, and t_0 a point of both.*
Where:

$$\text{Domain}(I) = \text{Domain}(J)$$

$$<^I = <^J$$

$$\textit{For all } s_0 > t_0\textit{:}\ \ s_0 \in P^I \iff s_0 \in P^J \qquad \textit{(for all } P \in \textit{Pred)}$$

Then:

$$\textit{For all } s_0 > t_0\textit{:}\ \ I, s_0 \Vdash A \ \ \textit{if and only if}\ \ J, s_0 \Vdash A$$

Proof. We have:

- $\text{Dual}(A)$ is pure past
- $\text{Domain}(I^{\text{op}}) = \text{Domain}(I) = \text{Domain}(J) = \text{Domain}(J^{\text{op}})$
- $<^{I^{\text{op}}} = >^I = >^J = <^{J^{\text{op}}}$
- For all $s_0 <^{I^{\text{op}}} t_0$: $s_0 >^I t_0$ and so $s_0 \in P^{I^{\text{op}}} = P^I \iff s_0 \in P^J = P^{J^{\text{op}}}$ (for all $P \in$ Pred)

We can therefore use propositions 17 and 106 and conclude $I, s_0 \Vdash A$ iff $I^{\text{op}}, s_0 \Vdash \text{Dual}(A)$ iff $J^{\text{op}}, s_0 \Vdash \text{Dual}(A)$ iff $J, s_0 \Vdash A$. $\square$

Proposition 108. *For every interpretation I, point t_0 of I, and separated temporal formula A over an extended signature:*

$$(\!|I|\!)_{t_0}, t_0 \Vdash A \ \ \textit{if and only if}\ \ I, t_0 \Vdash \textit{Unextend}(A)$$

Proof. We show this for a simple pure formula. The full result can be obtained by induction over the Boolean structure. The proof is by applying propositions 106, 104 and 107. We show only the pure past case.

Consider the interpretation structure (over the extended signature) J that is identical to $(\!|I|\!)_{t_0}$ except for the interpretations of Before, Now and After, which are:

$$\text{Before}^J = \text{Domain}(J)$$

$$\text{Now}^J = \emptyset$$

$$\text{After}^J = \emptyset$$

At all points before t_0, these coincide with the interpretations given by $(\!|I|\!)_{t_0}$, therefore proposition 106 gives us $(\!|I|\!)_{t_0}, t_0 \Vdash A$ is equivalent to $J, t_0 \Vdash A$.

But, in J, Before is always true, while Now and After are always false. So they can be replaced by $\top$, $\bot$ and $\bot$, respectively, which is precisely what Unextend does. Hence $J, t_0 \Vdash A$ if and only if $J, t_0 \Vdash$ Unextend(A).

Remember that Unextend(A) is over the unextended signature, and $(\!|I|\!)_{t_0}$ and I are identical except for their interpretation of the extended predicates, the same being true of $(\!|I|\!)_{t_0}$ and J. Hence I and J are also identical over the unextended signature, and so $J, t_0 \Vdash$ Unextend(A) iff $I, t_0 \Vdash$ Unextend(A). □

Now follows the well-founded relation we will be using in the proof of correctness of the translation algorithm.

Definition 109. We define a partial order $<$ on FOMLO formulas as the transitive closure of the union of the following partial orders:

(i) α is smaller than β if it has less quantifier depth

(ii) α is smaller than β if it is a proper subformula of β

Proposition 110. *The order defined in 109 is well-founded.*

Proof. First notice that (i) and (ii) are both well-founded.

A subformula never has increased quantifier depth, so (ii) extends (i) and their union is well-founded, by proposition 71. It's also known that the transitive closure of a well-founded relation is well-founded. □

We can finally prove the correctness of translation.

Proposition 111. *For every pulled out FOMLO formula $\varphi(t)$ with at most one free variable, we have the following:*

(i) Any Translate' calls that Translate'(φ) reduces to are with a smaller argument (i.e. Translate'(φ) terminates).

(ii) For all interpretations I and points t_0 of I:

$$I, [t \mapsto t_0] \Vdash_{\textit{FOMLO}} \varphi \quad \textit{if and only if} \quad I, t_0 \Vdash_{\textit{TL}} \textit{Translate'}(\varphi)$$

Proof. We proceed by induction on the well-founded partial order defined in 109.

Notice that Translate' exhaustively matches on formulas, so let us examine each case.

Cases 0,1,2,3,4: Trivial.

Cases 5,6,7:

(i) All calls are with proper subformulas.

(ii) Straightforward use of the induction hypothesis.

Case 8:

(i) Any occurence of $P(t)$ (for every $P \in$ Pred) in φ must be inside the scope of the $\exists_x$, therefore, since φ is pulled out, this $P(t)$ must occur inside a quantifier that binds t, and so this t does not occur free. Moreover, Extend removes all free occurrences of t in a binary predicate. Therefore Extend(t, α) has no free occurrences of t. It then has at most one free variable (s) and the quantifier depth is one fewer.

(ii)

	$I, [t \mapsto t_0] \Vdash \exists_s \alpha(t, s)$	
iff	there is some s_0 s.t.: $I, [t \mapsto t_0, s \mapsto s_0] \Vdash \alpha(t, s)$	
iff	there is some s_0 s.t.: $(\!|I|\!)_{t_0}, [t \mapsto t_0, s \mapsto s_0] \Vdash$ Extend$(t, \alpha)(s)$	(proposition 101)
iff	there is some s_0 s.t.: $(\!|I|\!)_{t_0}, s_0 \Vdash_{\mathsf{TL}} A$	(induction hypothesis)
iff	$(\!|I|\!)_{t_0}, t_0 \Vdash_{\mathsf{TL}} \blacklozenge A \vee A \vee \Diamond A$	(proposition 102)
iff	$(\!|I|\!)_{t_0}, t_0 \Vdash_{\mathsf{TL}}$ Sep$(\blacklozenge A \vee A \vee \Diamond A)$	(theorem 92)
iff	$I, t_0 \Vdash_{\mathsf{TL}}$ Unextend(Sep$(\blacklozenge A \vee A \vee \Diamond A)$)	(proposition 108)

Case 9:

(i) Similar to case 8.

(ii)

	$I, [t \mapsto t_0] \Vdash \forall_s \alpha(t, s)$	
iff	for all s_0: $I, [t \mapsto t_0, s \mapsto s_0] \Vdash \alpha(t, s)$	
iff	for all s_0: $(\!|I|\!)_{t_0}, [t \mapsto t_0, s \mapsto s_0] \Vdash$ Extend$(t, \alpha)(s)$	(proposition 101)
iff	for all s_0: $(\!|I|\!)_{t_0}, s_0 \Vdash_{\mathsf{TL}} A$	(induction hypothesis)
iff	$(\!|I|\!)_{t_0}, t_0 \Vdash_{\mathsf{TL}} \blacksquare A \wedge A \wedge \Box A$	(proposition 103)
iff	$(\!|I|\!)_{t_0}, t_0 \Vdash_{\mathsf{TL}}$ Sep$(\blacksquare A \wedge A \wedge \Box A)$	(theorem 92)
iff	$I, t_0 \Vdash_{\mathsf{TL}}$ Unextend(Sep$(\blacksquare A \wedge A \wedge \Box A)$)	(proposition 108)

□

Theorem 112. *For every $\varphi \in \mathcal{L}_{FOMLO}$ with at most one free variable, interpretation structure I and point t_0 of I:*

$$I, [t \mapsto t_0] \Vdash_{FOMLO} \varphi \text{ if and only if } I, t_0 \Vdash_{TL} \textit{Translate}(\varphi)$$

Proof.

$$I, [t \mapsto t_0] \Vdash_{\textsf{FOMLO}} \varphi$$

iff $I, [t \mapsto t_0] \Vdash_{\textsf{FOMLO}} \text{Pullout}(\varphi)$ (proposition 99)

iff $I, t_0 \Vdash_{\textsf{TL}} \text{Translate'}(\text{Pullout}(\varphi)) = \text{Translate}(\varphi)$ (proposition 111)

□

4.3 Extensions

4.3.1 LTL

This algorithm can also be adapted to translate FOMLO into LTL. But then we require that the order have a minimum element, and the equivalence only holds at that minimum element. This is quite natural as LTL cannot talk about the past since it has no past operators.

This can be done by performing the translation as before and then simply replacing any past formulas with $\bot$.

procedure ToLTL(X)

^{0}**if** X is non-past **then return** X

^{1}**else if** $X = \neg A$ **then return** $\neg$ToLTL(A)

^{2}**else if** $X = A \vee B$ **then return** ToLTL$(A) \vee$ ToLTL(B)

^{3}**else if** $X = A \wedge B$ **then return** ToLTL$(A) \wedge$ ToLTL(B)

^{4}**else if** $X = A\mathcal{S}B$ **then return** $\bot$

procedure TranslateLTL(φ)

return ToLTL(Translate(φ))

Proposition 113. *For every FOMLO formula φ, Translate(φ) is separated.*

Proof. We have to show that Translate' always produces a separated formula. This can be shown by induction on the same order used to show the correctness of Translate', defined in 109.

Cases 0,1,2,3,4 are immediate. Cases 5,6,7 are a straightforward use of the induction hypothesis. For cases 8 and 9, the result is obtained by calling Unextend, which preserves separation, on the output of Sep, which produces a separated formula by theorem 92. □

Proposition 114. *Let $X \in \mathcal{L}_{TL}$ be separated and I an interpretation structure with t_0 its minimum point. Then:*

$$I, t_0 \Vdash_{TL} X \text{ if and only if } I, t_0 \Vdash_{LTL} \textit{ToLTL}(X)$$

Proof. First notice that ToLTL matches exhaustively on X. $\bot$, $\top$, p and $A\mathcal{U}B$ are included in case 0, this last one because $A\mathcal{U}B$ is separated (by assumption) and so is non-past.

By induction on the Boolean structure, we can see that ToLTL terminates.

And again by induction on the Boolean structure we can show that X and ToLTL(X) agree at t_0, noticing that as t_0 is the minimum point, $A\mathcal{S}B$ can never be true there. □

Proposition 115. *For every $\varphi(t) \in \mathcal{L}_{FOMLO}$ with at most one free variable and all interpretations I with a minimum point t_0:*

$$I, [t \mapsto t_0] \Vdash_{FOMLO} \varphi \text{ if and only if } I, t_0 \Vdash_{LTL} \textit{TranslateLTL}(\varphi)$$

Proof.

$$
\begin{aligned}
& I, [t \mapsto t_0] \Vdash_{\mathsf{FOMLO}} \varphi \\
\text{iff} \quad & I, t_0 \Vdash_{\mathsf{TL}} \mathsf{Translate}(\varphi) && \text{(theorem 112)} \\
\text{iff} \quad & I, t_0 \Vdash_{\mathsf{LTL}} \mathsf{ToLTL}(\mathsf{Translate}(\varphi)) = \mathsf{TranslateLTL}(\varphi) && \text{(proposition 114)}
\end{aligned}
$$

□

4.3.2 Expressively Complete Temporal Logics

We have described a translation algorithm from FOMLO to the particular temporal logic we have been calling TL. But this translation algorithm in fact only requires that the temporal logic in question be separable and able to express ⧫ and ◊, it makes no direct use of $\mathcal{S}$ and $\mathcal{U}$.

We actually do not even need separation in the sense described in this text, called *syntactic separation*. A weaker notion of separation, the one described by propositions 106, 104, and 107, is all that is required. Additionally, the translation algorithm also makes no use of the fact that the linear orders are complete or discrete. This means that the very same algorithm can be used to translate from FOMLO to other temporal logics, over any class of linear orders, provided they meet these conditions.

We now consider general temporal logics, and define one as being the usual propositional logic extended with any number of operators. And we say that such a logic can express ⧫ (◊) if for every formula A in that logic there is a (computable) formula B such that ⧫A (◊A) $\equiv B$.

Definition 116. We say that a formula A, in any temporal logic, is *predicatively purely past* if it meets the conditions of proposition 106. That is, a formula A is predicatively purely past if for all interpretation structures I,J and point t_0 of both that meet the three conditions of proposition 106, the conclusion of proposition 106 is true for A.

We define *predicatively purely present* and *predicatively purely future* similarly, using propositions 104 and 107, respectively.

Definition 117. A temporal logic formula is *predicatively separated* if it a Boolean combination of predicatively pure formulas.

Proposition 118. *Let L be a temporal logic able to express ⧫ and ◊ and Sep an algorithm that predicatively separates L over a class of linear orders $\mathcal{C}$. Let I be an interpretation structure whose interpretation of $<$ is in $\mathcal{C}$ (not necessarily complete or discrete). Then, for every $\varphi \in \mathcal{L}_{\mathit{FOMLO}}$ and point t_0 of I:*

$$I, [t \mapsto t_0] \Vdash_{\mathit{FOMLO}} \varphi \text{ if and only if } I, t_0 \Vdash_L \mathit{Translate}(\varphi)$$

Proof. The proof is the same as the proof of theorem 112, noticing that the conditions on the logic imposed here are all that is used, and no other properties of the specific TL we have been using are required. □

Chapter 5

Conclusions

We have written fairly simple algorithms to perform separation of TL and translation from FOMLO to TL over complete and discrete time. Since the actual translation algorithm is quite agnostic about the temporal logic used, it can also be used to translate to any expressively complete logic with computable separation. It is also possible to eliminate the discreteness requirement by extracting an algorithm from the proof of separation for complete time found in [8], perhaps even write a simple algorithm to perform this separation.

We have not talked about the complexity of these algorithms so far, but it is in fact known that the translation algorithm must have non-elementary complexity. As explained in [9] and [10], it is known that there is a non-elementary succintness gap between FOMLO and TL, that is, there are FOMLO formulas whose smallest equivalent TL formula is non-elementarily larger. This means that a translation algorithm must also have non-elementary complexity. Another way to see this is that the decidability of FOMLO even over the naturals is known to be non-elementary (see [11]), while the decidability of TL is in PSPACE (see [12]). This implies that a translation from FOMLO to TL must be non-elementary as well, as otherwise we would be able to decide FOMLO with an elementary algorithm by translating to TL and back.

Although the complexity of translation is non-elementary, in empirical testing using large amounts (tens of thousands) of randomly generated formulas of the kind of size, number of quantifiers and quantifier depth at the upper limit of that would be written by a human in practice, the speed of execution seems good enough for use in practice. Using a conventional personal computer the vast majority of formulas are translated in around a millisecond, some rare ones take a few minutes, and in very rare cases one finds a formula whose translation takes up high amounts of space and time.

Predicative separation as we defined before is sufficient for this translation algorithm to work, but the more common and more intuitive idea is the slightly stronger concept of *semantic separation*, which does not allow a pure past formula to look into the future at all, even if it does so without using any predicates (and similarly for pure future formulas). As far as we know, the worst-case complexity of an algorithm that performs either syntactic or semantic separation for any linear temporal logic is still not known, although it is suspected to be non-elementary. A possible way to investigate this might be to try

to construct a translation algorithm that would be elementary if separation was as well, thus proving that separation must be non-elementary. Perhaps by performing the elimination of binary predicates over the whole formula as a first pass and then using separation only once. The author tried the naive approach to this and was unsuccessful.

Bibliography

[1] A. N. Prior. *Time and modality*. John Locke Lecture, 2003.

[2] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, SFCS '77, pages 46–57, Washington, DC, USA, 1977. IEEE Computer Society. doi: 10.1109/SFCS.1977.32. URL http://dx.doi.org/10.1109/SFCS.1977.32.

[3] D. Gabbay, A. Pnueli, S. Shelah, and J. Stavi. On the temporal analysis of fairness. In *Proceedings of the 7th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '80, pages 163–173, New York, NY, USA, 1980. ACM. ISBN 0-89791-011-7. doi: 10.1145/567446.567462. URL http://doi.acm.org/10.1145/567446.567462.

[4] H. Kamp. *Tense logic and the theory of linear orders*. PhD thesis, University of California, Los Angeles, 1968.

[5] A. Rabinovich. A proof of kamp's theorem. *Logical Methods in Computer Science*, 10(1), 2014. doi: 10.2168/LMCS-10(1:14)2014. URL https://doi.org/10.2168/LMCS-10(1:14)2014.

[6] N. Markey. Temporal Logic with Past is Exponentially More Succinct. *EATCS Bulletin*, 79:122–128, 2003. URL https://hal.archives-ouvertes.fr/hal-01194627.

[7] D. Gabbay. *The declarative past and imperative future*, pages 409–448. Springer Berlin Heidelberg, Berlin, Heidelberg, 1989. ISBN 978-3-540-46811-0. doi: 10.1007/3-540-51803-7_36. URL http://dx.doi.org/10.1007/3-540-51803-7_36.

[8] D. M. Gabbay, I. Hodkinson, and M. Reynolds. *Temporal Logic (Vol. 1): Mathematical Foundations and Computational Aspects*. Oxford University Press, Inc., New York, NY, USA, 1994. ISBN 0-19-853769-7.

[9] I. M. Hodkinson and M. Reynolds. Separation-past, present, and future. In *We Will Show Them!(2)*, pages 117–142, 2005.

[10] K. Etessami and T. Wilke. An until hierarchy and other applications of an ehrenfeucht-fraïssé game for temporal logic. *Inf. Comput.*, 160(1):88–108, July 2000. ISSN 0890-5401. doi: 10.1006/inco.1999.2846. URL http://dx.doi.org/10.1006/inco.1999.2846.

[11] L. J. Stockmeyer. *The complexity of decision problems in automata theory and logic.* PhD thesis, Massachusetts Institute of Technology, 1974.

[12] A. P. Sistla and E. M. Clarke. The complexity of propositional linear temporal logics. *J. ACM*, 32(3): 733–749, July 1985. ISSN 0004-5411. doi: 10.1145/3828.3837. URL http://doi.acm.org/10.1145/3828.3837.

[13] V. Goranko and A. Galton. Temporal logic. In E. N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, winter 2015 edition, 2015.

Appendix A

Code

A.1 Util.hs

```haskell
{-# LANGUAGE OverloadedLists, UnicodeSyntax #-}

module Util
  ( foldl1', setCartesianProduct, Inhabited (..)
  )
where

import Data.Foldable
import Data.Set (Set)
import qualified Data.List as List
import qualified Data.Set as Set

--------------------------------------------------------------------------------
foldl1' ∷ Foldable t ⇒ (a → a → a) → t a → a
foldl1' f = List.foldl1' f . toList

--------------------------------------------------------------------------------
setCartesianProduct ∷ Ord a ⇒ [Set a] → Set [a]
setCartesianProduct [] = [[]]
setCartesianProduct (s:ss) = foldMap (\l → Set.map (: l) s) (setCartesianProduct ss)

--------------------------------------------------------------------------------
class Inhabited a where
  inhabitant ∷ a

instance Inhabited [a] where
  inhabitant = []
```

A.2 BooleanCombination.hs

```haskell
{-# LANGUAGE DeriveGeneric, LambdaCase, MultiWayIf, OverloadedLists, UnicodeSyntax #-}

module BooleanCombination
  ( BC (..)
  , bot, top, (∨), (∧), (-->), (<-->), disjList, conjList, impl, biimpl
  , bcFlip
  , bcMap, bcTraverse, bcJoin
  , Literal (..), unlit, fromLiteral, complement
```

```haskell
  , bcImplies, bcSimplify
  , fromCNF, cnf, cnfWithSimplify
  , fromDNF, dnf, dnfWithSimplify
  )
where

import Control.DeepSeq
import Data.Monoid
import Data.Set (Set)
import qualified Data.Set as Set
import GHC.Generics
import Util

--------------------------------------------------------------------------------

data BC a
  = Prim a
  | Not (BC a)
  | Or (Set (BC a))
  | And (Set (BC a))
  deriving
    (Eq, Generic, Ord, Show)

instance NFData a ⇒ NFData (BC a)

bot :: Ord a ⇒ BC a
bot = Or []

top :: Ord a ⇒ BC a
top = And []

(∨) :: Ord a ⇒ BC a → BC a → BC a
(Or αs) ∨ (Or βs) = Or (αs <> βs)
(Or αs) ∨ β = Or (Set.insert β αs)
α ∨ (Or βs) = Or (Set.insert α βs)
α ∨ β = Or [α,β]

(∧) :: Ord a ⇒ BC a → BC a → BC a
(And αs) ∧ (And βs) = And (αs <> βs)
(And αs) ∧ β = And (Set.insert β αs)
α ∧ (And βs) = And (Set.insert α βs)
α ∧ β = And [α,β]

(-->) :: Ord a ⇒ BC a → BC a → BC a
a --> b = impl [a, b]

(<-->) :: Ord a ⇒ BC a → BC a → BC a
a <--> b = (a --> b) ∧ (b --> a)

disjList :: Ord a ⇒ [BC a] → BC a
disjList = Or . Set.fromList

conjList :: Ord a ⇒ [BC a] → BC a
conjList = And . Set.fromList

impl :: Ord a ⇒ [BC a] → BC a
impl [] = error "impl: empty list"
impl as = disjList $ last as : map Not (init as)
```

```haskell
biimpl :: Ord a ⇒ [BC a] → BC a
biimpl [] = error "biimpl: empty list"
biimpl as = conjList $ zipWith (<-->) as (tail as)

infixr 6 <-->
infixr 7 -->
infixl 8 ∨
infixl 8 ∧

bcFlip :: BC a → BC a
bcFlip (Not α) = α
bcFlip α = Not α

bcMap :: Ord b ⇒ (a → b) → BC a → BC b
bcMap f (Prim x) = Prim (f x)
bcMap f (Not α) = Not (bcMap f α)
bcMap f (Or αs) = Or (Set.map (bcMap f) αs)
bcMap f (And αs) = And (Set.map (bcMap f) αs)

bcTraverse :: (Applicative f, Ord b) ⇒ (a → f b) → BC a → f (BC b)
bcTraverse f (Prim x) = Prim <$> f x
bcTraverse f (Not α) = Not <$> bcTraverse f α
bcTraverse f (Or αs) = Or . Set.fromList
                     <$> traverse (bcTraverse f) (Set.toList αs)
bcTraverse f (And αs) = And . Set.fromList
                      <$> traverse (bcTraverse f) (Set.toList αs)

bcJoin :: Ord a ⇒ BC (BC a) → BC a
bcJoin (Prim α) = α
bcJoin (Not α) = Not (bcJoin α)
bcJoin (Or αs) = Or (Set.map bcJoin αs)
bcJoin (And αs) = And (Set.map bcJoin αs)

--------------------------------------------------------------------------------
instance Foldable BC where
  foldMap f (Prim x) = f x
  foldMap f (Not α) = foldMap f α
  foldMap f (Or αs) = foldMap (foldMap f) αs
  foldMap f (And αs) = foldMap (foldMap f) αs

--------------------------------------------------------------------------------
data Literal a
  = Neg a
  | Pos a
  deriving
    (Eq, Ord, Show)

unlit :: Literal a → a
unlit (Neg x) = x
unlit (Pos x) = x

fromLiteral :: Literal a → BC a
fromLiteral (Neg x) = Not (Prim x)
fromLiteral (Pos x) = Prim x

complement :: Literal a → Literal a
complement (Neg x) = Pos x
complement (Pos x) = Neg x
```

```
-------------------------------------------------------------------------------
-- | Detects some cases where x implies y
bcImplies :: Ord a ⇒ (a → a → Bool) → BC a → BC a → Bool
bcImplies primImplies x y = case (x,y) of
  (Prim a, Prim b) → a `primImplies` b
  (Not a, Not b) → bcImplies primImplies b a
  (Or as, Or bs) → flip all as (\a → flip any bs (\b → bcImplies primImplies a b))
  (Or as, b) → flip all as (\a → bcImplies primImplies a b)
  (a, Or bs) → flip any bs (\b → bcImplies primImplies a b)
  (And as, And bs) → flip all bs (\b → bcImplies primImplies (And as) b)
  (And as, b) → flip any as (\a → bcImplies primImplies a b)
  (a, And bs) → flip all bs (\b → bcImplies primImplies a b)
  (a,b) | (a == bot) → True
        | (b == top) → True
        | otherwise → False

--------------------
-- | Simplification
bcSimplify :: Ord a ⇒ (a → a → Bool) → (a → BC a) → BC a → BC a
bcSimplify primImplies primSimplify = final . fusion . recursive
  where
    recursive (Prim x) = primSimplify x
    recursive (Not α) = Not (bcSimplify primImplies primSimplify α)
    recursive (Or αs) = Or (Set.map (bcSimplify primImplies primSimplify) αs)
    recursive (And αs) = And (Set.map (bcSimplify primImplies primSimplify) αs)
    fusion = \case
      Or αs → Or $ foldMap (\case {Or βs → βs ; β → [β]}) αs
      And αs → And $ foldMap (\case {And βs → βs ; β → [β]}) αs
      α → α
    final = \case
      Not α | (α == bot) → top
            | (α == top) → bot
      Not (Not α) → α
      Or αs → let validates β | (β == top) = True
                              | otherwise = bcFlip β `Set.member` αs
              in if any validates αs
                 then top
                 else let isRemovable β =
                                flip any αs (\α → α /= β && bcImplies primImplies β α)
                              αs' = Set.filter (not . isRemovable) αs
                      in if | Set.null αs' → bot
                            | Set.size αs' == 1 → Set.elemAt 0 αs'
                            | otherwise → Or αs'
      And αs → let falsifies β | (β == bot) = True
                               | otherwise = bcFlip β `Set.member` αs
               in if any falsifies αs
                  then bot
                  else let isRemovable β =
                                 flip any αs (\α → α /= β && bcImplies primImplies α β)
                               αs' = Set.filter (not . isRemovable) αs
                       in if | Set.null αs' → top
                             | Set.size αs' == 1 → Set.elemAt 0 αs'
                             | otherwise → And αs'
      α → α

-------------------------------------------------------------------------------
fromCNF :: Ord a ⇒ Set (Set (Literal a)) → BC a
```

```haskell
fromCNF = And . Set.map (Or . Set.map fromLiteral)

fromDNF :: Ord a ⇒ Set (Set (Literal a)) → BC a
fromDNF = Or . Set.map (And . Set.map fromLiteral)

cnf :: (Ord a) ⇒ BC a → Set (Set (Literal a))
cnf = cnf_

dnf :: (Ord a) ⇒ BC a → Set (Set (Literal a))
dnf = dnf_

cnfWithSimplify :: Ord a ⇒ (a → a → Bool) → (a → BC a) → BC a → Set (Set (Literal a))
cnfWithSimplify primImplies primSimplify α = go (cnf_ α)
  where
    go cs = let cs' = cnf_ . bcSimplify primImplies primSimplify . fromCNF $ cs
            in if cs == cs'
               then cs'
               else go cs'

dnfWithSimplify :: Ord a ⇒ (a → a → Bool) → (a → BC a) → BC a → Set (Set (Literal a))
dnfWithSimplify primImplies primSimplify α = go (dnf_ α)
  where
    go cs = let cs' = dnf_ . bcSimplify primImplies primSimplify . fromDNF $ cs
            in if cs == cs'
               then cs'
               else go cs'

--------------------
cnf_ :: (Ord a) ⇒ BC a → Set (Set (Literal a))
cnf_ (Prim x) = [[Pos x]]
cnf_ (Not (Prim x)) = [[Neg x]]
cnf_ (Not (Not α)) = cnf_ α
cnf_ (Not (Or αs)) = cnf_ (And (Set.map Not αs))
cnf_ (Not (And αs)) = cnf_ (Or (Set.map Not αs))
cnf_ (Or αs) = Set.map mconcat . setCartesianProduct . map cnf_ . Set.toList $ αs
cnf_ (And αs) = foldMap cnf_ αs

dnf_ :: (Ord a) ⇒ BC a → Set (Set (Literal a))
dnf_ (Prim x) = [[Pos x]]
dnf_ (Not (Prim x)) = [[Neg x]]
dnf_ (Not (Not α)) = dnf_ α
dnf_ (Not (Or αs)) = dnf_ (And (Set.map Not αs))
dnf_ (Not (And αs)) = dnf_ (Or (Set.map Not αs))
dnf_ (Or αs) = foldMap dnf_ αs
dnf_ (And αs) = Set.map mconcat . setCartesianProduct . map dnf_ . Set.toList $ αs
```

A.3 TL.hs

```haskell
{-# LANGUAGE DeriveGeneric #-}
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE LambdaCase #-}
{-# LANGUAGE PatternSynonyms #-}
{-# LANGUAGE UnicodeSyntax #-}

module TL
  ( SimpleTL (..), TL, pattern Var, pattern S, pattern U
  , nextPast, next, eventuallyPast, eventually, foreverPast, forever
```

```haskell
  , simpleDual, dual
  , stlImplies, tlImplies, stlSimplify, tlSimplify
  , tlCNFWithSimplify , tlDNFWithSimplify
  )
where

import Control.DeepSeq
import Data.Set (Set)
import Data.String
import GHC.Generics
import BooleanCombination

--------------------------------------------------------------------------------
data SimpleTL p
  = Variable p
  | Since (TL p) (TL p)
  | Until (TL p) (TL p)
  deriving
    (Eq, Generic, Ord, Show)

instance NFData p => NFData (SimpleTL p)

type TL p = BC (SimpleTL p)

instance IsString (TL String) where
  fromString = Prim . Variable

pattern Var p = Prim (Variable p)
pattern S a b = Prim (Since a b)
pattern U a b = Prim (Until a b)

infix 9 `Since`
infix 9 `Until`
infix 9 `S`
infix 9 `U`

nextPast :: Ord p => TL p -> TL p
nextPast a = S bot a

next :: Ord p => TL p -> TL p
next a = U bot a

eventuallyPast :: Ord p => TL p -> TL p
eventuallyPast a = S top a

eventually :: Ord p => TL p -> TL p
eventually a = U top a

foreverPast :: Ord p => TL p -> TL p
foreverPast = Not . eventuallyPast . Not

forever :: Ord p => TL p -> TL p
forever = Not . eventually . Not

simpleDual :: Ord p => SimpleTL p -> SimpleTL p
simpleDual (Variable p) = Variable p
simpleDual (Since a b) = Until (dual a) (dual b)
simpleDual (Until a b) = Since (dual a) (dual b)
```

```
dual ∷ Ord p ⇒ TL p → TL p
dual = bcMap simpleDual

--------------------------------------------------------------------------------
-- These detect some cases where x implies y

stlImplies ∷ Ord p ⇒ SimpleTL p → SimpleTL p → Bool
stlImplies x y = case (x,y) of
  (Variable p, Variable p') → p == p'
  (Since a b, Since c d) → (a ∧ Not b) `tlImplies` c && b `tlImplies` d
  (Until a b, Until c d) → (a ∧ Not b) `tlImplies` c && b `tlImplies` d
  _ → False

tlImplies ∷ Ord p ⇒ TL p → TL p → Bool
tlImplies = bcImplies stlImplies

----------------------------------------
-- Simplification

stlSimplify ∷ Ord p ⇒ SimpleTL p → TL p
stlSimplify = final . recursive
  where
    recursive (Variable p) = Variable p
    recursive (Since a b) = Since (tlSimplify a) (tlSimplify b)
    recursive (Until a b) = Until (tlSimplify a) (tlSimplify b)
    final = \case
      Since a b | (b == bot) → bot
                | (b == top) → S bot top
                | (tlImplies (a ∧ Not b) b) → S bot b
                | otherwise → S a b
      Until a b | (b == bot) → bot
                | (b == top) → U bot top
                | (tlImplies (a ∧ Not b) b) → U bot b
                | otherwise → U a b
      a → Prim a

tlSimplify ∷ Ord p ⇒ TL p → TL p
tlSimplify = bcSimplify stlImplies stlSimplify

--------------------------------------------------------------------------------
tlCNFWithSimplify ∷ Ord p ⇒ TL p → Set (Set (Literal (SimpleTL p)))
tlCNFWithSimplify = cnfWithSimplify stlImplies stlSimplify

tlDNFWithSimplify ∷ Ord p ⇒ TL p → Set (Set (Literal (SimpleTL p)))
tlDNFWithSimplify = dnfWithSimplify stlImplies stlSimplify
```

A.4 FOMLO.hs

```
{-# LANGUAGE LambdaCase, OverloadedLists, PatternSynonyms, UnicodeSyntax #-}

module FOMLO
  ( SimpleFOMLO (..), FOMLO
  , pattern Pred, pattern Eq, pattern Less, pattern Exists, pattern Forall
  , simpleFreeVars, freeVars
  , sfomloImplies, fomloImplies, sfomloSimplify, fomloSimplify
  , fomloCNFWithSimplify, fomloDNFWithSimplify
```

```haskell
  )
where

import Data.Set (Set)
import qualified Data.Set as Set
import BooleanCombination

--------------------------------------------------------------------------------

data SimpleFOMLO p x
  = Predicate p x
  | Equal x x
  | LessThan x x
  | Existential x (FOMLO p x)
  | Universal x (FOMLO p x)
  deriving
    (Eq, Ord, Show)

type FOMLO p x = BC (SimpleFOMLO p x)

pattern Pred p x = Prim (Predicate p x)
pattern Eq x y = Prim (Equal x y)
pattern Less x y = Prim (LessThan x y)
pattern Exists x φ = Prim (Existential x φ)
pattern Forall x φ = Prim (Universal x φ)

simpleFreeVars :: Ord x ⇒ SimpleFOMLO p x → Set x
simpleFreeVars (Predicate _ x) = [x]
simpleFreeVars (Equal x y) = [x,y]
simpleFreeVars (LessThan x y) = [x,y]
simpleFreeVars (Existential x α) = Set.delete x (freeVars α)
simpleFreeVars (Universal x α) = Set.delete x (freeVars α)

freeVars :: Ord x ⇒ FOMLO p x → Set x
freeVars = foldMap simpleFreeVars

--------------------------------------------------------------------------------
-- These detect some cases where x implies y

sfomloImplies :: (Ord p, Ord x) ⇒ SimpleFOMLO p x → SimpleFOMLO p x → Bool
sfomloImplies φ χ = case (φ,χ) of
  (Predicate p x, Predicate p' x') → (p == p') && (x == x')
  (Equal x y, Equal x' y') → ((x == x') && (y == y')) || ((x == y') && (y == x'))
  (LessThan x y, LessThan x' y') → (x == x') && (y == y')
  -- These next two could do more by variable renaming
  (Existential x α, Existential y β) → (x == y) && fomloImplies α β
  (Universal x α, Universal y β) → (x == y) && fomloImplies α β
  _ → False

fomloImplies :: (Ord p, Ord x) ⇒ FOMLO p x → FOMLO p x → Bool
fomloImplies = bcImplies sfomloImplies

----------------------------------------
-- Simplification

sfomloSimplify :: (Ord p, Ord x) ⇒ SimpleFOMLO p x → FOMLO p x
sfomloSimplify = final . recursive
  where
    recursive (Existential x α) = Existential x (fomloSimplify α)
```

```haskell
    recursive (Universal x α) = Universal x (fomloSimplify α)
    recursive α = α
    final = \case
      Predicate p x → Pred p x
      Equal x y | (x == y) → top
                | otherwise → Eq x y
      LessThan x y | (x == y) → bot
                   | otherwise → Less x y
      Existential x α | x `Set.member` freeVars α → Exists x α
                      | otherwise → α
      Universal x α | x `Set.member` freeVars α → Forall x α
                    | otherwise → α

fomloSimplify ∷ (Ord p, Ord x) ⇒ FOMLO p x → FOMLO p x
fomloSimplify = bcSimplify sfomloImplies sfomloSimplify

--------------------------------------------------------------------------------
fomloCNFWithSimplify ∷ (Ord p, Ord x) ⇒ FOMLO p x → Set (Set (Literal (SimpleFOMLO p x)))
fomloCNFWithSimplify = cnfWithSimplify sfomloImplies sfomloSimplify

fomloDNFWithSimplify ∷ (Ord p, Ord x) ⇒ FOMLO p x → Set (Set (Literal (SimpleFOMLO p x)))
fomloDNFWithSimplify = dnfWithSimplify sfomloImplies sfomloSimplify
```

A.5 Separation.hs

```haskell
{-# LANGUAGE MultiWayIf, OverloadedLists, RecordWildCards, UnicodeSyntax #-}

module Separation
  ( sep, sepWithSimplify
  , t1, t2, t3, t4, t5, t6, t7, t8
  )
where

import Data.Foldable
import Data.List
import Data.Maybe
import Data.Monoid
import Data.Set (Set)
import Numeric.Natural
import qualified Data.Set as Set
import BooleanCombination
import TL

--------------------------------------------------------------------------------
simpleTDepth ∷ SimpleTL p → Natural
simpleTDepth (Variable _) = 0
simpleTDepth (Since a b) = succ (max (tDepth a) (tDepth b))
simpleTDepth (Until a b) = succ (max (tDepth a) (tDepth b))

tDepth ∷ TL p → Natural
tDepth a = let l = toList (bcMap simpleTDepth a)
           in if null l then 0 else maximum l

simpleIsPast0, simpleIsFut0, simpleIsFut, simpleIsSep ∷ SimpleTL p → Bool
simpleIsPast0 (Variable _) = True
simpleIsPast0 (Since a b) = isPast0 a && isPast0 b
simpleIsPast0 _ = False
```

```haskell
simpleIsFut0 (Variable _) = True
simpleIsFut0 (Until a b) = isFut0 a && isFut0 b
simpleIsFut0 _ = False
simpleIsFut (Variable _) = False
simpleIsFut (Until a b) = isFut0 a && isFut0 b
simpleIsFut _ = False
simpleIsSep (Variable _) = True
simpleIsSep (Since a b) = isPast0 a && isPast0 b
simpleIsSep (Until a b) = isFut0 a && isFut0 b

isPast0, isFut0, isSep :: TL p → Bool
isPast0 = all simpleIsPast0
isFut0 = all simpleIsFut0
isSep = all simpleIsSep

--------------------------------------------------------------------------------
data Params p
  = Params { pSimplify :: TL p → TL p
           , pCNF :: TL p → Set (Set (Literal (SimpleTL p)))
           , pDNF :: TL p → Set (Set (Literal (SimpleTL p)))
           }

sep :: Ord p ⇒ TL p → TL p
sep = sep_ (Params id cnf dnf)

sepWithSimplify :: Ord p ⇒ TL p → TL p
sepWithSimplify = sep_ (Params tlSimplify tlCNFWithSimplify tlDNFWithSimplify) . tlSimplify

--------------------
sep_ :: Ord p ⇒ Params p → TL p → TL p
sep_ params@(Params {..}) = pSimplify . bcJoin . bcMap (simpleSep params) -- cases 1,2,3

simpleSep :: Ord p ⇒ Params p → SimpleTL p → TL p
simpleSep params@(Params {..}) x
  | simpleIsSep x = Prim x -- case 0
  | otherwise = case x of
      Since a b | isSep a && isSep b → sep5 params (pCNF a) (pDNF b) -- cases 4,5
                | otherwise → let a' = sep_ params a
                                  b' = sep_ params b
                              in sep_ params (pSimplify $ S a' b') -- case 6
      Until a b → pSimplify . dual . sep_ params . dual $ Prim x -- case 7

-- | Case 5
sep5 :: Ord p
     ⇒ Params p
     → Set (Set (Literal (SimpleTL p)))
     → Set (Set (Literal (SimpleTL p)))
     → TL p
sep5 params@(Params {..}) as bs
  | Set.null bs = bot
  | Set.null as = pSimplify $ Or $ Set.map (sep4TopLeft params) bs
  | otherwise =
      pSimplify $ And (flip Set.map as (\a → Or (flip Set.map bs (\b → (sep4 params) a b))))

-- | Case 4 (Special case where the left side is ⊤)
sep4TopLeft :: Ord p
            ⇒ Params p
            → Set (Literal (SimpleTL p))
```

```haskell
         → TL p
sep4TopLeft params@(Params {..}) bs =
  let (_d_, _b_) = Set.partition (simpleIsFut . unlit) bs
  in if Set.null _d_
     then S top (And (Set.map fromLiteral bs)) -- Already separated
     else let e@(Until f g) = case4Pick [] _d_
              b' = And . Set.map fromLiteral
                 $ _b_ <> (_d_ `Set.difference` [Pos e, Neg e])
          in if | Pos e `Set.member` _d_ → sep_ params (pSimplify $ t2 top b' f g)
                | Neg e `Set.member` _d_ → sep_ params (pSimplify $ t5 top b' f g)

-- | Case 4
sep4 :: Ord p
     ⇒ Params p
     → Set (Literal (SimpleTL p))
     → Set (Literal (SimpleTL p))
     → TL p
sep4 params@(Params {..}) as bs =
  let (_c_, _a_) = Set.partition (simpleIsFut . unlit) as
      (_d_, _b_) = Set.partition (simpleIsFut . unlit) bs
  in if Set.null _c_ && Set.null _d_
     then S (Or (Set.map fromLiteral as))
            (And (Set.map fromLiteral bs)) -- Already separated
     else let e@(Until f g) = case4Pick _c_ _d_
              a' = Or . Set.map fromLiteral
                 $ _a_ <> (_c_ `Set.difference` [Pos e, Neg e])
              b' = And . Set.map fromLiteral
                 $ _b_ <> (_d_ `Set.difference` [Pos e, Neg e])
              test s = if | (Pos e `Set.member` s) → Just True
                          | (Neg e `Set.member` s) → Just False
                          | otherwise → Nothing
              transformation = case (test _c_, test _d_) of
                (Just True, Nothing) → t1
                (Nothing, Just True) → t2
                (Just True, Just True) → t3
                (Just False, Nothing) → t4
                (Nothing, Just False) → t5
                (Just True, Just False) → t6
                (Just False, Just False) → t7
                (Just False, Just True) → t8
          in sep_ params (pSimplify $ transformation a' b' f g)

-- | This picks the (F U G) in case 4
case4Pick :: Ord p
          ⇒ Set (Literal (SimpleTL p))
          → Set (Literal (SimpleTL p))
          → SimpleTL p
case4Pick _c_ _d_ =
  let sorted = sortBy (\a b → compare (simpleTDepth b)
                                      (simpleTDepth a))
             . map unlit
             $ Set.toList (_c_ <> _d_)
      candidates = takeWhile (\a → simpleTDepth a == simpleTDepth (head sorted))
                             sorted
      test c s = if | (Pos c `Set.member` s) → Just True
                    | (Neg c `Set.member` s) → Just False
                    | otherwise → Nothing
      score c = let order = [ (Just False, Just True) -- t8
```

```haskell
                              , (Just False, Just False) -- t7
                              , (Just True, Just False) -- t6
                              , (Just True, Just True) -- t3
                              , (Nothing, Just False) -- t5
                              , (Nothing, Just True) -- t2
                              , (Just False, Nothing) -- t4
                              , (Just True, Nothing) -- t1
                              ] -- The order we choose appears to affect runtime
                in fromJust $ lookup (test c _c_, test c _d_) (zip order [0..])
  in head . sortOn score $ candidates

--------------------------------------------------------------------------------
t1, t2, t3, t4, t5, t6, t7, t8 ∷ Ord p ⇒ TL p → TL p → TL p → TL p → TL p

t1 a b f g = p ∧ (p' `S` b)
  where
    n = (Not g ∧ Not b) `S` (Not a ∧ Not b)
    p' = n --> g ∨ f
    p = n --> g ∨ (f ∧ f`U`g)

t2 a b f g = hB ∨ hNA
  where
    hB = a `S` (g ∧ a ∧ (a ∧ f)`S`b)
    hNA = (a ∧ f)`S`b ∧ (g ∨ (f ∧ f`U`g))

t3 a b f g = hB ∨ hNA
  where
    n = (Not g)`S`(Not a)
    p' = n --> g ∨ f
    p = n --> g ∨ (f ∧ f`U`g)
    hB = p ∧ p'`S`(g ∧ f`S`b)
    hNA = f`S`b ∧ (g ∨ (f ∧ f`U`g))

t4 a b f g = Not (t2 (Not b) (Not a ∧ Not b) f g) ∨ eventuallyPast b

t5 a b f g = (a`S`(Not f ∧ Not g ∧ a ∧ (a ∧ Not g)`S`b))
           ∨ ((a ∧ Not g)`S`b ∧ Not g ∧ (Not f ∨ Not(f`U`g)))

t6 a b f g = (t1 a ((a ∧ Not g)`S`b ∧ Not g ∧ Not f ∧ a) f g)
           ∨ (t3 a ((a ∧ Not g)`S`b ∧ Not g ∧ Not f) f g)
           ∨ ((a ∧ Not g)`S`b ∧ Not g ∧ (Not f ∨ Not(f`U`g)))

t7 a b f g = Not (t3 (Not b) (Not a) f g) ∧ t5 top b f g

t8 a b f g = (t4 a ((a ∧ f)`S`b ∧ g ∧ a) f g)
           ∨ (t7 a ((a ∧ f)`S`b ∧ g) f g)
           ∨ ((a ∧ f)`S`b ∧ (g ∨ (f ∧ f`U`g)))
```

A.6 Translation.hs

```haskell
{-# LANGUAGE LambdaCase #-}
{-# LANGUAGE MultiWayIf #-}
{-# LANGUAGE OverloadedLists #-}
{-# LANGUAGE RankNTypes #-}
{-# LANGUAGE RecordWildCards #-}
{-# LANGUAGE UnicodeSyntax #-}
```

```haskell
module Translation
  ( TranslateError (..), Params (..)
  , translateWithParams, translate, translateWithSimplify
  , translateLTL, translateLTLWithSimplify
  )
where

import Prelude hiding (partition)
import Control.DeepSeq
import Control.Exception
import Data.Foldable
import qualified Data.List as List
import Data.Monoid
import Data.Set (Set)
import qualified Data.Set as Set
import Data.Typeable
import BooleanCombination
import FOMLO
import TL
import Separation
import Util

--------------------------------------------------------------------------------
data Params
  = Params { pSimplify :: ∀ p x. (Ord p, Ord x) ⇒ FOMLO p x → FOMLO p x
           , pCNF :: ∀ p x. (Ord p, Ord x)
                 ⇒ FOMLO p x → Set (Set (Literal (SimpleFOMLO p x)))
           , pDNF :: ∀ p x. (Ord p, Ord x)
                 ⇒ FOMLO p x → Set (Set (Literal (SimpleFOMLO p x)))
           , pSep :: ∀ p. Ord p ⇒ TL p → TL p
           }

--------------------------------------------------------------------------------
pullout :: (Ord p, Ord x) ⇒ Params → FOMLO p x → FOMLO p x
pullout params@(Params {..}) =
  pSimplify . bcJoin . bcMap (simplePullout params) -- cases 0,1,2,3

simplePullout :: (Ord p, Ord x) ⇒ Params → SimpleFOMLO p x → FOMLO p x
simplePullout params@(Params {..}) = \case
  φ@(Predicate _ _) → Prim φ -- case 0
  φ@(Equal _ _) → Prim φ -- case 0
  φ@(LessThan _ _) → Prim φ -- case 0
  φ@(Existential x α) → -- case 4
    Or $ flip Set.map (pDNF (pullout params α)) $ \conj →
      let (δs, γs) = Set.partition ((x `Set.member`) . freeVars)
                   . Set.map fromLiteral
                   $ conj
      in pSimplify $ And $ γs <> [Exists x (And δs)]
  φ@(Universal x α) → -- case 5
    And $ flip Set.map (pCNF (pullout params α)) $ \disj →
      let (δs, γs) = Set.partition ((x `Set.member`) . freeVars)
                   . Set.map fromLiteral
                   $ disj
      in pSimplify $ Or $ γs <> [Forall x (Or δs)]

--------------------------------------------------------------------------------
data Extended p
  = Before
```

```haskell
  | Now
  | After
  | Base p
  deriving
    (Eq, Ord, Show)

simpleTrivialExtend :: (Ord p, Ord x) ⇒ SimpleFOMLO p x → SimpleFOMLO (Extended p) x
simpleTrivialExtend (Predicate p x) = Predicate (Base p) x
simpleTrivialExtend (Equal x y) = Equal x y
simpleTrivialExtend (LessThan x y) = LessThan x y
simpleTrivialExtend (Existential x α) = Existential x (trivialExtend α)
simpleTrivialExtend (Universal x α) = Universal x (trivialExtend α)

trivialExtend :: (Ord p, Ord x) ⇒ FOMLO p x → FOMLO (Extended p) x
trivialExtend = bcMap simpleTrivialExtend

simpleExtend :: (Ord p, Ord x) ⇒ x → SimpleFOMLO p x → FOMLO (Extended p) x
simpleExtend _ (Predicate p x) = Pred (Base p) x
simpleExtend t (Equal x y) | (x == t) && (y == t) = top
                           | (x == t) = Pred Now y
                           | (y == t) = Pred Now x
                           | otherwise = Eq x y
simpleExtend t (LessThan x y) | (x == t) && (y == t) = bot
                              | (x == t) = Pred After y
                              | (y == t) = Pred Before x
                              | otherwise = Less x y
simpleExtend t (Existential x α) | (x == t) = Exists x (trivialExtend α)
                                 | otherwise = Exists x (extend t α)
simpleExtend t (Universal x α) | (x == t) = Forall x (trivialExtend α)
                               | otherwise = Forall x (extend t α)

extend :: (Ord p, Ord x) ⇒ x → FOMLO p x → FOMLO (Extended p) x
extend t = bcJoin . bcMap (simpleExtend t)

----------------------------------------
simpleUnextend :: (Ord p) ⇒ SimpleTL (Extended p) → TL p
simpleUnextend (Variable Before) = bot
simpleUnextend (Variable Now) = top
simpleUnextend (Variable After) = bot
simpleUnextend (Variable (Base p)) = Var p
simpleUnextend a@(Since _ _) = simpleUnextendBefore a
  where
    simpleUnextendBefore (Variable Before) = top
    simpleUnextendBefore (Variable Now) = bot
    simpleUnextendBefore (Variable After) = bot
    simpleUnextendBefore (Variable (Base p)) = Var p
    simpleUnextendBefore (Since a b) = S (unextendBefore a) (unextendBefore b)
    simpleUnextendBefore (Until _ _) = error "simpleUnextendBefore: not separated"
    unextendBefore = bcJoin . bcMap simpleUnextendBefore
simpleUnextend a@(Until _ _) = simpleUnextendAfter a
  where
    simpleUnextendAfter (Variable Before) = bot
    simpleUnextendAfter (Variable Now) = bot
    simpleUnextendAfter (Variable After) = top
    simpleUnextendAfter (Variable (Base p)) = Var p
    simpleUnextendAfter (Since _ _) = error "simpleUnextendAfter: not separated"
    simpleUnextendAfter (Until a b) = U (unextendAfter a) (unextendAfter b)
    unextendAfter = bcJoin . bcMap simpleUnextendAfter
```

```haskell
unextend :: Ord p ⇒ TL (Extended p) → TL p
unextend = bcJoin . bcMap simpleUnextend

--------------------------------------------------------------------------------
data TranslateError
  = TooManyFreeVariables
  deriving
    (Eq, Show, Typeable)

instance Exception TranslateError

findFreeVariable :: (Inhabited x, Ord p, Ord x) ⇒ FOMLO p x → x
findFreeVariable φ =
  if | n == 0 → inhabitant -- arbitrary variable
     | n == 1 → Set.elemAt 0 xs
     | otherwise → throw TooManyFreeVariables
  where
    xs = freeVars φ
    n = Set.size xs

----------------------------------------
translateWithParams :: (Inhabited x, NFData x, Ord p, Ord x) ⇒ Params → FOMLO p x → TL p
translateWithParams params@(Params {..}) φ =
  let t = findFreeVariable φ
  in deepseq t (translate_ params t . pullout params . fomloSimplify $ φ)

translate :: (Inhabited x, NFData x, Ord p, Ord x) ⇒ FOMLO p x → TL p
translate = translateWithParams $ Params id cnf dnf sep

translateWithSimplify :: (Inhabited x, NFData x, Ord p, Ord x) ⇒ FOMLO p x → TL p
translateWithSimplify = translateWithParams $ Params fomloSimplify
                                                     fomloCNFWithSimplify
                                                     fomloDNFWithSimplify
                                                     sepWithSimplify

--------------------
translate_ :: (Ord p, Ord x) ⇒ Params → x → FOMLO p x → TL p
translate_ params@(Params {..}) t =
  tlSimplify . bcJoin . bcMap (simpleTranslate_ params t) -- cases 0,1,5,6,7

simpleTranslate_ :: (Ord p, Ord x) ⇒ Params → x → SimpleFOMLO p x → TL p
simpleTranslate_ params@(Params {..}) t = \case
  Predicate p _ → Var p -- case 2
  Equal _ _ → top -- case 3
  LessThan _ _ → bot -- case 4
  φ@(Existential s α) → -- case 8
    let a = translate_ params s . pSimplify . extend t $ α
    in tlSimplify . unextend . pSep $ eventuallyPast a ∨ a ∨ eventually a
  φ@(Universal s α) → -- case 9
    let a = translate_ params s . pSimplify . extend t $ α
    in tlSimplify . unextend . pSep $ foreverPast a ∧ a ∧ forever a

--------------------------------------------------------------------------------
-- Converts a separated TL formula into LTL, equivalent at the minimum point
toLTL :: Ord p ⇒ TL p → TL p
toLTL = bcJoin . bcMap simpleToLTL
  where
```

```haskell
    simpleToLTL (Variable p) = Var p
    simpleToLTL (Since _ _) = bot
    simpleToLTL (Until a b) = U (toLTL a) (toLTL b)

translateLTL ∷ (Inhabited x, NFData x, Ord p, Ord x) ⇒ FOMLO p x → TL p
translateLTL = toLTL . translate

translateLTLWithSimplify ∷ (Inhabited x, NFData x, Ord p, Ord x) ⇒ FOMLO p x → TL p
translateLTLWithSimplify = toLTL . translateWithSimplify
```

A.7 Parse.hs

```haskell
{-# LANGUAGE FlexibleContexts, UnicodeSyntax #-}

module Parse
  ( tlP, fomloP
  , ParseError (..), parseText, parseString
  )
where

import Data.ByteString (ByteString)
import Data.Char
import Data.Functor.Identity
import Data.Text (Text)
import Text.Parsec
import qualified Data.ByteString.UTF8 as BSU8
import qualified Data.Text as Text
import BooleanCombination
import FOMLO
import TL
import Util

--------------------------------------------------------------------------------
choiceTry ∷ Stream s m t ⇒ [ParsecT s u m a] → ParsecT s u m a
choiceTry = choice . map try

parse_ ∷ Stream s Identity t ⇒ Parsec s () a → s → Either ParseError a
parse_ p = parse p ""

--------------------------------------------------------------------------------
manyTill1 ∷ Stream s m t ⇒ ParsecT s u m a → ParsecT s u m end → ParsecT s u m ([a], end)
manyTill1 p end = p >>= \x → go (x :)
  where
    go k =  (try end >>= \e → pure (k [], e))
        <|> (p >>= \x → go (k . (x :)))

many2 ∷ Stream s m t ⇒ ParsecT s u m a → ParsecT s u m [a]
many2 p = (:) <$> p <*> many1 p

list2 ∷ Stream s m Char ⇒ ParsecT s u m a → ParsecT s u m [a]
list2 p = many2 (spaces *> p)

parens ∷ Stream s m Char ⇒ ParsecT s u m a → ParsecT s u m a
parens p = spaces *> between (char '(' *> spaces) (spaces *> char ')') p

identifierP ∷ Stream s m Char ⇒ ParsecT s u m String
identifierP = many1 (satisfy (\c → isAlphaNum c || elem c ("_-" ∷ [Char])))
```

```
predicateNameP ∷ Stream s m Char ⇒ ParsecT s u m String
predicateNameP = identifierP

variableNameP ∷ Stream s m Char ⇒ ParsecT s u m String
variableNameP = identifierP

bcP ∷ (Ord a) ⇒ Stream s m Char ⇒ ParsecT s u m a → ParsecT s u m (BC a)
bcP primP = spaces *> choiceTry [ botP
                                , topP
                                , negP
                                , orP
                                , andP
                                , implP
                                , biimplP
                                , Prim <$> primP]
  where
    variadicOp names constructor = parens $ do
      choiceTry $ string <$> names
      spaces
      constructor <$> many (bcP primP)
    variadic1Op names constructor = parens $ do
      choiceTry $ string <$> names
      spaces
      constructor <$> many1 (bcP primP)
    botP = (choiceTry $ string <$> ["⊥", "bot", "Bot", "false", "False"]) *> pure bot
    topP = (choiceTry $ string <$> ["⊤", "top", "Top", "true", "True"]) *> pure top
    negP = parens $ do
      choiceTry $ string <$> ["¬", "not", "Not", "neg", "Neg"]
      spaces
      Not <$> bcP primP
    orP = variadicOp ["∨", "or", "Or"] disjList
    andP = variadicOp ["∧", "and", "And"] conjList
    implP = variadic1Op ["→", "->", "implies", "Implies"] impl
    biimplP = variadic1Op ["↔", "<->"] biimpl

--------------------
simpleTlP ∷ Stream s m Char ⇒ ParsecT s u m (TL String)
simpleTlP = spaces *> choiceTry [ variableP
                                , sinceP
                                , untilP
                                , nextPastP
                                , nextP
                                , eventuallyPastP
                                , eventuallyP
                                , foreverPastP
                                , foreverP
                                ]
  where
    variableP = Var <$> predicateNameP
    binaryOpP names constructor = parens $ do
      choiceTry $ string <$> names
      spaces
      arg0 ← tlP
      spaces
      arg1 ← tlP
      pure $ constructor arg0 arg1
    sinceP = binaryOpP ["since", "Since", "s", "S"] S
```

```
    untilP = binaryOpP ["until", "Until", "u", "U"] U
    unaryOp names constructor = parens $ do
      choiceTry $ string <$> names
      spaces
      constructor <$> tlP
    nextPastP = unaryOp ["●", "prev", "Prev"] nextPast
    nextP = unaryOp ["○", "next", "Next"] next
    eventuallyPastP = unaryOp ["♦", "eventually-past", "Eventually-Past"] eventuallyPast
    eventuallyP = unaryOp ["◇", "eventually", "Eventually"] eventually
    foreverPastP = unaryOp ["■", "forever-past", "Forever-Past"] foreverPast
    foreverP = unaryOp ["□", "forever", "Forever"] forever

tlP :: Stream s m Char ⇒ ParsecT s u m (TL String)
tlP = bcJoin <$> bcP simpleTlP

--------------------
simpleFomloP :: Stream s m Char ⇒ ParsecT s u m (FOMLO String String)
simpleFomloP = spaces *> choiceTry [ equalP
                                   , lessThanP
                                   , lessEqP
                                   , greaterThanP
                                   , greaterEqP
                                   , existentialP
                                   , universalP
                                   , predicateP
                                   ]
  where
    binaryPredP names constructor = parens $ do
      choiceTry $ string <$> names
      args ← spaces *> list2 variableNameP
      pure $ conjList (zipWith constructor args (tail args))
    equalP = binaryPredP ["="] Eq
    lessThanP = binaryPredP ["<"] Less
    lessEqP = binaryPredP ["≤", "<="] (\x y → (Less x y) ∨ (Eq x y))
    greaterThanP = binaryPredP [">"] (flip Less)
    greaterEqP = binaryPredP ["≥", ">="] (\x y → (Eq x y) ∨ (Less y x))
    quantifierP names constructor = parens $ do
      choiceTry $ string <$> names
      spaces
      (vars, body) ← manyTill1 (spaces *> variableNameP) fomloP
      pure $ ($ body) . foldl1' (.) . map constructor $ vars
    existentialP = quantifierP ["∃", "exists", "Exists"] Exists
    universalP = quantifierP ["∀", "forall", "Forall"] Forall
    predicateP = parens (Pred <$> (spaces *> predicateNameP) <*> (spaces *> variableNameP))

fomloP :: Stream s m Char ⇒ ParsecT s u m (FOMLO String String)
fomloP = bcJoin <$> bcP simpleFomloP

--------------------------------------------------------------------------------
parseText :: Parsec String () a → Text → Either ParseError a
parseText p = parse_ p . Text.unpack

parseString :: Parsec String () a → String → Either ParseError a
parseString = parse_
```

A.8 Pretty.hs

```haskell
{-# LANGUAGE LambdaCase, MultiWayIf, OverloadedStrings, UnicodeSyntax #-}

module Pretty
  ( tl, ppTL
  )
where

import Numeric.Natural
import Text.PrettyPrint
import qualified Data.Set as Set
import BooleanCombination
import TL

--------------------------------------------------------------------------------
tlDepth :: TL a → Natural
tlDepth (Var _) = 1
tlDepth (S a b) = succ (max (tlDepth a) (tlDepth b))
tlDepth (U a b) = succ (max (tlDepth a) (tlDepth b))
tlDepth (Not a) = succ (tlDepth a)
tlDepth (Or as) = if Set.null as then 1 else succ (maximum (Set.map tlDepth as))
tlDepth (And as) = if Set.null as then 1 else succ (maximum (Set.map tlDepth as))

tlVertexCount :: TL a → Natural
tlVertexCount (Var _) = 1
tlVertexCount (S a b) = succ (tlVertexCount a + tlVertexCount b)
tlVertexCount (U a b) = succ (tlVertexCount a + tlVertexCount b)
tlVertexCount (Not a) = succ (tlVertexCount a)
tlVertexCount (Or as) = succ (sum . map tlVertexCount . Set.toList $ as)
tlVertexCount (And as) = succ (sum . map tlVertexCount . Set.toList $ as)

belowIndentationThreshold :: TL a → Bool
belowIndentationThreshold a = (tlDepth a <= 3) && (tlVertexCount a <= 10)

tl :: TL String → Doc
tl = \case
  Var p → text p
  a@(S b c) | (b == bot) →
                "(● " <> nest 3 (tl c) <> ")"
            | (b == top) →
                "(♦ " <> nest 3 (tl c) <> ")"
            | belowIndentationThreshold a →
                "(S " <> tl b <+> tl c <> ")"
            | otherwise →
                "(S " <> nest 3 (tl b) $$ nest 3 (tl c) <> ")"
  a@(U b c) | (b == bot) →
                "(○ " <> nest 3 (tl c) <> ")"
            | (b == top) →
                "(◊ " <> nest 3 (tl c) <> ")"
            | belowIndentationThreshold a →
                "(U " <> tl b <+> tl c <> ")"
            | otherwise →
                "(U " <> nest 3 (tl b) $$ nest 3 (tl c) <> ")"
  Not a → "(¬ " <> nest 3 (tl a) <> ")"
  a@(Or bs) →
    let n = Set.size bs
    in if | n == 0 → "⊥"
```

```
          | n == 1 → tl (Set.elemAt 0 bs)
          | belowIndentationThreshold a →
              "(∨ " <> (hsep . map tl . Set.toList $ bs) <> ")"
          | otherwise →
              "(∨ " <> (nest 3 (vcat . map tl . Set.toList $ bs)) <> ")"
  a@(And bs) →
    let n = Set.size bs
    in if | n == 0 → "⊤"
          | n == 1 → tl (Set.elemAt 0 bs)
          | belowIndentationThreshold a →
              "(∧ " <> (hsep . map tl . Set.toList $ bs) <> ")"
          | otherwise →
              "(∧ " <> (nest 3 (vcat . map tl . Set.toList $ bs)) <> ")"

----------------------------------------
ppTL :: TL String → String
ppTL = render . tl
```

A.9 Main.hs

```
{-# LANGUAGE FlexibleContexts, MultiWayIf, UnicodeSyntax #-}

import Control.Exception
import Data.List (intersperse)
import System.Console.GetOpt
import System.Environment
import Text.Parsec
import Parse
import Pretty
import Separation
import Translation

--------------------------------------------------------------------------------
instance Exception ParseError

data Flags
  = Help
  | NoSimplification
  | Sep
  | Translate
  | TranslateLTL
  deriving
    (Eq, Show)

optionDescription :: [OptDescr Flags]
optionDescription =
  [ Option ['h'] ["help"] (NoArg Help) "Help"
  , Option ['n'] ["no-simplification"] (NoArg NoSimplification) "Disable simplification"
  , Option ['s'] ["sep"] (NoArg Sep) "Run the Sep algorithm"
  , Option ['t'] ["translate"] (NoArg Translate) "Run the Translate algorithm"
  , Option ['l'] ["translateLTL"] (NoArg TranslateLTL) "Run the TranslateLTL algorithm"
  ]

main :: IO ()
main = do
  (options, _, _) ← getOpt Permute optionDescription <$> getArgs
  let mkParser p = spaces *> choice [ eof *> pure Nothing
```

```
                                  , Just <$> ((,) <$> p <*> getInput) ]
    sepAlg = if NoSimplification `elem` options
             then sep
             else sepWithSimplify
    translateAlg = if NoSimplification `elem` options
                   then translate
                   else translateWithSimplify
    translateLTLAlg = if NoSimplification `elem` options
                      then translateLTL
                      else translateLTLWithSimplify
    loop parser alg printSeparator str = case parseString parser str of
      Left e → throwIO e
      Right Nothing → pure ()
      Right (Just (x, str')) → do
        let y = alg x
        printSeparator
        putStrLn . ppTL $ y
        loop parser alg (putStrLn "") str'
if Help `elem` options
  then let headerLines =
             [ "Reads formulas from standard input and outputs to standard output."
             , "The default is to run the Translate algorithm with simplification."
             , ""
             , "Formulas in both TL and FOMLO are denoted by s-expressions."
             , "  Available Boolean operators:"
             , "    - False. Names: ⊥, bot, Bot, false, False"
             , "    - True. Names: ⊤, top, Top, true, True"
             , "    - Negation (unary). Names: ¬, not, Not, neg, Neg"
             , "    - Disjunction (any number of arguments). Names: ∨, or, Or"
             , "    - Conjunction (any number of arguments). Names: ∧, and, And"
             , "    - Implication (at least one argument). Names: →, ->, implies, Implies"
             , "    - Bi-implication (at least one argument). Names: ↔, <->"
             , "  Available TL operators:"
             , "    - Variable. String consisting of alphanumeric characters"
             , "    - Since. Names: since, Since, s, S"
             , "    - Until. Names: until, Until, u, U"
             , "    - Previous. Names: ●, prev, Prev"
             , "    - Next. Names: ○, next, Next"
             , "    - Eventually in the past. Names: ♦, eventually-past, Eventually-Past"
             , "    - Eventually. Names: ◊, eventually, Eventually"
             , "    - Forever in the past. Names: ■, forever-past, Forever-Past"
             , "    - Forever. Names: □, forever, Forever"
             , "  Available FOMLO operators:"
             , "    - Predicate (one alphanumeric variable).\
               \ The predicate name itself is also a string of alphanumeric characters.\
               \ Example: (P x)"
             , "    - Equality (at least two arguments). Names: ="
             , "    - Less (at least two arguments). Names: <"
             , "    - Less or equal (at least two arguments). Names: ≤, <="
             , "    - Greater (at least two arguments). Names: >"
             , "    - Greater or equal (at least two arguments). Names: ≥, >="
             , "    - Existential (one or more alphanumeric variables followed by a formula).\
               \ Names: ∃, exists, Exists"
             , "    - Universal (one or more alphanumeric variables followed by a formula).\
               \ Names: ∀, forall, Forall"
             , "  Examples:"
             , "    - FOMLO: (∀ x y z (→ (∧ (< x y) (< y z)) (< x z)))\
               \ is a formula denoting transitivity"
```

```
          , "    - TL: (→ E1 (∧ (¬ E2) (Until (¬ E2) L1)))"
          , "Options:"
          ]
        header = concat $ intersperse "\n" headerLines
    in putStr $ usageInfo header optionDescription
  else getContents >>= if | Translate `elem` options →
                              loop (mkParser fomloP) translateAlg (pure ())
                          | TranslateLTL `elem` options →
                              loop (mkParser fomloP) translateLTLAlg (pure ())
                          | Sep `elem` options →
                              loop (mkParser tlP) sepAlg (pure ())
                          | otherwise →
                              loop (mkParser fomloP) translateAlg (pure ())
```