Master's thesis to obtain the degree
Master in pure and applied logic

Interpretability logics and generalized Veltman semantics in Agda

Jan Mas Rovira

Supervised by Joost J. Joosten
and Luka Mikec

Universitat de Barcelona

Facultat de Filosofia de Barcelona and
Facultat de Matemàtiques de Barcelona
November 2020

Abstract

Sufficiently strong arithmetical theories such as PA can formalize their interpretability predicate. This predicate expresses the concept of relativized interpretability between finite extensions of the theory. Interpretability logics study the provable structural behaviour of the interpretability predicate.

Interpretability logics extend the provability logic GL. As such, interpretability logics inherit a number of similarities from GL. For instance, the possibility to give relational semantics. In this thesis we focus on the study of two variations of relational semantics à la Kripke known as *ordinary Veltman semantics* and *generalized Veltman semantics*.

In the literature we find various definitions of generalized Veltman semantics. In particular, there are several conditions of *quasi-transitivity*, a property that generalized frames must satisfy. We study the interrelations between all of these conditions and discuss their adequacy.

In this thesis we compare the expressiveness of ordinary and generalized Veltman semantics. Furthermore, we give procedures that under some assumptions transform, while preserving modal theoremhood, ordinary models to generalized models and vice versa.

We study the frame conditions of various relevant interpretability principles present in the literature. Moreover, we provide novel frame conditions for the the principle R_1 and the R^n series of principles from [16] with respect to generalized Veltman semantics.

We have formalized our findings in Agda, which is a state-of-the-art proof assistant based on an intuitionistic theory which features dependent types. Apart from giving a solid base to our claims, we hope that our Agda library will provide a rallying point for researchers willing to formalize theorems in the field of interpretability logics, or at least, to encourage more research in that direction. Our work is, to our knowledge, the first attempt at formalizing interpretability logics in any proof assistant.

Resum

Teories aritmètiques suficientment fortes tals com PA poden formalitzar el seu predicat d'interpretabilitat. Aquest predicat expressa el concepte d'interpretabilitat relativitzada entre extensions finites de la teoria. Les lògiques d'interpretabilitat estudien el comportament estructural demostrable del predicat d'interpretabilitat.

Les lògiques d'interpretabilitat estenen la lògica de demostrabilitat GL. Com a tals, hereten algunes semblances de GL. Per exemple, la possibilitat de definir una semàntica relacional. En aquesta tesi ens centrem en l'estudi de dues variacions de semàntiques relacionals a l'estil Kripke que es coneixen com *semàntica ordinària Veltman* i *semàntica generalitzada Veltman*.

En la literatura trobem vàries definicions de semàntica generalitzada Veltman. En particular, hi ha diverses condicions de *quasi-transitivitat*, una propietat que els marcs generalitzats de Veltman han de satisfer. Estudiem les interrelacions entre aquestes condicions i comentem la seva pertinència.

En aquesta tesi comparem l'expressivitat de les semàntiques Veltman ordinàries i generalitzades. A més, descrivim procediments que sota certes hipòtesis transformen, preservant els teoremes modals, models ordinaris Veltman a models generalitzats Veltman i viceversa.

Estudiem les condicions de marc per diversos principis d'interpretabilitat que es troben a la literatura. A més, donem noves condicions de marc pel principi R_1 i la sèrie de principis R^n presentats en [16] respecte semàntica generalitzada Veltman.

Hem formalitzat les nostres aportacions en Agda, que és un assistent de demostració modern basat en una lògica intuïcionista amb tipus dependents. A banda de donar solidesa als nostres avenços, esperem que la nostra llibreria d'Agda pugui ser un punt de trobada pels investigadors que desitgin formalitzar teoremes en el camp de les lògiques d'interpretabilitat, o si més no, que encoratgi més investigació en aquesta direcció. La nostra llibreria és, que sapiguem, el primer intent de formalitzar lògiques d'interpretabilitat en algun assistent de la demostració.

Acknowledgments

We want to express our gratitude to Nicola Olivetti, Rineke Verbrugge and the rest of the organizing committee of Advances in Modal Logic 2020 (Helsinki) for allowing us to present there an extended abstract of this thesis. We want to extend our gratitude to all the reviewers who gave valuable feedback.

We want to express our gratitude to Rineke Verbrugge. She kindly agreed to publish a manuscript by hers, which has significant historical and mathematical value in the field of interpretability logics, as an appendix of this thesis.

I want to thank my supervisors Joost and Luka. Their incredible work ethic and vast knowledge on the field have been a source of inspiration throughout this thesis. I am ever grateful for their patience and valuable corrections.

I want to thank Esperanza Buitrago for proof-reading the thesis and providing helpful feedback.

I would like to thank all the teachers in the Master in Pure and Applied logic from the University of Barcelona, with a special thanks to Ramon Jansana. I feel privileged for having been part of this master.

I want to share the joy that I am feeling at the time of finishing this thesis with my very good friends from the master Martín and Stefano. I really hope that we can meet again somewhere in Raval rather sooner than later.

Contents

Part I.

Introduction

1. Overview of interpretability logics

This thesis studies interpretability logics. These logics were originally conceived ([19, 38]) to study the provably structural behaviour of formalized relative interpretability. In this overview we will give some context and definitions to understand what this means.

We begin by giving a short introduction on provability logics. For more information refer to [37]. The reason to start here is that interpretability logics extend provability logics. Furthermore the ecosystem of interpretability logics is similar to the one of provability logics, albeit more complex, thus it will help establish an idea of the problems that we will tackle further in the thesis.

Since Gödel's incompleteness theorems, we have been interested in studying how much arithmetical theories like Peano Arithmetic can say about themselves, more precisely, about their provability capacity. As we know, sufficiently strong arithmetical theories like PA can represent, by means of a clever syntactical encoding, their provability predicate, $\mathsf{Prov}_T(A)$, which states "formula A is provable in T". A landmark result on this predicate is Gödel's second incompleteness theorem ([13, 30]) and Löb's theorem ([21, 37]). Gödel second incompleteness theorem states that consistent and sufficiently strong theories like PA cannot proof their own consistency. Löb's theorem, states that PA proves $\mathsf{Prov}_T(\ulcorner A \urcorner) \to A$ only if PA proves A. Löb's theorem can be formalized in the language of PA thus:

$$\mathsf{PA} \vdash \mathsf{Prov}_{\mathsf{PA}}(\ulcorner \mathsf{Prov}_{\mathsf{PA}}(\ulcorner A \urcorner) \to A \urcorner) \to \mathsf{Prov}_{\mathsf{PA}}(\ulcorner A \urcorner).$$

This result sparked an interest in studying provability logic from a modal point of view which yielded the axiomatization of the propositional provability logic GL, named after Gödel and Löb. The logic GL extends propositional classical logic and its language consists of the language of propositional logic $\langle \to, \bot \rangle$ plus the modal operator $\Box$, which stands for the provability predicate. In this language Löb's theorem reads as

$$\Box(\Box A \to A) \to \Box A.$$

The above *principle* is in fact one of the axioms of GL. The importance of GL depends on the arithmetical soundness and completeness theorems that we will present shortly, but first we need to introduce the concept of *arithmetical realization*. An arithmetical realization $*$ is a map from modal formulas in the language of GL to arithmetical sentences of T such that it satisfies the following:

$$\begin{aligned} p^* &\text{ is a } T\text{-sentence;} \\ \bot^* &= (0 = 1); \\ (A \to B)^* &= A^* \to B^*; \\ (\Box A)^* &= \mathsf{Prov}_T(\ulcorner A \urcorner). \end{aligned}$$

Note that specifying how an arithmetical realization $*$ acts on propositional variables fixes the behaviour of $*$ on all modal formulas.

The arithmetical soundness theorem states that if $\mathsf{GL} \vdash A$ then $\mathsf{PA} \vdash A^*$ for any realization $*$. Solovay proved ([32]) that arithmetical completeness, which is the converse of soundness, also holds.

Provability logics have Kripke (relational) semantics. As a reminder, a Kripke model is a tuple $\langle W, R, V \rangle$ where W is a non-empty set of worlds, R is a binary relation on worlds and

$V \subseteq W \times \mathsf{Var}$ is a valuation. Then we define a forcing relation $\Vdash$ that relates worlds and modal formulas. We write $w \Vdash A$ to say that world w forces A:

1. $w \nVdash \bot$;
2. if p is a propositional variable, then $w \Vdash p$ iff: $\langle w, p\rangle \in V$;
3. if A and B are formulas, then $w \Vdash A \to B$ iff: if $w \Vdash A$ then $w \Vdash B$;
4. if A is a formula, then $w \Vdash \Box A$ iff: if for any world u such that wRu then $u \Vdash A$.

In order to state a powerful modal soundness and completeness theorem we ought to restrict the class of frames that we consider. We say that a frame $\langle W, R\rangle$ is a GL-frame if R is transitive and conversely well-founded, that is, there are no infinite ascending chains $w_0 R w_1 R \dots$. Then we have that

$$\mathsf{GL} \vdash A \Leftrightarrow (\langle W, R, V\rangle \Vdash A \text{ for any } \mathsf{GL}\text{-frame } \langle W, R\rangle \text{ and valuation } V).$$

With this we conclude the introduction to provability logics and continue with interpretability logics.

We introduce the concept of *interpretation* as a first step towards understanding interpretability logics. Given two theories V and U, an interpretation of V into U is a translation f from V-formulas to U-formulas such that if $V \vdash A$ then $U \vdash f(A)$. As such, we work with the notion of *theorem's interpretability*. Moreover, we will allow for domain relativization. In the history of mathematics we find numerous examples ([38]) of theorems that use interpretations as a cornerstone in their proofs. For instance, we have that Gödel's interpretation of ZF plus the axiom of constructibility ($V = L$) in ZF serves as a proof of relative consistency of the continuum hypothesis with respect to ZF.

Interpretability logics, as the name foretells, studies the behaviour of interpretations. There are a number of things that we need to specify. We do not study the behaviour of interpretations between two arbitrary theories. Instead, we fix a base theory T and we only consider interpretability between finite extensions of such a theory. Thus when we say that $T + A$ interprets $T + B$ we assume that A and B are sentences. In interpretability logics we only consider first order arithmetical theories which are recursively enumerable and are sequential[1]. Moreover, we are only concerned with the provable, within T, interpretability properties. In other words, we study the behaviour of the binary formalized interpretability predicate $\mathsf{Int}_T(.,.)$. Naturally, the predicate $\mathsf{Int}_T(\ulcorner A\urcorner, \ulcorner B\urcorner)$ precisely holds when we can $T + A$ interprets $T + B$. For a theory T, we denote the logic that describes the behaviour of the interpretability predicate of T as $\mathsf{IL}(T)$. The language of $\mathsf{IL}(T)$ extends the language of GL with the binary modal operator $\rhd$, thus the language becomes $\langle \bot, \to, \Box, \rhd\rangle$. In order to give a more detailed definition of $\mathsf{IL}(T)$ we first extend the definition of arithmetical realization with an extra clause for the $\rhd$ case. An arithmetical realization $*$ is a map from formulas in $\mathsf{IL}(T)$ to sentences in the language of T. Moreover, it must satisfy the following constraints:

$$\begin{aligned} p^* &\text{ is a } T\text{-sentence;}\\ \bot^* &= (0 = 1);\\ (A \to B)^* &= A^* \to B^*;\\ (\Box A)^* &= \mathsf{Prov}_T(\ulcorner A\urcorner);\\ (A \rhd B)^* &= \mathsf{Int}_T(\ulcorner A\urcorner, \ulcorner B\urcorner). \end{aligned}$$

Then we define the interpretability logic of a theory T in the following way:

$$\mathsf{IL}(T) := \{A \mid T \vdash A^* \text{ for any realization } * \}.$$

[1]A sequential theory is an arithmetical which can code and decode sequences of elements.

There is an important difference with respect to provability logics that we need to highlight here. In provability logics we have that GL is an axiomatization of the provability logics for all sufficiently strong consistent arithmetical theories. However, we do not have an equivalent Swiss army knife that works for all theories in interpretability logics. In other words, we do not have a single axiomatization of $\mathsf{IL}(T)$. Instead, we define different logics for various classes of theories. There is a basic logic which we call IL that serves as the base for all these variants. We define these variants as extensions of IL by adding axiom schemas to it as we will see later in this overview. The logic IL extends the GL logic with five new axiom schemas. We will precisely define the logic IL and comment on the significance of the new axiom schemas in Chapter 5.

The logic IL has two relational semantics, which are the main topic of this thesis, known as *ordinary Veltman semantics* (introduced by Frank Veltman in [17]) and *generalized Veltman semantics* (introduced by Rineke Verbrugge in [36]). Ordinary Veltman semantics are defined in much the same spirit as Kripke semantics for GL. However, ordinary Veltman frames have an additional family of relations indexed by each of the worlds[2]: $\{S_w : w \in W\}$. Each of the S_w is a binary relation which relates two worlds. The family of relations S is used to model the behaviour of the binary modal operator $\rhd$.

In generalized Veltman semantics each of the S_w relations, instead of relating two individual worlds, relates an individual world to a set of worlds. This type of definition seems to be inspired by the neighborhood semantics that exist for other modal logics with unary modal operators. Neighborhood semantics have not been studied yet for interpretability logics. Both kinds of Veltman semantics are suitable for IL as we have soundness (which we present in this thesis) and completeness proofs ([17]).

As we have already mentioned, IL is just the logic that we use as a base in interpretability logics, thus, each time we extend it with an axiom scheme (or principle, as we call them) we will need to find new semantics for it. Given a collection of principles, we get the semantics for the extension of IL with these principles by finding their frame condition. More precisely, a *frame condition* for a principle X is a first (or higher) order formula $\mathcal{C}_{\mathsf{X}}$ such that for every Veltman frame F we have

$$F \vDash \mathcal{C}_{\mathsf{X}} \Leftrightarrow \langle F, V\rangle \Vdash \mathsf{X} \text{ for any valuation } V \text{ and instance of } \mathsf{X}.$$

To give an example, let us put our focus on the interpretability logic of PA. It happens ([5, 31]) that the axiomatization of $\mathsf{IL}(\mathsf{PA})$ is given by adding the so called principle M to the logic IL. We denote this new logic as ILM. The principle M is defined as follows:

$$\mathsf{M} := (A \rhd B) \to ((A \wedge \Box C) \rhd (B \wedge \Box C)).$$

Then, consider the following condition:

$$C_{\mathsf{M}} := \text{if } xS_w y \text{ and } yRz \text{ then } xRz.$$

Now, if we consider the class of Veltman frames that satisfy condition C_{M}, we have soundness and completeness of ILM and thus the presented condition is a suitable frame condition for M.There are plenty of principles similar to M in the literature. In this thesis we list a number of interpretability principles, including M, and we present their frame condition for both ordinary and generalized Veltman semantics. The language of frame conditions is usually taken to be the language of first or higher order formulas, where we have the R and S relations as primitive symbols.

Some principles, like M, are useful because they allow us to axiomatize the interpretability logic for certain classes of theories. However, there are other principles which are interesting because they are arithmetically valid in a large number of theories. These principles play a

[2]Alternatively, we may refer to S as a ternary relation, where the first component corresponds to the index of the family.

crucial role in the search of an axiomatization for the theory $\mathsf{IL}(\text{All})$. The theory $\mathsf{IL}(\text{All})$ is defined to be the intersection of the interpretability logics of all reasonable arithmetical theories ([39, 40]). Finding an axiomatization of this logic, as already hinted, remains an open problem, however a lot of progress has been made in the form of finding lower bounds for this logic ([16, 18, 20]). In this thesis we study the frame conditions for two series of principles: R^{n} and R_{n}, which appear in the current best known lower bound of $\mathsf{IL}(\text{All})$.

2. Original contributions

This thesis includes the following original contributions:

1. We have found a generalized frame condition for R_1 (in collaboration with Mikec). See Theorem 18.2.

2. We have found a generalized frame condition for R^n. See Theorem 20.2.

3. We performed a detailed analysis of the quasi-transitivity conditions available in the literature for generalized semantics. See Chapter 7.

4. We analyzed how a monotonicity condition for generalized semantics that is often assumed or taken as part of the definition for generalized frames interacts with the quasi-transitivity conditions present in the literature. Furthermore, we justify why in a sense assuming such condition if it is not required by the definition is harmless. See Chapter 8.

5. We discovered a proof in a published article which needs to be repaired. See Chapter 12.

6. We compare the expressiveness of ordinary and generalized Veltman models. As part of this, we worked out all the details of a proof in an unpublished manuscript by Verbrugge ([36]). We attach this manuscript in Appendix D.

7. We present the implementation of a verified language to write Hilbert style proofs (for the logic IL) in Agda with paper-like syntax. See Chapter 31.

8. We give details for an embedding of propositional intuitionistic logic into Martin Löf's logical framework. This result is expected. Our contribution has been to provide detailed definitions and proofs, which we were unable to find somewhere else. The detailed proof that $n + 0 = n$ in Martin Löf's logical framework is also original. See Section 23.4.

9. During the development of this thesis we have coauthored two publications:
   - *An overview of Generalised Veltman Semantics* ([19]). In this publication we give an up-to-date overview of interpretability logics with a focus on generalized semantics.
   - *Generalised Veltman Semantics in Agda* ([25]). In this publication we focus on the frame conditions for generalized Semantics for the principle R_1 and the series R^n. We also comment on the Agda formalization. This extended abstract was presented in August in the peer-reviewed AiML2020 conference in Helsinki (online).

10. We have implemented an Agda library for interpretability logics which includes every theorem and proof marked with that is presented in this thesis. It is worth pointing out that we started from scratch since there was no previous published work of interpretability logics in Agda, or in any other proof assistant. The library comprises ~5000 lines of code and is freely available online:

    gitlab.com/janmasrovira/interpretability-logics

    We also provide the Agda code in the annex of this thesis. See Appendix B.

    Note: Throughout this thesis we present the proofs (in English) of all the lemmas and theorems listed. Often we skip details or we only present part of the proofs. It is important that we emphasize that when we mark a proof with the symbol it means that the whole proof (not only the commented part) has been formalized down to the definitions in Agda.

11. We have reimplemented a small portion of the Agda library in the Coq proof assistant ([33]). This portion includes the definitions of the theorems and proofs marked with . Namely this subset is composed of the definition of ordinary Veltman semantics, the axiomatization of the logic IL and its proof of soundness. The library comprises ~500 lines of code and is available online:

    gitlab.com/janmasrovira/coq-interpretability-logics

    We also provide the Coq code in the annex of this thesis. See Appendix C.

3. Notation

In this chapter we will fix notation that is used throughout the thesis. Of course, it makes most sense to skip the section at first and come back to it while reading the remainder of the thesis.

3.1. Text

Here we list a number of notational conventions that we use in the text of the thesis:

1. If R and R' are binary relations, then $wRuR'v$ means wRu and $uR'v$. For instance $wRuS_xv$ means wRu and uS_xv. Another example: $wRu \Vdash A$ means wRu and $u \Vdash A$.
2. $Y \Vdash A$ iff for all $y \in Y$ we have $y \Vdash A$.
3. $Y \nVdash A$ iff there is some $y \in Y$ such that $y \nVdash A$;
4. $[\![A]\!] := \{w : w \Vdash A\}$.
5. When we write a dot after the quantification of some variables, the scope of the variables extends to the rightmost part of what follows, of course, without escaping parentheses. Hence the formula $\forall x \exists y.Pxy \wedge \forall z.Pyz$ is equivalent to $\forall x \exists y(Pxy \wedge \forall z(Pyz))$.
6. We use commas to denote conjunction, so $\forall x.A(x), B(x)$ should be read as for all x we have $A(x)$ and $B(x)$.
7. If X is a principle (or axiom schema), we denote by ILX the logic which consists in adding the axiom schema X to the logic IL. We will write ILXY to denote that we add principles X and Y to the IL, and so on.
8. In lambda calculus and Martin Löf's logical framework (Part IV), we write $e[x \mapsto a]$ to say that all free occurrences of x in e are replaced by a. We write $e[x_1 \mapsto a_1, ..., x_\mathrm{n} \mapsto a_\mathrm{n}]$ instead of $(...(e[x_1 \mapsto a_1])...)[x_\mathrm{n} \mapsto a_\mathrm{n}]$.

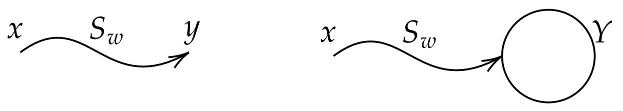
3.2. Diagrams

Throughout the thesis we present some diagrams to represent ordinary and generalized Veltman frames. We believe that diagrams can help the reader have a better understanding of the underlying formula. However, diagrams are not meant to be a replacement as they cannot unambiguously convey all the information in the formula. Here we list some conventions that we use to help the reader understand the presented diagrams.

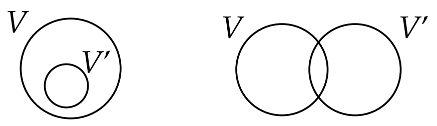
- **Straight arrows**: We use straight arrows to represent the R relation. For instance we represent xRy thus:

$$\begin{tikzcd} x \arrow[r] & y \end{tikzcd}$$

- **Curvy arrows**: We use curvy arrows to represent the S relation. For instance, if we have xS_wy we will draw a curvy line from x to y with label S_w as drawn below. In the case that we are drawing a generalized Veltman frame then Y is a set of worlds which we draw as a circle.



- **Circles and frames**: Circles denote sets of worlds in generalized Veltman frames. As expected, we will draw inner circles to denote subsets and intersected circles to denote that the intersection is nonempty. For instance, to denote that we have $V' \subseteq V$ we will draw the left picture. If we want to express that $V \cap V' \neq \emptyset$ we will draw the right picture.



We will use framed variables to denote singleton sets. We will represent the singleton set $\{y\}$ as $\boxed{y}$ in a diagram.

- **Quantfier annotations**: When a variable that represents a world or a set of worlds in the formula, we may tag that variable with the corresponding quantifier. We only follow this convention in cases when we think it improves the readability of the diagram. For instance, if we want to express the condition $\forall y(\forall x(xRy) \Rightarrow \exists z(yRz))$ we will draw the picture below. As a rule of thumb we do not annotate the quantification of variables which are universally quantified for the whole formula or which are free.

$$\begin{tikzcd} & \exists z \\ \forall x \arrow[r] & y \arrow[u] \end{tikzcd}$$

- **Red and black ink**: We use black ink for conditions which appear in a negative position (assumptions) and red ink for conditions which appear in a positive position (consequences). Thus, the intended meaning of color is that "if everything drawn in black holds, then what is drawn in red must hold". For instance, in order to represent the transitivity condition $xRyRz \Rightarrow xRz$ we would draw the following:

$$\begin{tikzcd} & z \\ x \arrow[r] \arrow[ru, red] & y \arrow[u] \end{tikzcd}$$

4. The language of modal interpretability

The symbols of interpretability logics are $\bot, \to, \rhd$.

Definition 4.1. Modal formula The set of well-formed modal formulas, which we denote with Fm, is defined recursively as usual:

1. *Variable.* If x is a variable, then x is a formula. We assume that we have an infinite countable set of variables. In particular we shall define $\mathsf{Var} := \mathbb{N}$. However, we use non-capital letters to refer to variables.
2. *Bottom.* The constant $\bot$ is a formula.
3. *Implication.* If A and B are formulas, then $(A \to B)$ is a formula.
4. *Interprets.* If A and B are formulas, then $(A \rhd B)$ is a formula.

We will often omit outer parentheses.

Definition 4.2. Convenience operators We define the usual operators and constants in the following way:

1. $\neg A := (A \to \bot)$;
2. $\top := \neg\bot$;
3. $A \vee B := ((\neg A) \to B)$;
4. $A \wedge B := \neg(A \to \neg B)$;
5. $A \leftrightarrow B := (A \to B) \wedge (B \to A)$;
6. $\Box A := ((\neg A) \rhd \bot)$;
7. $\Diamond A := \neg\Box\neg A$.

The precedence from higher to lower of the operators is in the following order:

$$\wedge, \vee, \rhd, \to$$

The scope of unary symbols $\Box, \Diamond, \neg$ is as small as possible. Thus $\Box A \wedge \neg\neg B$ is the same as $(\Box A) \wedge (\neg\neg B)$. Also, $\to$ has right associativity.

As an example consider:

$$A \rhd B \to A \wedge \Box C \rhd B \wedge \Box C$$

It should be read as:

$$(A \rhd B) \to ((A \wedge \Box C) \rhd (B \wedge \Box C))$$

Even though the notation with no parenthesis is acceptable with the rules that we have given, we will often add parentheses to facilitate reading non-trivial formulas.

5. Logic IL

As explained in the overview (Chapter 1), the logic IL is the logic that we use as the base for other interpretability logics. The logic IL extends GL with five new axiom schemas denoted by J1–J5. These new axioms reflect some facts about formalized interpretability.

After the definition of IL we proceed by showing a number of theorems related to it. All of these theorems have been verified by a computer using the proof-assistant Agda. In Part IV we will explain how Agda works and what it means for a proof to be formally verified in Agda. In Part V we will explain how we have formalized the presented theorems and their respective proofs in Agda. The Agda proofs of the theorems presented in this section can be found in Appendix B.23.

Definition 5.1. In this definition we present the Hilbert Calculus for the logic IL. We write $\Pi \vdash_{\mathsf{IL}} A$ to denote that the formula A follows from the set of assumptions Π. If Π is empty we simply write $\vdash_{\mathsf{IL}} A$.

Let us now define the relation $\vdash_{\mathsf{IL}} \subseteq \mathcal{P}(\mathsf{Fm}) \times \mathsf{Fm}$ inductively:

- *Axiom*. If A is an instantiation of any of the IL axiom schemas, then A is a theorem of IL, which we denote with $\vdash_{\mathsf{IL}} A$. We list all IL axiom schemas just below.
- *Modus ponens*: if $\Pi \vdash_{\mathsf{IL}} A \to B$ and $\Pi \vdash_{\mathsf{IL}} A$ then $\Pi \vdash_{\mathsf{IL}} B$.
- *Necessitation*: if $\vdash_{\mathsf{IL}} A$ then $\Pi \vdash_{\mathsf{IL}} \Box A$. Notice that the necessitation rule requires A to be provable from an empty set of assumptions. In the consequence of the rule ($\Pi \vdash_{\mathsf{IL}} \Box A$) we allow an arbitrary context Π.
- *Identity*: If $A \in \Pi$ then $\Pi \vdash_{\mathsf{IL}} A$.

The axioms of IL. The logic IL encompasses all classical theorems in the new language (given by C1, C2 and C3), all theorems of GL in the new language (given by L and K) plus some new axiom schemas:

- C1: $A \to (B \to A)$;
- C2: $(A \to (B \to C)) \to ((A \to B) \to (A \to C))$;
- C3: $(\neg A \to \neg B) \to (B \to A)$;
- K: $\Box(A \to B) \to \Box A \to \Box B$;

  This provability principle is known as the *distribution axiom* and implies that if we can prove that A implies B, then we can prove B from a proof of A.
- L: $\Box(\Box A \to A) \to \Box A$.

  This provability principle corresponds to Löbs theorem.
- J1: $\Box(A \to B) \to A \rhd B$.

  This interpretability principle expresses the fact that if a theory T can prove that A is at least as strong as B, then $T + A$ interprets $T + B$ using the identity interpretation.
- J2: $(A \rhd B) \wedge (B \rhd C) \to A \rhd C$.

  This interpretability principle gives us transitivity for interpretations.

- J3: $((A \rhd C) \wedge (B \rhd C)) \to (A \vee B) \rhd C$.

  This interpretability principle allows us to build interpretations by cases.

- J4: $A \rhd B \to (\Diamond A \to \Diamond B)$.

  This interpretability principle reflects the fact that relative interpretability gives us a proof of relative consistency.

- J5: $(\Diamond A) \rhd A$.

  The last interpretability principle expresses the property that from a consistency proof of A we can build an interpretation of A itself. This reflects the fact that the Henkin construction can be formalized.

Remark 5.2. While it is acceptable to have an infinite set of assumptions Π, when verifying properties in Agda we have restricted ourselves to finite sets and thus we assume that Π is finite in the Agda proof.

Theorem 5.3. ***Weakening*** *If* $\Pi \vdash_{\mathsf{IL}} A$ *then* $B, \Pi \vdash_{\mathsf{IL}} A$.

Proof. The proof is by induction on the proof. In Agda it is done by an induction on the proof. We only need to take care of shifting one position the references to assumptions. □

Theorem 5.4. ***Deduction*** $\Pi \vdash_{\mathsf{IL}} A \to B$ *iff* $A, \Pi \vdash_{\mathsf{IL}} B$.

Proof. The $\Rightarrow$ direction is trivial. For the other direction we proceed by induction on the proof $A, \Pi \vdash_{\mathsf{IL}} B$. We need to show that if $A, \Pi \vdash_{\mathsf{IL}} B$ then $\Pi \vdash_{\mathsf{IL}} A \to B$. If B is an instance of any of the axioms, we can show that $A \to B$ follows from MP, C1 and the corresponding axiom. If $B = \Box B'$ follows from the necessitation rule then by definition of the necessitation rule we have $\vdash_{\mathsf{IL}} B'$ and thus by necessitation, C1 and MP we can prove $\Pi \vdash_{\mathsf{IL}} A \to \Box B'$. If B follows from an assumption we have two cases. If $B = A$ then we show $\vdash_{\mathsf{IL}} A \to A$ as we do in classical logic. If $B \in \Pi$ we proceed as before using MP, C1. Finally, if B is the result of a MP application then we have that $A, \Pi \vdash_{\mathsf{IL}} C \to B$ and $A, \Pi \vdash_{\mathsf{IL}} C$ by the IH we have $\Pi \vdash_{\mathsf{IL}} A \to (C \to B)$ and $\Pi \vdash_{\mathsf{IL}} A \to C$, thus we can show $\Pi \vdash_{\mathsf{IL}} A \to B$ by the C2 axiom and two applications of MP. □

Theorem 5.5. ***Cut*** *If* $\Pi \vdash_{\mathsf{IL}} B$ *and* $B, \Pi \vdash_{\mathsf{IL}} A$ *then* $\Pi \vdash_{\mathsf{IL}} A$.

Proof. It follows by an easy induction on the proof $\Pi \vdash_{\mathsf{IL}} B$. □

Theorem 5.6. ***Structurality*** *If* $\Pi \vdash_{\mathsf{IL}} A$ *and* σ *is a substitution then* $\sigma[\Pi] \vdash_{\mathsf{IL}} \sigma(A)$.

Proof. It follows by an easy induction on the proof $\Pi \vdash_{\mathsf{IL}} A$. □

Theorem 5.7. ***Conjunction*** $\Pi \vdash_{\mathsf{IL}} A \wedge B$ *iff* $\Pi \vdash_{\mathsf{IL}} A$ *and* $\Pi \vdash_{\mathsf{IL}} B$.

Proof. The key part is to show that $\Pi \vdash_{\mathsf{IL}} A \to B \to (A \wedge B)$, $\Pi \vdash_{\mathsf{IL}} A \wedge B \to A$ and $\Pi \vdash_{\mathsf{IL}} A \wedge B \to B$ as we do in classical logic. □

Theorem 5.8. *The following formulas are theorems of* IL*:*

1. $\vdash_{\mathsf{IL}} A \to A$;
2. $\vdash_{\mathsf{IL}} A \rhd A$;
3. $\vdash_{\mathsf{IL}} (A \to B) \to (B \to C) \to A \to C$;
4. $\vdash_{\mathsf{IL}} A \to \neg\neg A$;

5. $\vdash_{\mathsf{IL}} (\neg\neg A) \to A$;
6. $\vdash_{\mathsf{IL}} (A \to B) \to \neg B \to \neg A$;
7. $\vdash_{\mathsf{IL}} A \to \top$;
8. $\vdash_{\mathsf{IL}} \bot \to A$;
9. $\vdash_{\mathsf{IL}} \neg A \to A \to B$;
10. $\vdash_{\mathsf{IL}} A \wedge B \to A$;
11. $\vdash_{\mathsf{IL}} A \wedge B \to B$;
12. $\vdash_{\mathsf{IL}} (A \to B \to C) \to B \to A \to C$;
13. $\vdash_{\mathsf{IL}} A \to B \to A \wedge B$;
14. $\vdash_{\mathsf{IL}} A \to A \vee B$;
15. $\vdash_{\mathsf{IL}} B \to A \vee B$;
16. $\vdash_{\mathsf{IL}} A \rhd (A \vee \Diamond A)$;
17. $\vdash_{\mathsf{IL}} (A \vee \Diamond A) \rhd A$;
18. $\vdash_{\mathsf{IL}} A \to B \quad \Rightarrow \quad \vdash_{\mathsf{IL}} \Box A \to \Box B$;
19. $\vdash_{\mathsf{IL}} A \leftrightarrow B \quad \Rightarrow \quad \vdash_{\mathsf{IL}} \Box A \leftrightarrow \Box B$;
20. $\vdash_{\mathsf{IL}} \Box(A \wedge B) \leftrightarrow (\Box A \wedge \Box B)$;
21. $\vdash_{\mathsf{IL}} A \to B \quad \Rightarrow \quad \vdash_{\mathsf{IL}} \Diamond A \to \Diamond B$;
22. $\vdash_{\mathsf{IL}} A \leftrightarrow B \quad \Rightarrow \quad \vdash_{\mathsf{IL}} \Diamond A \leftrightarrow \Diamond B$;
23. $\vdash_{\mathsf{IL}} \neg(A \wedge B) \leftrightarrow \neg A \vee \neg B$;
24. $\vdash_{\mathsf{IL}} (A \vee \neg B) \to (A \wedge B \vee \neg B)$.

Proof. All proofs have been formalized in Agda. Here we only show two examples. Consider theorems 16 and 17, namely $A \rhd (A \vee \Diamond A)$ and $(A \vee \Diamond A) \rhd A$. To prove 16 we assume that we have already showed theorem 14, that is, $\vdash_{\mathsf{IL}} A \to A \vee B$.

1. $A \to A \vee \Diamond A$	by $A \to A \vee B$
2. $\Box(A \to A \vee \Diamond A)$	by Nec
3. $\Box(A \to A \vee \Diamond A) \to A \rhd (A \vee \Diamond A)$	by J1
4. $A \rhd (A \vee \Diamond A)$	by MP on 2, 3

To prove 17 we assume we have already showed theorem 2, that is, $\vdash_{\mathsf{IL}} A \rhd A$.

1. $A \rhd A$	
2. $(\Diamond A \rhd A)$	by J5
3. $(A \rhd A) \wedge (\Diamond A \rhd A)$	Conjunction, Theorem 5.7
4. $(A \rhd A) \wedge (\Diamond A \rhd A) \to ((A \vee \Diamond A) \rhd A)$	by J3
5. $(A \vee \Diamond A) \rhd A$	by MP 3, 4

In Chapter 31 we will present a verified language to write verified Hilbert style proofs using a computer with paper-like syntax. □

6. Veltman Semantics

In this thesis we consider two variants of relational semantics for interpretability logics similar to Kripke semantics for other modal logics.

6.1. Ordinary Veltman semantics

Ordinary Veltman semantics were the first relational semantics for interpretability logics and were introduced by Frank Veltman ([17]).

All the definitions in this section have been formalized in Agda and can be found in Appendices B.24 and B.31. Later in Chapter 28 we will comment on the Agda implementation of ordinary Veltman semantics.

Definition 6.1. An ordinary Veltman frame $F = \langle W, R, S \rangle$ is a structure constituted by a non-empty set of worlds W, a binary relation $R \subseteq W^2$ and a ternary relation $S \subseteq W \times W \times W$. We write wRu instead of $\langle w, u \rangle \in R$ and $uS_w v$ instead of $\langle w, u, v \rangle \in S$. The structure must satisfy the following conditions:

OF-1: R is transitive;

OF-2: R is conversely well-founded. That is, there is no infinite ascending chain $w_1 R w_2 R ...$;

OF-3: if $uS_w v$ then wRu and wRv;

$$\begin{tikzcd}
u \arrow[rr, bend left, "S_w"] & & v \\
& w \arrow[ul] \arrow[ur] &
\end{tikzcd}$$

OF-4: if wRu then $uS_w u$;

$$\begin{tikzcd}
u \arrow[loop above, "S_w"] \\
w \arrow[u]
\end{tikzcd}$$

OF-5: if wRu and uRv then $uS_w v$;

$$\begin{tikzcd}
u \arrow[r] \arrow[r, bend right, "S_w"'] & v \\
w \arrow[u] &
\end{tikzcd}$$

OF-6: for every w, S_w is transitive.

The Agda definition can be found in Appendix B.24.

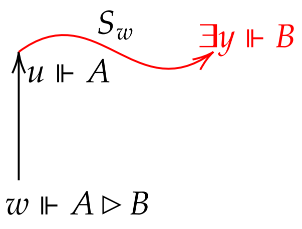
As usual, we can build a model by attaching a valuation to a frame. The valuation will tell us which propositional variables are true in each world.

Definition 6.2. An ordinary Veltman model $M = \langle F, V\rangle$ is a structure constituted by an ordinary Veltman frame F and a valuation $V \subseteq W \times \mathsf{Var}$. If $F = \langle W, R, S\rangle$ we will write $M = \langle W, R, S, V\rangle$ instead of $M = \langle\langle W, R, S\rangle, V\rangle$.
The Agda definition can be found in Appendix B.31.

We will proceed by defining a forcing relation. The forcing relation tells us what formulas are forced (hold) in each world.

Definition 6.3. Given a model M, we define a forcing relation $\Vdash^{ord}_{M} \subseteq W \times \mathsf{Fm}$. We write $M, w \Vdash A$ instead of $\langle w, A\rangle \in \Vdash^{ord}_{M}$ or simply $w \Vdash A$ when the model is clear from the context. We write $w \nVdash A$ when $\langle w, A\rangle \notin \Vdash_M$.

1. $w \nVdash \bot$;
2. if $p \in \mathsf{Var}$, then $w \Vdash p$ iff $\langle w, p\rangle \in V$;
3. if $A, B \in \mathsf{Fm}$, then $w \Vdash A \to B$ iff if $w \Vdash A$ then $w \Vdash B$;
4. if $A, B \in \mathsf{Fm}$, then $w \Vdash A \rhd B$ iff if wRu and $u \Vdash A$ then there exists some v such that $v \Vdash B$ and $uS_w v$. Below we draw the condition for a world w to force $A \rhd B$.



The Agda definition can be found in Appendix B.31.

If M is an ordinary Veltman model and A a formula, we write $M \Vdash A$ to denote that for every world w we have $M, w \Vdash A$. Similarly, if F is an ordinary Veltman frame and A a formula, we write $F \Vdash A$ to denote that for every valuation V we have $\langle F, V\rangle \Vdash A$.

Lemma 6.4. Let M be an ordinary Veltman model and let w be a world in M. It can be shown that:

1. If $A, B \in \mathsf{Fm}$, then $w \Vdash A \wedge B$ iff $w \Vdash A$ and $w \Vdash B$;
2. If $A, B \in \mathsf{Fm}$, then $w \Vdash A \vee B$ iff $w \Vdash A$ or $w \Vdash B$;
3. if $A \in \mathsf{Fm}$, then $w \Vdash \neg A$ iff $w \nVdash A$;
4. if $A \in \mathsf{Fm}$, then $w \Vdash \Box A$ iff for every u such that wRu we have $u \Vdash A$;
5. if $A \in \mathsf{Fm}$, then $w \Vdash \Diamond A$ iff there exists u such that wRu and $u \Vdash A$.

Proof. Here we show the case for $w \Vdash \Box A$. Assume $w \Vdash \Box A$ and let u be a world such that wRu. If $u \Vdash A$ we are done, otherwise we have $u \Vdash \neg A$. As described before $w \Vdash \Box A$ is notation for $w \Vdash (\neg A) \rhd \bot$, thus there exists z such that $uS_w z \Vdash \bot$, but this is a contradiction.

For the other direction assume that we have a world w and for every u such that wRu we have $u \Vdash A$. We see that $w \Vdash (\neg A) \rhd \bot$ clearly holds as there is no u such that $wRu \Vdash \neg A$.

The Agda proof can be found in Appendix B.30. □

Theorem 6.5. ***Decidability*** *This theorem must be understood in the context of a proof assistant. In our case, Agda. If W is finite and R, S, V are decidable Agda relations, then the forcing relation associated with the model $M := \langle W, R, S, V \rangle$ is decidable.*

Proof. We have implemented a verified algorithm that given the mentioned conditions, a world w and a formula A, constructs either a proof of $M, w \Vdash A$ or a proof of $M, w \nVdash A$. The implementation of this algorithm can be found in Appendix B.25. We give details of our Agda formalization in Part V. □

Theorem 6.6. ***Local soundness for ordinary semantics*** *If $\Pi \vdash_{\mathsf{IL}} A$ and M is an ordinary model with a world w such that $w \Vdash \Pi$, then $w \Vdash A$.*

Proof. The proof is by induction on the proof $\Pi \vdash_{\mathsf{IL}} A$. The cases for necessitation and modus ponens follow immediately from the IH. If A is an instance of C1, C2 or C3 then it is routine to check that all of these axioms hold in w. For axioms L and K we proceed as we do for GL and Kripke semantics. In Section 29.1 we show how we proved soundness for L in Agda. Finally we need to check soundness for axioms J1–J5.

- J1: $\Box(A \to B) \to A \rhd B$. Assume $w \Vdash \Box(A \to B)$ and $wRu \Vdash A$, then it follows that $u \Vdash B$ and by condition 4 of ordinary frames we get $uS_w u \Vdash B$.
- J2: $A \rhd B \wedge B \rhd C \to A \rhd C$. Assume $w \Vdash A \rhd B$ and $w \Vdash B \rhd C$ and $wRu \Vdash A$. It follows that there exists v such that $uS_w v \Vdash B$, then we have wRv from definition of ordinary frame and thus there exists z such that $vS_w z \Vdash C$. Finally by transitivity of S_w we get $uS_w z \Vdash C$.
- J3: $(A \rhd C \wedge B \rhd C) \to (A \vee B) \rhd C$. Assume $w \Vdash A \rhd C$ and $w \Vdash B \rhd C$ and $wRu \Vdash A \vee B$. If $u \Vdash A$ then since $w \Vdash A \rhd C$ we have that there exists v such that $uS_w v \Vdash C$. On the other hand if $u \Vdash B$ we proceed analogously.
- J4: $A \rhd B \to \Diamond A \to \Diamond B$. Assume $w \Vdash A \rhd B$ and $w \Vdash \Diamond A$. Then there exists u such that $wRu \Vdash A$. Since $w \Vdash A \rhd B$ it follows that there exists v such that $uS_w v \Vdash B$. Finally by OF-3 of ordinary frames we have wRv and thus $w \Vdash \Diamond B$.
- J5: $\Diamond A \rhd A$. Assume that there exists u such that $wRu \Vdash \Diamond A$. Then there exists v such that $uRv \Vdash A$. By OF-5 of ordinary frames we have $uS_w v \Vdash A$.

The Agda proof can be found in Appendix B.23. □

6.2. Generalized Veltman semantics

Generalized Veltman semantics were introduced by Verbrugge in an unpublished manuscript ([36]). We were given permission by the author herself to include the manuscript in Appendix D, where we also include some comments about the authorship of some handwritten notes in the document.

Generalized semantics generalize ordinary Veltman semantics in the sense that each S_w relates worlds to sets of worlds. Thanks to this change, generalized Veltman semantics turn out to be more convenient and necessary in some cases. For instance, the logic $\mathsf{ILP_0}$ (we will define principle $\mathsf{P_0}$ in Chapter 16) is complete with respect to its characteristic class of generalized Veltman frames but incomplete with respect to ordinary Veltman semantics.

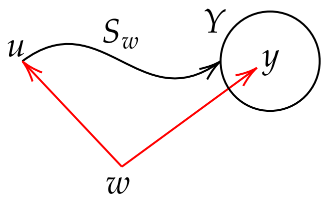
In Chapter 29 we will comment on the Agda implementation of generalized Veltman semantics.

Definition 6.7. A generalized Veltman frame $F = \langle W, R, S \rangle$ is a structure constituted by a non-empty set of worlds W, a binary relation $R \subseteq W^2$ and a ternary relation $S \subseteq W \times W \times (\mathcal{P}(W) \setminus \{\emptyset\})$. We write wRu instead of $\langle w, u \rangle \in R$ and uS_wY instead of $\langle w, u, Y \rangle \in S$. The structure must satisfy the following conditions:

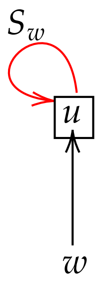
GF-1: R is transitive;

GF-2: R is conversely well-founded. That is, there is no infinite ascending chain $w_1Rw_2R...$;

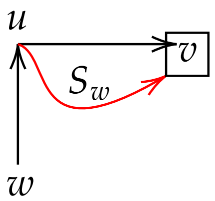
GF-3: if uS_wY, then wRu and for all $y \in Y$ we have wRy;



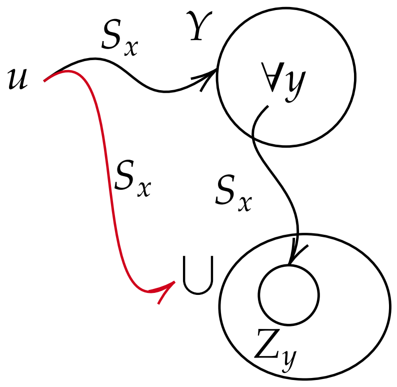
GF-4: *quasi-reflexivity*: if wRu then $uS_w\{u\}$;



GF-5: if wRu and uRv then $uS_w\{v\}$;



- *quasi-transitivity*: if uS_xY and yS_xZ_y for all $y \in Y$, then $uS_x\left(\bigcup_{y \in Y} Z_y\right)$.



Notice that we did not label the last condition GF-6. The reason is that the above condition is a particular (the standard) condition of quasi-transitivity. However, throughout this thesis we explore a total of eight notions, see Chapter 7. Because of that, we will refer to those as quasi-transitivity Condition j for $1 \leq j \leq 8$. If it is clear by the context we may simply say Condition i.

The Agda definition can be found in Appendix B.6.

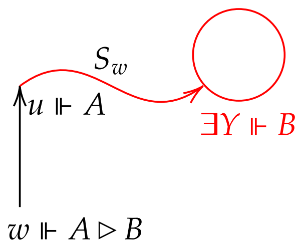
As we did for ordinary semantics, we may endow a generalized frame with a valuation to obtain a generalized model.

Definition 6.8. A generalized Veltman model $M = \langle F, V\rangle$ is a structure constituted by a generalized Veltman frame F and a valuation $V \subseteq W \times \mathsf{Var}$.
The Agda definition can be found in Appendix B.20.

We define the forcing relation in a similarly to ordinary semantics, the only difference being in the case for the operator $\rhd$.

Definition 6.9. Given a model M, we define a forcing relation $\Vdash^{gen}_{M} \subseteq W \times \mathsf{Fm}$. We use the same notational conventions as in the ordinary semantics.

1. $w \nVdash \bot$;
2. if $p \in \mathsf{Var}$, then $w \Vdash p$ iff $\langle w, p\rangle \in V$;
3. if $A, B \in \mathsf{Fm}$, then $w \Vdash A \to B$ iff if $w \Vdash A$ then $w \Vdash B$;
4. if $A, B \in \mathsf{Fm}$, then $w \Vdash A \rhd B$ iff if wRu and $u \Vdash A$ then there exists some Y such that $Y \Vdash B$ and uS_wY. When we write $Y \Vdash B$ we mean that for all $y \in Y$ we have $y \Vdash B$. Below we draw the condition for a world w to force $A \rhd B$.



The Agda definition can be found in Appendix B.20.

We have the same notation conventions that we have for ordinary semantics: If M is a generalized Veltman model and A a formula, we write $M \Vdash A$ to denote that for every world w we have $M, w \Vdash A$. Similarly, if F is a generalized Veltman frame and A a formula, we write $F \Vdash A$ to denote that for every valuation V we have $\langle F, V\rangle \Vdash A$.

Lemma 6.10. We can show the same results presented in Lemma 6.4 for generalized semantics:

1. If $A, B \in \mathsf{Fm}$, then $w \Vdash A \wedge B$ iff $w \Vdash A$ and $w \Vdash B$;
2. If $A, B \in \mathsf{Fm}$, then $w \Vdash A \vee B$ iff $w \Vdash A$ or $w \Vdash B$;
3. If $A \in \mathsf{Fm}$, then $w \Vdash \neg A$ iff $w \nVdash A$;
4. If $A \in \mathsf{Fm}$, then $w \Vdash \Box A$ iff for every u such that wRu we have $u \Vdash A$;

5. If $A \in \mathsf{Fm}$, then $w \Vdash \Diamond A$ iff there exists u such that wRu and $u \Vdash A$.

Proof. Here we show the case for $w \Vdash \Box A$. The proof goes in a very similar way to ordinary semantics. Assume $w \Vdash \Box A$ and let u be a world such that wRu. If $u \Vdash A$ we are done, otherwise we have $u \Vdash \neg A$. Since $w \Vdash \Box A$ is notation for $w \Vdash (\neg A) \rhd \bot$, there exists Z such that $uS_wZ \Vdash \bot$. Since Z is nonempty we have a contradiction.

For the other direction assume that we have a world w and for every u such that wRu we have $u \Vdash A$. We see that $w \Vdash (\neg A) \rhd \bot$ clearly holds as there is no u such that $wRu \Vdash \neg A$.

The Agda formalization can be found in Appendix B.19. □

Theorem 6.11. ***Local soundness for generalized semantics*** *If $\Pi \vdash_{\mathsf{IL}} A$ and M is a generalized Veltman model with a world w such that $w \Vdash \Pi$, then $w \Vdash A$.*

Proof. We only show the case where A is an instance of the J2 $(A \rhd B \wedge B \rhd C \to A \rhd C)$ axiom. The rest of the cases are proved in an analogous way to ordinary Semantics. Also, here we only show it the quasi-transitivity property given in Definition 6.7 and also for the quasi-transitivity Condition 8 in Table 7.1. However, we have verified this in Agda for all the alternative quasi-transitivity conditions presented in Table 7.1.

Assume $w \Vdash A \rhd B$ and $w \Vdash B \rhd C$ and that there exists u such that $wRu \Vdash A$. It follows that there exists V such that $uS_wV \Vdash B$. By GF-3 of a generalized frame we have that $\forall v \in V. wRv \Vdash B$. Then for every $v \in V$ we have that there exists Z_v such that $vS_wZ_v \Vdash C$. It follows from the quasi-transitivity condition that $uS_w(\bigcup_{v \in V} Z_v)$ and clearly $\bigcup_{v \in V} Z_v \Vdash C$.

Now for quasi-transitivity Condition 8. Assume $w \Vdash A \rhd B$ and $w \Vdash B \rhd C$ and that there exists u such that $wRu \Vdash A$. It follows that there exists V such that $uS_wV \Vdash B$. If $V \Vdash C$ we are done, otherwise there exists $v \in V$ such that $v \nVdash C$. By GF-3 of a generalized frame we have that wRv. Since $wRv \Vdash B$ and $w \Vdash B \rhd C$ it follows that there exists Z such that $vS_wZ \Vdash C$. Finally since $v \nVdash C$ we know that $v \notin Z$ so by quasi-transitivity Condition 8 we can conclude $uS_wZ \Vdash C$.

The Agda proof can be found in Appendix B.23. □

7. Quasi-transitivity

In the literature one can find several semantic requirements for the quasi-transitivity condition which we present in the table below. We observe that in Definition 6.7 we used Condition 2 from the table below. Theorem 7.1 presents some direct implications between conditions presented in Table 7.1. Theorems 6.11 and 7.2 are sufficient to argue that all of them are appropriate for proving completeness of IL. It is worth mentioning however, that not all of them are sufficiently expressive to prove completeness for extensions of IL.

Nr.	Semantic requirement for quasi-transitivity	First mentioned in
1	$uS_xY \Rightarrow \forall\, \{Z_y\}_{y\in Y}\Big((\forall\, y \in Y\ yS_xZ_y) \Rightarrow \exists V \subseteq \bigcup_{y\in Y} Z_y \wedge uS_xV\Big)$	Joosten et al. '20 [19]
2	$uS_xY \Rightarrow \forall\, \{Z_y\}_{y\in Y}\Big((\forall\, y \in Y\ yS_xZ_y) \Rightarrow uS_x \bigcup_{y\in Y} Z_y\Big)$	Verbrugge '92 '20 [36]
3	$uS_xY \Rightarrow \exists\, y \in Y\ \forall Y'(yS_xY' \Rightarrow \exists\, Y''\subseteq Y' \wedge uS_xY'')$	Joosten et al. [19]
4	$uS_xY \Rightarrow \exists\, y \in Y\ \forall Y'(yS_xY' \Rightarrow uS_xY')$	Joosten '98 [18]
5	$uS_xY \Rightarrow \forall\, y \in Y\ \forall Y'(yS_xY' \Rightarrow \exists\, Y''\subseteq Y' \wedge uS_xY'')$	Joosten et al. '20 [19]
6	$uS_xY \Rightarrow \forall\, y \in Y\ \forall Y'(yS_xY' \Rightarrow uS_xY')$	Verbrugge '92 [36]
7	$uS_xY \Rightarrow \forall\, y \in Y\ \forall Y'(yS_xY' \wedge y \notin Y' \Rightarrow \exists\, Y''\subseteq Y'\ uS_xY'')$	Joosten et al. '20 [19]
8	$uS_xY \Rightarrow \forall\, y \in Y\ \forall Y'(yS_xY' \wedge y \notin Y' \Rightarrow uS_xY')$	Goris, Joosten '09 [14]

Table 7.1.: Semantic requirements for quasi-transitivity mentioned in the literature.

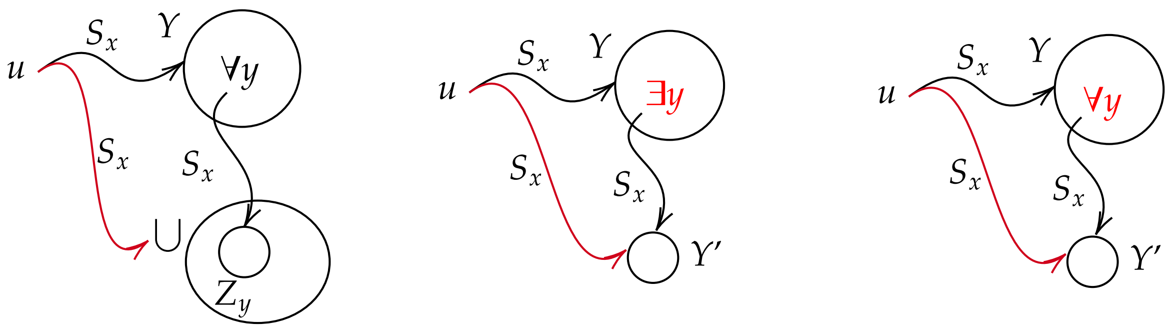


Figure 7.1.: Diagrams for Conditions 2, 4 and 6.

Theorem 7.1. *Let F be a generalized Veltman frame. Let*

$$\mathsf{Mon} := \forall w, u, V, Z(uS_wV \subseteq Z \subseteq \{u : wRu\} \Rightarrow uS_wZ)$$

represent the monotonicity condition. The following implications hold.
The first item should be read as $F \vDash \mathsf{Mon} \wedge (1) \to (2)$.

1. $\mathsf{Mon} \wedge (1) \Rightarrow (2)$
2. $(2) \Rightarrow (1)$
3. $\mathsf{Mon} \wedge (3) \Rightarrow (4)$
4. $(4) \Rightarrow (3)$
5. $(5) \Rightarrow (1)$
6. $\mathsf{Mon} \wedge (5) \Rightarrow (2)$
7. $(5) \Rightarrow (3)$
8. $\mathsf{Mon} \wedge (5) \Rightarrow (4)$
9. $\mathsf{Mon} \wedge (5) \Rightarrow (6)$
10. $(5) \Rightarrow (7)$
11. $\mathsf{Mon} \wedge (5) \Rightarrow (8)$
12. $(6) \Rightarrow (1)$
13. $\mathsf{Mon} \wedge (6) \Rightarrow (2)$
14. $(6) \Rightarrow (3)$
15. $(6) \Rightarrow (4)$
16. $(6) \Rightarrow (5)$
17. $(6) \Rightarrow (7)$
18. $(6) \Rightarrow (8)$
19. $\mathsf{Mon} \wedge (7) \Rightarrow (8)$
20. $(8) \Rightarrow (7)$

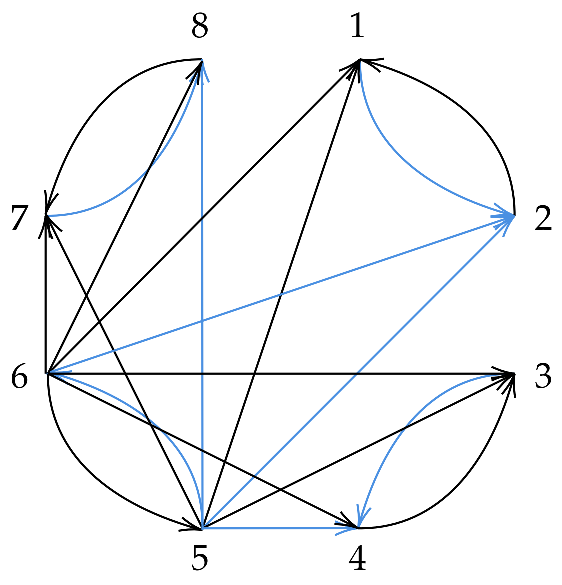


Figure 7.2.: Graphical representation of Theorem 7.1. Blue lines require monotonicity.

Proof. Here we only show item 13: $\mathsf{Mon} \wedge (6) \Rightarrow (2)$. Assume that F is a generalized Veltman frame that satisfies the monotonicity condition and the quasi-transitivity Condition 6. Now assume that uS_xY and consider an arbitrary family of sets of worlds $\{Y_y : y \in Y\}$. Assume also that for every $y \in Y$ we have yS_xY_y. Since Y is nonempty we may pick $y_0 \in Y$. Then we have that $yS_xY_{y_0}$. Then by quasi-transitivity Condition 6 we have that $uS_xY_{y_0}$ and since for any $y \in Y$ we have yS_xY_y it follows by GF-3 of a generalized frame that $Y_y \subseteq \{v : xRv\}$. Finally we see that $Y_{y_0} \subseteq \bigcup_{y \in Y} Y_y \subseteq \{v : xRv\}$ and by monotonicity it follows that $uS_x(\bigcup_{y \in Y} Y_y)$.

The Agda proof can be found in Appendix B.5. □

Theorem 7.2. *Given an ordinary Veltman model $M = \langle W, R, S, V \rangle$ we can find some generalized Veltman model $M' = \langle W, R, S', V \rangle$, where we can replace our notion of quasi-transitivity by any of the Conditions 1–8. Furthermore, for every world w and formula A:*

$$M, w \Vdash A \Leftrightarrow M', w \Vdash A.$$

Proof. We prove it for the quasi-transitivity Condition 2. The rest can be proven in the same way. Let $M = \langle W, R, S, V \rangle$ be an ordinary model. Let $M' := \langle W, R, S', V \rangle$ with S' defined thus:

$$S' := \{\langle w, x, \{y\}\rangle : \langle w, x, y \rangle \in S\}.$$

It is easy to observe that M' satisfies GF-1, ..., GF-5. It is also easy to see that it satisfies quasi-transitivity Condition 2. We show that they force the same formulas by induction on the complexity of the formula. The only interesting case is $A \rhd B$.

- Assume $M, w \Vdash A \rhd B$ and that for some x we have $wRx \Vdash A$. It follows that there exists some y such that $xS_w y \Vdash B$. By definition of M' we have $xS'_w\{y\}$ and also $\{y\} \Vdash B$, therefore $M', w \Vdash A \rhd B$.
- Assume $M, w \nVdash A \rhd B$, then there exists some x such that $wRx \Vdash A$ and $\forall y(xS_w y \Rightarrow y \nVdash B)$. It is obvious that for M' we have $\forall y(xS'_w\{y\} \Rightarrow y \nVdash B)$ and also $\forall Y(xS'_w Y \Rightarrow Y \nVdash B)$, which is the required property.

$\square$

8. Monotonicity

Recall the monotonicity condition that we presented in the previous chapter:

$$\text{if } uS_w V \subseteq Z \subseteq \{v : wRv\} \text{ then } uS_w Z.$$

It happens that this condition can be assumed (and in fact, is a standard assumption in the more recent literature) to be satisfied by generalized Veltman frames without harm. This is desirable as a good number of proofs and definitions (especially definitions related to filtrations) can be simplified when assuming the monotonicity condition. By "can be assumed without harm", we mean that for any generalized Veltman frame, we can find another generalized Veltman frame that satisfies the monotonicity condition. Moreover, both frames will be modally equivalent when expanded to a generalized Veltman model with a valuation. In the following theorem we prove this fact.

Theorem 8.1. *Let $F = \langle W, R, S\rangle$ be a generalized Veltman frame with quasi-transitivity Condition i for $i \in \{1, ..., 8\}$. Let $F' = \langle W, R, S'\rangle$ where S' is the monotone closure of S:*

$$S' := \{\langle w, x, Y'\rangle : \langle w, x, Y\rangle \in S, Y \subseteq Y' \subseteq \{u : wRu\}\}.$$

Then F' is a generalized Veltman frame satisfying quasi-transitivity Condition 2. Furthermore, let V be an arbitrary valuation and A an arbitrary formula. Let $M := \langle F, V\rangle$ and $M' := \langle F', V\rangle$. We have that for every world w:

$$M, w \Vdash A \Leftrightarrow M', w \Vdash A.$$

Proof. We check conditions listed in Definition 6.7.

- GF-1 and GF-2 are clear since R is unchanged;
- GF-3 follows from the fact that in the definition of S' we require $Y' \subseteq \{u : wRu\}$;
- for GF-4 and GF-5 observe that $S \subseteq S'$. Then, since these conditions hold for F they also hold for F';
- for quasi-transitivity Condition 2 assume that $uS'_x Y'$ and that for every $y' \in Y'$ we have $y'S'_x\Upsilon_{y'}$. We need to show that $uS'_x \bigcup_{y'\in Y'} \Upsilon_{y'}$. By definition of S' it follows that there exists $Y \subseteq Y'$ such that uS_xY, furthermore, for every $y' \in Y'$ we have that there exists $f(\Upsilon_{y'}) \subseteq \Upsilon_{y'}$ such that $y'S_x f(\Upsilon_{y'})$. From $Y \subseteq Y'$ it follows that for all $y \in Y$ there exists $f(\Upsilon_y) \subseteq \Upsilon_y$ such that $yS_x f(\Upsilon_y)$. Then by (2) for F it follows that $uS_x \bigcup_{y\in Y} f(\Upsilon_y)$. Then see that $\bigcup_{y\in Y} f(\Upsilon_y) \subseteq \bigcup_{y'\in Y'} \Upsilon_{y'}$. It remains to show $\bigcup_{y'\in Y'} \Upsilon_{y'} \subseteq \{u : xRu\}$. Consider some u such that there is some $y' \in Y'$ with $u \in \Upsilon_{y'}$. By assumption we have $y'S'_x\Upsilon_{y'}$ and thus xRu.

To show $M, w \Vdash A \Leftrightarrow M', w \Vdash A$ we proceed by induction on A. The only interesting case is $A \rhd B$.

- Assume that $M, w \Vdash A \rhd B$ and that there is some world x such that wRx and $M', x \Vdash A$. By IH we have $M, x \Vdash A$, so there exists some Y such that xS_wY and $M, Y \Vdash B$. By IH we have $M', Y \Vdash B$ and by definition of S' it follows that xS'_wY, therefore $M', w \Vdash A \rhd B$.

- Assume that $M, w \nVdash A \rhd B$. It follows that there is some x such that wRx, $M, x \Vdash A$ and
$$\forall Y(xS_{\mathrm{wY}} \Rightarrow M, Y \nVdash B). \tag{8.1}$$
We want to prove that $\forall Y'(xS'_w Y' \Rightarrow M', Y' \nVdash B)$. Assume that for some Y' we have $xS'_w Y'$. By definition of S' it follows there exists some Y such that $Y \subseteq Y'$ and $xS_w Y$. Hence by Eq. (8.1) we have that $M, Y \nVdash B$ and thus there exists $y \in Y$ such that $M, y \nVdash B$. By IH we get that $M', y \nVdash B$ and since $y \in Y \subseteq Y'$ we have $Y' \nVdash B$, so $M', w \nVdash A \rhd B$.

The Agda proof can be found in Appendix B.5. □

Remark 8.2. Taking the monotone closure of each S_w is essentially different from assuming that each S_w is monotone by definition of the frame, as then the forcing relation may change. In the following example we present a generalized Veltman model with Condition 8 that showcases such behaviour.

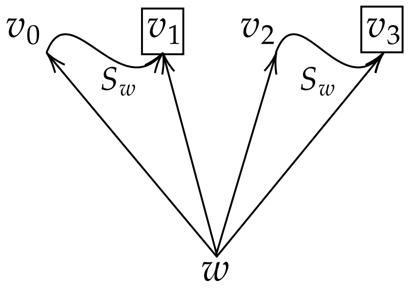


Figure 8.1.: Example frame: $wRv_0, wRv_1, wRv_2, wRv_3$, $v_0S_w\{v_1\}$, $v_2S_w\{v_3\}$.

Let M be a model based on the frame displayed[1] in Fig. 8.1 such that $[\![p]\!] = \{v_0\}$, $[\![q]\!] = \{v_3\}$. We see that $w \Vdash \neg(p \rhd q)$ as p is only true in v_0 and we only have $v_0S_w\{v_1\}$ and $v_0S_w\{v_0\}$ (by quasi-reflexivity) with $v_0 \nVdash q$ and $v_1 \nVdash q$. If we take the monotonic closure of S we have $v_0S_w\{v_1, v_2\}$ and by quasi-transitivity Condition 8 we get $v_0S_w\{v_3\}$ and consequently $w \Vdash \neg(p \rhd q)$ is no longer true.

[1]In the figure we do not show the S_w relations required by quasi-reflexivity for clarity.

Part II.

Generalized vs ordinary models

In this part we explore the expressiveness of ordinary and generalized Veltman semantics. In particular, we discuss how we can transform an ordinary model into a generalized model and vice versa. Needless to say, we have the requirement that the transformation preserves the modal theoremhood of the original model. The notion of modal theoremhood is made precise by Definitions 8.3 and 8.4.

Definition 8.3. Modally equivalent worlds. Given models M and M', we say that two worlds $w \in M$ and $w' \in M'$ are modally equivalent iff for every formula A we have:

$$M, w \Vdash A \Leftrightarrow M, w' \Vdash A.$$

Definition 8.4. Modally equivalent models. Given models M and M', we say that M and M' are modally equivalent iff for every world $w \in M$ there is a world $w' \in M'$ such that w and w' are modally equivalent. And vice versa, for every world in $w' \in M'$ there exists a world $w \in M$ such that w and w' are modally equivalent

In Chapter 9 we see a straightforward transformation from an ordinary Veltman model into a generalized Veltman model. In Chapter 10 we see an involved transformation from a generalized model into an ordinary model. This transformation is due to Verbrugge and was described in [36]. The proof was originally described to work with quasi-transitivity Condition 6. We have slightly improved the result by showing that the same transformation also works for Condition 3, 4 and 5. In Chapter 11 we show a transformation that achieves the same as Verbrugge's transformation but it is much simpler. The simpler transformation was suggested by Mikec during online correspondence.

9. From ordinary to generalized

In this chapter we present a theorem that shows how an ordinary model M naturally gives rise to a generalized model M for any of the presented quasi-transitivity conditions. The resulting generalized model M' has the same set of worlds as the original and is modally equivalent to M.

Theorem 9.1. *Let $M = \langle W, R, S, V\rangle$ be an ordinary Veltman model. We define $M' := \langle W, R, S', V\rangle$ where $S' := \{\langle w, u, \{v\}\rangle : \langle w, u, v\rangle \in S\}$. Now we distinguish two cases.*

- *If we want M' to satisfy quasi-transitivity Condition 2 we take the monotone closure of S' as described in Theorem 8.1;*
- *for the rest of the quasi-transitivity conditions we keep S' as defined.*

Then M' is a generalized Veltman frame with quasi-transitivity condition $(i) \in \{1, ..., 8\}$. Furthermore, for any world w and formula A we have that

$$M, w \Vdash A \Leftrightarrow M', w \Vdash A.$$

Proof. We observe that the transitivity condition for the ordinary models M entails the quasi-transitivity Condition 6 for the M' generalized model. Keep in mind that by definition of M' we only have singleton sets in the third component of S'. Now assume that $uS'_x\{y\}$ and $yS'_x\{y'\}$. By definition of S' it follows that $uS_x yS_x y'$ and by quasi-transitivity of M we have $uS_x y'$ and thus $uS_x\{y'\}$. Then, by Theorem 7.1 we know that quasi-transitivity Condition 6 implies Conditions 1, 3–8, thus, the presented transformation works for any of those notions of quasi-transitivity. Moreover, if we chose to obtain a generalized Veltman frame with quasi-transitivity Condition 2, the same reasoning applies since as we know by Theorem 8.1, taking the monotone closure does not alter the modal theory of the model. We leave the rest of the details to be worked out by the reader. □

10. From generalized to ordinary

In this chapter we show that given a generalized Veltman model M with quasi-transitivity condition $(i) \in \{3, 4, 5, 6\}$, we can build an ordinary Veltman model M' such that for every world in M we can find a world in M' which is modally equivalent. The definitions and proofs involved in this chapter can be found formalized in Agda in Appendix B.17.

It is worth mentioning that there exists a much simpler transformation which we will present in Chapter 11 and works for the same quasi-transitivity conditions as the transformation presented here. We still believe that this transformation holds value for historical reasons. It was the first transformation from generalized to ordinary models and it was written by Verbrugge in an unpublished manuscript ([36]). In that manuscript there is a comment where the author says that the transformation may also hold for Condition 2 although she has not checked it yet. Unfortunately some steps in the proof do not work if we take a generalized model with quasi-transitivity Condition 2. In [41] a variation of this transformation is presented with the claim that it works for Condition 2. However, as we will comment in Chapter 12, the proof of the claim is in need of repair.

For the rest of this chapter we fix a generalized Veltman model $M := \langle W, R, S, V\rangle$.

We define an ordinary Veltman model $M' := \langle W', R', S', V'\rangle$ where

$$\begin{aligned}
W' :=& \{\langle x, A\rangle : A \subseteq W^2, \\
& \quad (W1)\ \forall \langle u, v\rangle \in A\ \exists Y (x S_u Y, v \in Y), \\
& \quad (W2)\ \forall u \forall V (x S_u V \Rightarrow \exists v \in V (\langle u, v\rangle \in A)\}; \\
R' :=& \{\langle \langle x, A\rangle, \langle y, B\rangle\rangle : xRy, \forall w \forall z (wRx \Rightarrow \langle w, z\rangle \in B \Rightarrow \langle w, z\rangle \in A)\}; \\
S' :=& \{\langle \langle w, C\rangle, \langle x, A\rangle, \langle y, B\rangle\rangle : \langle w, C\rangle R' \langle x, A\rangle, \langle w, C\rangle R' \langle y, B\rangle, \forall v (\langle w, v\rangle \in B \Rightarrow \langle w, v\rangle \in A)\}; \\
V' :=& \{\langle \langle x, A\rangle, p\rangle : \langle x, p\rangle \in V, \langle x, A\rangle \in W', p \in \mathsf{Var}\}.
\end{aligned}$$

Lemma 10.1. The structure $\langle W', R', S', V'\rangle$ is an ordinary Veltman model.

Proof. Here we check that the S'_w is a transitive relation for each $w \in W'$. It is routine to check that the rest of the requirements are satisfied. Assume that we have

$$\langle x, A\rangle S'_{\langle w,D\rangle} \langle y, B\rangle S'_{\langle w,D\rangle} \langle z, C\rangle$$

. We want to show $\langle x, A\rangle S'_{\langle w,D\rangle} \langle z, C\rangle$, thus by the definition of S' we need to prove the following:

1. $\langle w, D\rangle R' \langle x, A\rangle$: it follows from the definition of S' and $\langle x, A\rangle S'_{\langle w,D\rangle} \langle y, B\rangle$;
2. $\langle w, D\rangle R' \langle z, C\rangle$: it follows from the definition of S' and $\langle y, B\rangle S'_{\langle w,D\rangle} \langle z, C\rangle$;
3. $\forall v(\langle w, v\rangle \in C \Rightarrow \langle w, v\rangle \in A)$: from $\langle x, A\rangle S'_{\langle w,D\rangle} \langle y, B\rangle$ and the definition of S' we get that

   $$\forall v(\langle w, v\rangle \in B \Rightarrow \langle w, v\rangle \in A).$$

   Likewise, from $\langle y, B\rangle S'_{\langle w,D\rangle} \langle z, C\rangle$ and the definition of S' we get that

   $$\forall v(\langle w, v\rangle \in C \Rightarrow \langle w, v\rangle \in B).$$

   Then, from the composition of the previous formulas we get the desired fact.

□

We will now introduce two conditions. These conditions offer a convenient way to show that the transformation works for various conditions of quasi-transitivity. In fact, the presented transformation works for a generalized Veltman frame with quasi-transitivity Condition 3, 4, 5 or 6.

Let the conditions (C_0) and (C_1) be defined thus:

$(C_0) := \forall w \forall x \forall V. xS_wV \Rightarrow \exists y \in V. \forall b \forall V'. yS_bV' \Rightarrow \exists v \in V'. (b = w \Rightarrow xS_b\{v\}), (bRw \Rightarrow wS_b\{v\})$;
$(C_1) := \forall w \forall b \forall x \forall V. wRx \Rightarrow xS_bV \Rightarrow \exists v \in V. xS_b\{v\}, (bRw \Rightarrow wS_b\{v\})$.

Theorem 10.2. *If M satisfies both conditions (C_0) and (C_1) then for any world $\langle w, C\rangle \in W'$ and formula D:*

$$w \Vdash D \Leftrightarrow \langle w, C\rangle \Vdash D$$

Proof. We proceed by induction on the formula. Here we only consider the case $D \rhd E$ as the other cases are easy.

- $\boxed{\Rightarrow}$ Assume $w \Vdash D \rhd E$ and let C be such that $\langle w, C\rangle \in W'$. We want to prove $\langle w, C\rangle \Vdash D \rhd E$. Assume that for some $\langle x, A\rangle \in W'$ we have $\langle w, C\rangle R' \langle x, A\rangle \Vdash D$. By IH it follows that $x \Vdash D$ and hence there exists V such that $xS_wV \Vdash E$. By (C_0) there is some $y \in V$ such that

  $$\forall bV'. yS_bV' \Rightarrow \exists v \in V'. (b = w \Rightarrow xS_b\{v\}), (bRw \Rightarrow wS_b\{v\}) \tag{10.1}$$

  We proceed by showing that there is some B such that $\langle x, A\rangle S'_{\langle w, C\rangle} \langle y, B\rangle$. Let B be defined thus:

  $$B := \{\langle u, v\rangle : \exists Y. yS_uY, v \in Y, (u = w \Rightarrow \langle w, v\rangle \in A), (uRw \Rightarrow \langle u, v\rangle \in C)\}$$

  To show $\langle y, B\rangle \in W'$ we need to prove that $(W1)$ and $(W2)$ hold. The condition $(W1)$ follows immediately from the definition of B. To show $(W2)$ assume that for some b and V we have yS_bV. We need to see that there exists $v \in V$ such that $\langle b, v\rangle \in B$. From yS_bV and Eq. (10.1) we get that there exists $v \in V'$ such that

  $$b = w \Rightarrow xS_b\{v\}, \tag{10.2}$$
  $$bRw \Rightarrow wS_b\{v\} \tag{10.3}$$

  To show that $\langle b, v\rangle \in B$ we first see that $b = w \Rightarrow \langle w, v\rangle \in A$. Assume $b = w$, then by Eq. (10.2) it follows that $xS_b\{v\}$ and therefore by condition $(W2)$ for A it follows $\langle b, v\rangle \in A$. We proceed likewise and use Eq. (10.3) to show $bRw \Rightarrow \langle b, v\rangle \in C$. This concludes the proof that $\langle y, B\rangle \in W'$.

  We now check the conditions for $\langle x, A\rangle S'_{\langle w, C\rangle} \langle y, B\rangle$. We already have $\langle w, C\rangle R' \langle x, A\rangle$ by assumption. To see that $\langle w, C\rangle R' \langle y, B\rangle$ we first observe that wRy holds since xS_wV and $y \in V$. Then assume that for some b, z we have bRw and $\langle b, z\rangle \in B$. Then from the definition of B it follows that $\langle b, z\rangle \in C$. The condition $\forall v(\langle w, v\rangle \in B \Rightarrow \langle w, v\rangle \in A)$ follows immediately from the definition of B.

  Finally, since $V \Vdash E$ and $y \in V$ we have $y \Vdash E$ and thus by IH it follows that $\langle y, B\rangle \Vdash E$.

- $\boxed{\Leftarrow}$ We proceed by contraposition. Assume $w \nVdash D \rhd E$, then there exists x such that wRx and

  $$\forall Y(vS_wY \Rightarrow \exists y \in Y(y \nVdash E)). \tag{10.4}$$

Let A be defined thus:

$$A := \{\langle b, v\rangle : (\exists Y . x S_b Y, v \in Y), (b = w \Rightarrow M, v \nVdash E), (bRw \Rightarrow \langle b, v\rangle \in C)\}.$$

We first show that $\langle x, A\rangle \in W'$. Condition $(W1)$ follows directly from the definition of A. To show that $(W2)$ holds assume that for some b and V we have xS_bV. We need to see that for some $v \in V$ we have $\langle b, v\rangle \in A$. Since wRx and xS_bV it follows from condition (C_1) that there exists $v \in V$ such that

$$xS_b\{v\}, \tag{10.5}$$
$$bRw \Rightarrow wS_b\{v\}. \tag{10.6}$$

The first condition to show $\langle b, v\rangle \in A$, namely that $\exists Y . xS_bY, v \in Y$, is met trivially. For the next condition assume $b = w$, then see that we have $xS_w\{v\}$ by Eq. (10.5) and thus by Eq. (10.4) it follows that $v \nVdash E$. For the remaining condition assume bRw, then by Eq. (10.6) we have $wS_b\{v\}$ and thus by $(W2)$ for C we have $\langle b, v\rangle \in C$. Therefore we conclude $\langle b, v\rangle \in A$ and thus $\langle x, A\rangle \in W'$.

To see that $\langle w, C\rangle R' \langle x, A\rangle$ we already have wRx by assumption. The remaining condition, $\forall bz(bRx \Rightarrow \langle b, z\rangle \in A \Rightarrow \langle b, z\rangle \in C)$, follows directly from the definition of A.

Since $x \Vdash D$, it follows from the IH that $\langle x, A\rangle \Vdash D$.

Lastly, assume that for some $\langle y, B\rangle \in W'$ we have $\langle x, A\rangle S'_{\langle w, C\rangle} \langle y, B\rangle$. By definition of S' we have xS_wy and thus wRy. By quasi-reflexivity of S we then have $yS_w\{y\}$ and thus by $(W2)$ for B we have $\langle w, y\rangle \in B$. By definition of S' we also have that $\forall v(\langle w, v\rangle \in B \Rightarrow \langle w, v\rangle \in A)$, hence $\langle w, y\rangle \in A$. By definition of A it follows that $y \nVdash E$ and by IH we have $\langle y, B\rangle \nVdash E$, which concludes the proof.

□

Theorem 10.3. *If a generalized Veltman frame satisfies quasi-transitivity Condition 3, 4, 5 or 6, then it satisfies conditions (C_0) and (C_1).*

Proof. Here we prove the property for a generalized Veltman frame satisfying quasi-transitivity Condition 3. Conditions 4–6 imply Condition 3 as shown in Theorem 7.1.

Assume F is a generalized Veltman frame satisfying quasi-transitivity Condition 3. It is easy to observe that the following property holds:

$$uS_xY \Rightarrow \exists y \in Y\, \forall z(yS_x\{z\} \Rightarrow uS_x\{z\}). \tag{10.7}$$

- $\boxed{(C_0)}$ Assume that for some w, x, V we have xS_wV. Then by Eq. (10.7) there is some $y \in V$ such that
$$\forall z(yS_w\{z\} \Rightarrow xS_w\{z\}). \tag{10.8}$$
Now assume that for some b, V' we have yS_bV'. It follows by Eq. (10.7) that there is some $v \in V'$ such that
$$\forall z(vS_b\{z\} \Rightarrow yS_b\{z\}). \tag{10.9}$$
Assume that $b = w$, we need to see that $xS_b\{v\}$. From xS_wV and $y \in V$ it follows that wRy. Then by quasi-reflexivity we have $yS_w\{y\}$ and by Eq. (10.8) we get $xS_w\{v\}$ which is the same as $xS_b\{v\}$. Assume that bRw, we need to see that $wS_b\{v\}$. From $bRwRy$ we have $wS_b\{y\}$ and from property Eq. (10.7) we get
$$\forall z(yS_b\{z\} \Rightarrow wS_b\{z\}). \tag{10.10}$$
Then since yS_bV' and $v \in V'$ we have bRv so by quasi-reflexivity we have $vS_b\{v\}$. Finally by Eq. (10.9) we get $yS_b\{v\}$ and by Eq. (10.10) we get $wS_b\{v\}$.

- $\boxed{(C_1)}$ Assume that for some w, b, x, V we have $wRxS_bV$. By Eq. (10.7) it follows that there is some $v \in V$ such that

$$\forall z(vS_b\{z\} \Rightarrow xS_b\{z\}). \tag{10.11}$$

We first see that $xS_b\{v\}$. From xS_bV and $v \in V$ we get bRv and by quasi-reflexivity we get $vS_b\{v\}$. Then by Eq. (10.11) we have $xS_b\{v\}$. Assume bRw, we need to see $wS_b\{v\}$. By quasi-reflexivity we get $vS_b\{v\}$ and by Eq. (10.11) we get $xS_b\{v\}$. By $bRwRx$ we get $wS_b\{x\}$ and thus by Eq. (10.7) we have

$$\forall z(xS_b\{z\} \Rightarrow wS_b\{z\}). \tag{10.12}$$

Finally by $xS_b\{v\}$ and Eq. (10.12) we get $wS_b\{v\}$.

□

11. From generalized to ordinary (a simpler approach)

In this chapter we present a transformation that achieves the same effect as the one presented in Chapter 10. However, the process described here is much simpler as it does not modify the set of worlds. The definitions and proofs in this chapter can be found formalized in Agda in Appendix B.8.

Theorem 11.1. *Let M a generalized Veltman model with quasi-transitivity Condition 3, 4, 5 or 6. By Theorem 7.1 we shall assume without loss of generality that M satisfies quasi-transitivity Condition 3. We remind the reader that the condition reads thus:*

$$uS_xY \Rightarrow \exists\, y \in Y\, \forall Y'(yS_xY' \Rightarrow \exists\, Y''{\subseteq}Y' \wedge uS_xY'').$$

For every $\langle x, u, Y\rangle$ such that uS_xY we fix the y that is highlighted in blue and name it y_{xuY}.

We define $M' := \langle W, R, S', V\rangle$ with $S' := \{\langle w, x, v\rangle : \exists Y.wS_xY \wedge y_{xwY} = v\}$. Then M' is an ordinary Veltman frame. Furthermore models M and M' are modally equivalent, that is, it holds that for every $w \in W$ and $A \in \mathsf{Fm}$ we have

$$M, w \Vdash A \Leftrightarrow M', w \Vdash A.$$

Proof. Here we only check that the transitivity condition holds. It is not hard to check that the rest of the conditions also hold.

Assume that we have $xS'_wyS'_wz$. By definition of S' it follows that we have $xS'_w\{y\}S'_w\{z\}$ and furthermore y and z satisfy quasi-transitivity Condition 3. In particular for y it holds that $\forall Y(yS_wz \Rightarrow \exists Y' \subseteq Y \wedge xS_wY')$. If we set $Y := \{z\}$ then have that there exists $Y' \subseteq \{z\}$ such that xS_wY'. Since Y' must be nonempty we have that $Y' = \{z\}$ and thus $xS_w\{z\}$. Finally by definition of S' we have xS'_wz. $\square$

12. A proof in need of repair

As we all know, mathematical proofs can get long and tedious to follow. It is an art to guide the reader through the key steps of the proof and prevent them from getting lost in the details. In order to achieve that, a resource that is often used is to omit details of trivial claims during the proof. Omitting details saves a lot of time and usually there is no harm in it. However, we may mistakenly believe that something is trivial, whereas in reality it may not be trivial, or in the worst case, it may not even be true. If the mistake is overlooked, we may end up building on top of an inconsistent basis. Needless to say, this is an unacceptable situation which we should try to avoid at all costs. In this chapter we present an example of a published proof where some steps remain unjustified. We discovered these gaps while trying to formalize the proof in Agda. We proceed by giving some context to the proof.

The transformations from a generalized to an ordinary Veltman model given in Chapters 10 and 11 work for simple notions of quasi-transitivity but do not seem to work directly for Condition 2, which is the standard. In [41] a transformation which is claimed to work for Condition 2 is presented. We studied the transformation and started to formalize in Agda the proof of its correctness following the proof of Proposition 2.8 in [41].

Despite our strong efforts to fill in the missing steps, we could not reproduce the original reasoning. Although we did not find an explicit counterexample, the original author agrees with us that the missing steps could render the proof invalid. The validity of Proposition 2.8 of [41] comes into question and until this is resolved, we may consider the problem of finding transformation from a generalized model (with Condition 2) to an ordinary model such that it preserves some structure of the original model to be an open question once again. By "preserves some structure" we mean that we should have a property of the form:

$$M, w \Vdash A \Leftrightarrow M', f(w) \Vdash A \quad \text{for every } w \in W, A \in \mathsf{Fm}.$$

Where f is a map from the set of worlds of M to the set of worlds of M'.

We believe that this example should be taken as a humbling reminder that we are humans and we make mistakes. Even if a proof has been through skilled reviewers from a well established journal it is still suspect of being flawed in some subtle way. For this reason, we believe that computer checked proofs should gain relevance in all fields of logic and mathematics. We know that nowadays proof assistants are far from perfect and usually require a lot of time investment both on learning and in formalizing big scale mathematical proofs. However, the confidence level that they offer certainly outweighs their negatives in some situations.

12.1. The details

In this chapter we present the details to better understand which are the problematic steps in the proof. For that, we copy[1] Definitions 2.5, 2.7 and Propositions 2.6, 2.8 in [41].

The definitions and theorems in this section can be found formalized in Agda in Appendix B.18. Note that Proposition 2.8 does not have a finished proof due to the reasons that we will expose.

Definition 2.5 of [41] Let $F = \langle W, R, \{S_w : w \in W\}\rangle$ be a generalized Veltman frame. Let W' consist of all pairs $\langle v, C\rangle$, where $v \in W$ and C is a set of ordered pairs $\langle x, y\rangle \in W^2$ such that:

[1]with very slim adaptations to accommodate our notation.

1. If xRv, then $\langle x, v\rangle \in C$;
2. for each $x \in W$ and each $V \subseteq W[x]$ such that vS_xV there are $V' \subseteq W[x]$ and $y \in V'$ such that $V \subseteq V'$ and $\langle x, y\rangle \in C$;
3. if $\langle x, y\rangle \in C$, then there is $V \subseteq W[x]$ such that vS_xV and $y \in V$.

We define a relation $R' \subseteq W' \times W'$ by

$$\langle w, A\rangle R' \langle u, B\rangle \text{ iff: } wRu \text{ and } \forall x \forall y (xRw \Rightarrow \langle x, y\rangle \in B \Rightarrow \langle x, y\rangle \in A)$$

We define a relation $S'_{\langle w,A\rangle}$ for each $\langle w, A\rangle \in W'$ by

$$\langle u, B\rangle S'_{\langle w,A\rangle} \langle v, C\rangle \text{ iff: } \langle w, A\rangle R' \langle u, B\rangle \text{ and } \langle w, A\rangle R' \langle v, C\rangle \text{ and } \forall y (\langle w, y\rangle \in C \Rightarrow \langle w, y\rangle \in B)$$

We denote an ordered triple $\langle W', R', \{S'_{w'} : w' \in W'\}\rangle$ by of(F).

Proposition 2.6 of [41] Let M be a generalized Veltman frame, then of(M) is an ordinary Veltman frame.

Proof. The proof is omitted in the original paper and we omit it here too. However, the proof has been formalized in Agda and can be found in Appendix B.18. □

Definition 2.7 of [41] Let $M = \langle F, V\rangle$ be a generalized Veltman model, then we define V' by

$$\langle\langle w, A\rangle, p\rangle \in V' \text{ iff: } \langle w, p\rangle \in V.$$

We denote the model $\langle \text{of}(F), V'\rangle$ by o(M).

Proposition 2.8 of [41]: Let $M = \langle W, R, \{S_w : w \in W\}, V\rangle$ be a generalized Veltman model. Let

$$\text{o}(M) = \langle W', R', \{S'_{w'} \in W'\}, V\rangle.$$

Then for each formula ϕ and each $\langle w, A\rangle \in W'$ we have

$$W, w \Vdash \phi \text{ iff: } \text{o}(M), \langle w, A\rangle \Vdash \phi.$$

Partial proof. The proof goes by induction on the formula ϕ. The only interesting case is $\rhd$. Now, suppose that $w \Vdash \phi \rhd \psi$. We want to show $\langle w, A\rangle \Vdash \phi \rhd \psi$. Assume $\langle w, A\rangle R' \langle u, B\rangle$ and $\langle u, B\rangle \Vdash \phi$. By IH we have $u \Vdash \phi$. Then from $\langle w, A\rangle \Vdash \phi \rhd \psi$ we get that there exists a set of worlds $V_0 \subseteq \{x : wRx\}$ such that uS_wV_0 and $v \Vdash \psi$ for each $v \in V_0$. Let v_0 be any element of the set V_0.

In the proof it is claimed that the following holds:

$$\forall x (xRw \text{ and } xRv_0 \Rightarrow \langle x, v_0\rangle \in A).$$

However, details on why this holds are not given. This is one of the steps that has not yet been repaired neither by us or the author.

Further in the proof it is claimed[2] that the following holds:

$$\langle w, v_0\rangle \in B.$$

We find ourselves again in a situation which we have not been able to justify.

As a closing remark, we want to emphasize that we do not deem this proof as definitely flawed since we have not found a counterexample to the theorem. However, the fact that we were unable to fill the presented gaps in this section hints that the at least the definitions involved in the proof should be tweaked.

[2]In the original paper a stronger property is claimed, but the property highlighted here is the only piece missing.

Part III.

Frame conditions

13. Introduction to principles and frame conditions

An interpretability principle is a schema of modal formulas that carries some special significance. The relevance of a principle mainly stems from two factors. On the one hand we have principles, such as the principle M, that give rise to interpretability logics for certain theories. For instance, ILM is the interpretability logic of PA: $\mathsf{ILM} = \mathsf{IL}(\mathsf{PA})$ ([5, 31]). On the other hand we have principles which are valid in all reasonable arithmetical theories and thus are interesting in the search of an axiomatization for the logic $\mathsf{IL}(\mathrm{All})$. For instance, the series of principles R^{n} and R_{n} ([16]), which we will present in this part, are central to the best known lower bound for $\mathsf{IL}(\mathrm{All})$.

Regardless of the area of interest, finding the frame condition for a principle is always the first step towards studying the nature of the principle. Once we have established a frame condition for a principle, say X, one has more tools when studying, for instance, modal soundness and completeness of the logic ILX. Soundness in itself is already an interesting result, but the usefulness of a soundness theorem grows when it is applied to prove independence results between principles. To prove that two principles X and Y are independent means to show $\mathsf{ILX} \nvdash \mathsf{Y}$ and $\mathsf{ILY} \nvdash \mathsf{X}$, which is done by building countermodels (for instance, see [18]), as usual in modal logics. To prove that a principle does not entail another principle is a necessary step in order to improve the lower bound of $\mathsf{IL}(\mathrm{All})$ since we need to show that the new lower bound does indeed not follow from the previous lower bound. In this thesis however, we are not concerned with independence results and we will only focus on the frame conditions.

As we have briefly mentioned in the overview at the beginning of this thesis, a frame condition captures the relational semantic nature of the principle. To be more precise, assume that we have a principle X which is not necessarily valid in an arbitrary ordinary Veltman frame. A frame condition is a first (or higher) order formula (X) such that for any ordinary Veltman frame F we have:

$$F \vDash (\mathsf{X}) \Leftrightarrow F \Vdash \mathsf{X}.$$

In the expression above, $F \vDash (\mathsf{M})$ denotes that F models the condition (M) in the sense of a first (or higher) order structure. The $F \Vdash \mathsf{X}$ part means that we have $\langle F, V\rangle \Vdash \mathsf{X}$ for any valuation V and any particular instance of X. We define frame conditions for generalized Veltman frames in an analogous way. As a convention, for a principle X we will write (X) and $(\mathsf{X})_{\mathrm{gen}}$ to denote the frame conditions with respect to ordinary and generalized semantics, respectively.

In Chapter 21 we will verify in Agda that there is a mechanical procedure to obtain frame conditions for any axiom schema. However, the generated condition encodes a quantification over valuations in the frame condition and thus it is usually not suitable as an *adequate* frame condition. There is no precise definition on what constitutes an adequate frame condition. Usually one considers a frame condition to be adequate if it is relatively elegant and succinct. Let us continue the discussion by using an example. Consider Löb's axiom given in modal form:

$$\mathsf{L} = \Box(\Box A \to A) \to \Box A.$$

We know from the characterization of GL-frames that the frame condition for L is the following:

The R relation must be transitive and Noetherian.

The same condition expressed in formulas reads as follows (we refer to these formulation as the *original*):

$$\forall x.\forall y.\forall z.xRyRz \Rightarrow xRz \ ;$$

$$\forall \mathbb{S}. \mathbb{S} \neq \emptyset \Rightarrow \exists s \in \mathbb{S} \forall s' (s \not R s').$$

And now observe the generic frame condition that arises from a mechanical procedure[1](we will refer to this formulation as the *generic*):

$$\forall \mathbb{A}. \forall u(wRu \Rightarrow (\forall u'(uRu' \Rightarrow u' \in \mathbb{A})) \Rightarrow u \in \mathbb{A}) \Rightarrow \forall u(wRu \Rightarrow u \in \mathbb{A}).$$

If we compare the original and the generic conditions we see that they are both second order formulas as they quantify over sets: We have $\forall \mathbb{S}$ in the original and $\forall \mathbb{A}$ in the generic. If we do the mental exercise to forget for a moment that we have the concepts of *transitivity* and *Noetherian relation* in our mathematical metalanguage, it is not completely obvious how to objectively justify why the original formulation is preferred in the literature over the generic formulation. However, the case is that we *do* have the concepts of transitivity and Noetherian relations in our commonly used metalanguage and moreover we are used to operating with them. Furthermore, it is easy for us to imagine a transitive relation that has no infinite ascending chains. On the contrary the picture that is described by the generic condition is rather blurry for us. All in all, we see that we can precisely define the minimal requirement (frame validity) for a frame condition, but comparing the usefulness of a frame condition remains open to interpretation. However, we will lean towards conditions which are easy to visualize.

In this part we present a number of principles in conjunction with their respective frame conditions for ordinary semantics as well as generalized semantics. We will prove that all the given conditions satisfy the minimal requirement to be considered frame conditions. Moreover, we attach a diagram representation of most of the presented frame conditions in an attempt to convey their visual value.

[1]The mechanical procedure for Veltman semantics is presented in detail in Chapter 21. Basically, the procedure consists in writing the definition of a frame condition as a second order formula.

14. The principle M

The M principle reads as follows:

$$A \rhd B \to (A \wedge \Box C) \rhd (B \wedge \Box C).$$

The M principle is named after Franco Montagna because the principle appeared during discussions between Franco Montagna and Albert Visser about interpretability logic ([6]).

The theorems of ILM are the set of interpretability principles that are always provable in theories which are Σ_1-sound and have full induction. ([5, 19, 38]). An example of such a theory is PA.

14.1. Ordinary semantics

The frame condition for M for ordinary semantics, which we write as (M), reads as follows:

$$\forall w, x, y, z(xS_w yRz \Rightarrow xRz).$$

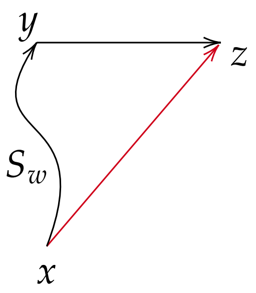


Figure 14.1.: Ordinary frame condition for M.

Theorem 14.1. *For any ordinary frame F, we have that F satisfies the (M) condition iff any model based on F forces every instantiation of the M principle. In symbols:*

$$F \vDash (\mathsf{M}) \Leftrightarrow F \Vdash \mathsf{M}.$$

Proof.

- $\boxed{\Rightarrow}$ Let M be a model based on F and let w be any world. Assume that $w \Vdash A \rhd B$ and that there is a world x such that wRx and $x \Vdash A \wedge \Box C$. Our aim is to find a world z such that $xS_w z \Vdash B \wedge \Box C$. Since $wRx \Vdash A$ and $w \Vdash A \rhd B$ there is a world z such that $xS_w z \Vdash B$. We now show that $z \Vdash \Box C$. Consider an arbitrary u such that zRu. By the frame condition it follows that xRu and we know $x \Vdash \Box C$ hence $u \Vdash C$ and thus $z \Vdash \Box C$. Hence z is the desired world.

- $\boxed{\Leftarrow}$ Let $a, b, c \in \mathsf{Var}$, assume $F \Vdash a \rhd b \to (a \wedge \Box c) \rhd (b \wedge \Box c)$. Assume also that for some x, w, u we have $xS_w zRu$. Our goal is to prove xRu. Consider a model such that the following holds.

$$[\![a]\!] = \{x\};$$
$$[\![b]\!] = \{z\};$$
$$[\![c]\!] = \{v : xRv\}.$$

We observe that $w \Vdash a \rhd b$ because a is only forced in x and we have $xS_w z \Vdash b$. Then it follows that $w \Vdash (a \wedge \Box c) \rhd (b \wedge \Box c)$. It is easy to observe that $x \Vdash a \wedge \Box c$, furthermore we have that by the definition of an ordinary frame $xS_w z \Rightarrow wRx$, hence wRx and thus there must exist some v such that $xS_w v \Vdash b \wedge \Box c$. Since b is only true in z it must be $z \Vdash b \wedge \Box c$. Then, because zRu we have $u \Vdash c$, therefore xRu.

The Agda proof can be found in Appendix B.27. □

14.2. Generalized semantics

The frame condition for M for generalized semantics, which we write as $(\mathsf{M})_{\text{gen}}$, reads as follows:

$$\forall w, x, V(xS_w V \Rightarrow \exists V' \subseteq V(xS_w V', \forall v' \in V' \forall z(v'Rz \Rightarrow xRz))).$$

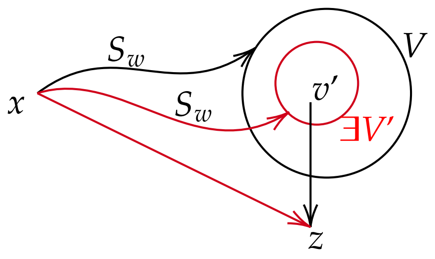


Figure 14.2.: Generalized frame condition for M.

Theorem 14.2. *For any generalized frame F, we have that F satisfies the $(\mathsf{M})_{gen}$ condition iff any model based on F forces every instantiation of the M principle. In symbols:*

$$F \vDash (\mathsf{M})_{gen} \Leftrightarrow F \Vdash M.$$

Proof.

- $\boxed{\Rightarrow}$ Let M be a model based on F and let w be any world. Assume that $w \Vdash A \rhd B$ and that there is a world x such that wRx and $x \Vdash A \wedge \Box C$. Our aim is to find a set Z such that $xS_w Z \Vdash B \wedge \Box C$. Since $wRx \Vdash A$ and $w \Vdash A \rhd B$ there is set Z such that $xS_w Z \Vdash B$. Then by the $(\mathsf{M})_{\text{gen}}$ condition it follows that there is a set $Z' \subseteq Z$ such that $xS_w Z'$ and $\forall v \in Z' \forall z(vRz \Rightarrow xRz)$. Now we show $Z' \Vdash \Box C$. Let $v \in Z'$ and u such that vRu, by the condition above it follows xRu and since $x \Vdash \Box C$ we have $u \Vdash C$. Hence Z' is the desired set.

- $\boxed{\Leftarrow}$ Let $a, b, c \in \mathsf{Var}$ and assume $F \Vdash a \rhd b \to (a \wedge \Box c) \rhd (b \wedge \Box c)$ and $uS_w V$. Consider a model satisfying the following

$$[\![a]\!] = \{u\};$$
$$[\![b]\!] = V;$$
$$[\![c]\!] = \{v : uRv\}.$$

We see that $w \Vdash a \rhd b$ since a is only true in u and we have $uS_w V \Vdash b$. It follows that $w \Vdash (a \wedge \Box c) \rhd (b \wedge \Box c)$. It is easy to see that $u \Vdash a \wedge \Box c$, hence there must exist V' such that $uS_w V' \Vdash b \wedge \Box c$. Clearly $V' \subseteq V$ since b is forced exactly in V. Now let v', z such that $v' \in V'$ and $v'Rz$. Since $v' \Vdash \Box c$, then $z \Vdash c$ and thus uRz. Therefore V' is the desired set.

The Agda proof can be found in Appendix B.9. □

15. The principle $\mathsf{M_0}$

The $\mathsf{M_0}$ principle reads as follows:

$$A \rhd B \to \Diamond A \land \Box C \rhd B \land \Box C.$$

The $\mathsf{M_0}$ principle first appears in [40], where it is proved that $\mathsf{M_0}$ is arithmetically sound. Moreover it is claimed that Dick de Jongh showed that $\mathsf{ILM_0W} = \mathsf{ILW}^*$. During a short period of time, $\mathsf{ILM_0}$ was a candidate to be $\mathsf{IL}(\mathrm{All})$, however, such possibility was ruled out with the discovery of the principle $\mathsf{P_0}$. The logic $\mathsf{ILM_0}$ is complete with respect to ordinary and generalized semantics ([15]).

15.1. Ordinary semantics

The $(\mathsf{M_0})$ condition reads as follows:

$$\forall w, x, y, z(wRxRyS_w z \Rightarrow \forall u(zRu \Rightarrow xRu)).$$

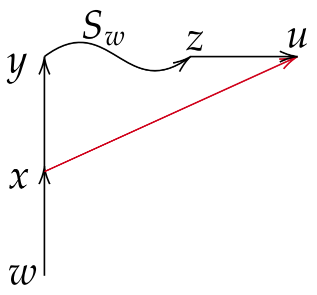


Figure 15.1.: Ordinary frame condition for $\mathsf{M_0}$.

Theorem 15.1. *For any ordinary frame F, we have that F satisfies the $(\mathsf{M_0})$ condition iff any model based on F forces every instantiation of the $\mathsf{M_0}$ principle. In symbols:*

$$F \vDash (\mathsf{M_0}) \Leftrightarrow F \Vdash \mathsf{M_0}.$$

Proof.

- $\boxed{\Rightarrow}$ Let M be a model based on F and let w be any world. Assume that $w \Vdash A \rhd B$ and that there exists some x such that $wRx \Vdash \Diamond A \land \Box C$. It follows that there exists some world y such that $xRy \Vdash A$, then since wRy and $w \Vdash A \rhd B$ there exists a world z such that $yS_w z \Vdash B$. Observe that from $wRxRy$ it follows that $xS_w y$ and by transitivity of S_w and $yS_w z$ we get $xS_w z$. It remains to show $z \Vdash \Box C$. Consider some world u such that zRu, then by the $(\mathsf{M_0})$ condition we have that $\forall u(zRu \Rightarrow xRu)$ and thus it follows that xRu and since $x \Vdash \Box C$ we also have $u \Vdash C$.

- $\boxed{\Leftarrow}$ Let $a, b, c \in \mathsf{Var}$ and assume $F \Vdash a \rhd b \to (\Diamond a \land \Box c) \rhd (b \land \Box c)$ and assume that for some w, x, y, z we have $wRxRyS_w z$. Consider a model based on F such that the following holds:

$$[\![a]\!] = \{y\};$$
$$[\![b]\!] = \{z\};$$
$$[\![c]\!] = \{v : xRv\}.$$

Observe that $w \Vdash a \rhd b$ since a is forced only in y and we have $yS_w z \Vdash b$. It follows that $w \Vdash (\Diamond a \wedge \Box c) \rhd (b \wedge \Box c)$. Clearly $x \Vdash \Diamond a \wedge \Box c$, hence there must exist some world v such that $xS_w v \Vdash b \wedge \Box c$ but since b is only forced in z we have $z = v$ and thus $xS_w z$. To prove the remaining implication let u such that zRu, then $u \Vdash c$ and thus xRu.

The Agda proof can be found in Appendix B.27. $\Box$

15.2. Generalized semantics

The $(\mathsf{M_0})_{\text{gen}}$ condition reads as follows:

$$\forall w, x, y, Y(wRxRyS_w Y \Rightarrow \exists Y' \subseteq Y(xS_w Y', \forall y' \in Y' \forall z(y'Rz \Rightarrow xRz))).$$

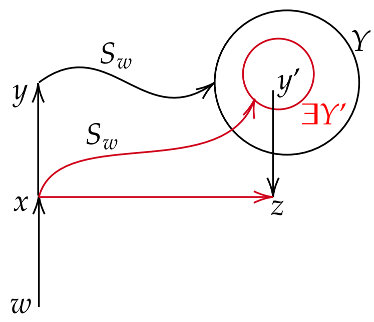


Figure 15.2.: Generalized frame condition for $\mathsf{M_0}$.

Theorem 15.2. *For any ordinary frame F, we have that F satisfies the $(\mathsf{M_0})_{gen}$ condition iff any model based on F forces every instantiation of the $\mathsf{M_0}$ principle. In symbols:*

$$F \vDash (\mathsf{M_0})_{gen} \Leftrightarrow F \Vdash \mathsf{M_0}.$$

Proof.

- $\boxed{\Rightarrow}$ Let M be a model based on F and let w be any world. Assume that $w \Vdash A \rhd B$ and that there is a world x such that $wRx \Vdash \Diamond A \wedge \Box C$. Then there must exist some world y such that $xRy \Vdash A$. Since wRy and $w \Vdash A \rhd B$ there exists some set Y such that $yS_w Y \Vdash B$. Then by the $(\mathsf{M_0})_{\text{gen}}$ condition we have that there exists some $Y' \subseteq Y$ such that $xS_w Y'$ and $(\star)$ $\forall y' \in Y' \forall z(y'Rz \Rightarrow xRz)$. Clearly $Y' \Vdash B$ since $Y' \subseteq Y$. To show that $Y' \Vdash \Box C$ consider some $y' \in Y'$ and some z such that $y'Rz$. Then, by $(\star)$ it follows that xRz and since $x \Vdash \Box C$ we also have $x \Vdash C$.

- $\boxed{\Leftarrow}$ Let $a, b, c \in \mathsf{Var}$ and assume $F \Vdash a \rhd b \to (\Diamond a \wedge \Box c) \rhd (b \wedge \Box c)$ and assume that for some w, x, y, Y we have $wRxRyS_w Y$. Then consider a model based on F such that.

$$[\![a]\!] = \{y\};$$
$$[\![b]\!] = Y;$$
$$[\![c]\!] = \{v : xRv\}.$$

  Observe that $w \Vdash a \rhd b$ as a is only forced in y and we have $yS_w Y \Vdash b$. Consequently it holds that $w \Vdash (\Diamond a \wedge \Box c) \rhd (b \wedge \Box c)$. See also that $x \Vdash \Diamond a$ since $xRy \Vdash a$ and also $x \Vdash \Box c$ by definition of the model. Then there must exist some set Y' such that $xS_w Y' \Vdash b \wedge \Box c$. Clearly $Y' \subseteq Y$ since $Y' \Vdash b$. To show the remaining condition pick some $y' \in Y'$ and some z such that $y'Rz$. Since $Y' \Vdash \Box c$ then $z \Vdash c$ and thus xRz.

The Agda proof can be found in Appendix B.10. $\Box$

16. The principle P_0

The P_0 principle reads as follows:

$$A \rhd \Diamond B \to \Box(A \rhd B).$$

We give some context to the principle P_0, borrowed from [20]. The principle P_0 was discovered in 1998 by Albert Visser. This principle appeared while trying to develop the completeness proof of ILM_0. In an attempt to strengthen the logic, Visser modified the frame condition of ILM_0 to make it stronger and arrived at a stronger principle, which was given the name P_0. The frame condition with respect to ordinary semantics of P_0 implies the frame condition (with respect to ordinary semantics) of M_0. In [18] it is proven that $\mathsf{ILP}_0 \nvdash \mathsf{ILM}_0$ using ordinary Veltman semantics. Thus we have that ILP_0 is modally incomplete with respect to ordinary semantics. However, it was shown recently in [26] that P_0 is in fact complete with respect to generalized Veltman semantics.

The principle P_0 is valid in all reasonable arithmetical theories and thus it is in $\mathsf{IL}(\mathrm{All})$.

16.1. Ordinary semantics

The (P_0) condition reads as follows:

$$\forall w, x, y, z, u(wRxRyS_w zRu \Rightarrow yS_x u).$$

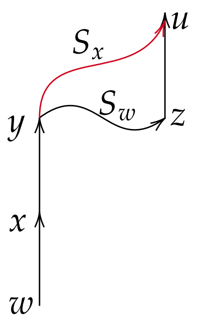


Figure 16.1.: Ordinary frame condition for P_0.

Theorem 16.1. *For any ordinary frame F, we have that F satisfies the (P_0) condition iff any model based on F forces every instantiation of the P_0 principle. In symbols:*

$$F \models (\mathsf{P}_0) \Leftrightarrow F \Vdash \mathsf{P}_0.$$

Proof.

- $\boxed{\Rightarrow}$ Let M be a model based on F and let w be any world. Assume that $w \Vdash A \rhd \Diamond B$ and that there is a world x such that wRx. Our goal is to show that $x \Vdash A \rhd B$. Consider a world y such that $xRy \Vdash A$. As wRy and $w \Vdash A \rhd \Diamond B$ then there exist some worlds z, u such that $yS_w zRu \Vdash B$. By the (P_0) condition it follows that $yS_x u$ and thus $x \Vdash A \rhd B$.

- $\boxed{\Leftarrow}$ Let $a, b \in \mathsf{Var}$ and assume $F \Vdash a \rhd \Diamond b \to \Box(a \rhd b)$ and assume that $wRxRyS_wzRu$. We want to show yS_xu. Consider a model based on F such that:

$$\llbracket a \rrbracket = \{y\};$$
$$\llbracket b \rrbracket = \{u\}.$$

Observe that $w \Vdash a \rhd \Diamond b$ as the only world that forces a is y and we have $yS_wz \Vdash \Diamond b$, because $zRu \Vdash b$. Consequently we have $w \Vdash \Box(a \rhd b)$ and therefore $x \Vdash a \rhd b$. Then, since $xRy \Vdash a$ it follows that there exist some v such that $yS_xv \Vdash b$, but since b is only forced in u, it must be $u = v$ and so yS_xu.

The Agda proof can be found in Appendix B.28. □

16.2. Generalized semantics

The $(\mathsf{P}_0)_{\text{gen}}$ condition reads as follows:

$$\forall w, x, y, V, Z((wRxRyS_wV, \forall v \in Y \exists z \in Z(vRz)) \Rightarrow \exists Z' \subseteq Z(yS_xZ')).$$

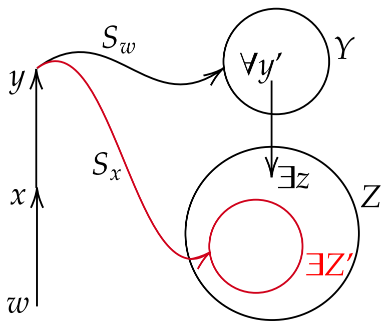


Figure 16.2.: Generalized frame condition for P_0.

Theorem 16.2. *For any generalized frame F, we have that F satisfies the $(\mathsf{P}_0)_{gen}$ condition iff any model based on F forces every instantiation of the P_0 principle. In symbols:*

$$F \vDash (\mathsf{P}_0)_{gen} \Leftrightarrow F \Vdash \mathsf{P}_0.$$

Proof.

- $\boxed{\Rightarrow}$ Let M be a model based on F and let w be any world. Assume that $w \Vdash A \rhd \Diamond B$ and that there is a world x such that wRx. We aim to show that $x \Vdash A \rhd B$. Assume there is a world u such that $xRu \Vdash A$ and as wRu and $w \Vdash A \rhd \Diamond B$ then there exists a set V $uS_xV \Vdash \Diamond B$. Let $\mathbb{B} = \{v : v \Vdash B\}$. Then observe that because $V \Vdash \Diamond B$ we have that for all v in V there exists some $z \in \mathbb{B}$ such that vRz. Hence by the $(\mathsf{P}_0)_{\text{gen}}$ condition there exists some $\mathbb{B}' \subseteq \mathbb{B}$ such that $yS_x\mathbb{B}'$. Clearly $\mathbb{B}' \Vdash B$, therefore $x \Vdash A \rhd B$.

- $\boxed{\Leftarrow}$ Let $a, b \in \mathsf{Var}$ and assume $F \Vdash a \rhd \Diamond b \to \Box(a \rhd b)$ and assume that for some w, x, y, V, Z we have $wRxRyS_wY$ and $(\star)$ $\forall v \in V \exists z \in Z(vRz)$. Consider a model based on F such that:

$$\llbracket a \rrbracket = \{y\};$$
$$\llbracket b \rrbracket = Z.$$

See that $w \Vdash a \rhd \Diamond b$ as the only world that forces a is y and we have yS_wV and by $(\star)$ it follows that $V \Vdash \Diamond b$. Consequently it holds that $w \Vdash \Box(a \rhd b)$ and since wRx then $x \Vdash a \rhd b$. Also, since $xRy \Vdash a$ then there exists Z' such that $yS_xZ' \Vdash b$. Clearly $Z' \Vdash b$ implies $Z' \subseteq Z$ so we are done.

The Agda proof can be found in Appendix B.11. □

17. The principle R

The R principle reads as follows:

$$A \rhd B \to \neg(A \rhd \neg C) \rhd (B \wedge \Box C) \ .$$

The principle R was introduced by Goris and Joosten in [14]. The authors show that R does follow semantically but not syntactically from $\mathsf{ILP_0W^*}$, which was the best known lower bound for IL(All) in at a time. Goris and Joosten also show that R is valid in all reasonable arithmetical theories and thus giving a strictly better lower bound for IL(All).

The principle R is the same as $\mathsf{R_0}$ and $\mathsf{R^0}$. The $\mathsf{R^n}$ and $\mathsf{R_n}$ series of principles, which generalize R, will be discussed in Chapters 18 and 20.

17.1. Ordinary semantics

The (R) condition reads as follows:

$$\forall w, x, y, z, u(wRxRyS_wzRu \Rightarrow yS_xu).$$

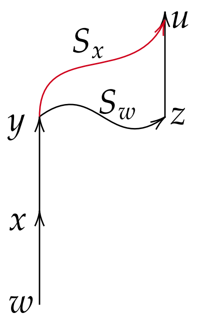


Figure 17.1.: Ordinary frame condition for R.

Theorem 17.1. *For any ordinary frame F, we have that F satisfies the (R) condition iff any model based on F forces every instantiation of the R principle. In symbols:*

$$F \models (\mathsf{R}) \Leftrightarrow F \Vdash \mathsf{R} \ .$$

Proof. • $\boxed{\Rightarrow}$ Let M be a model based on F and let w be any world. Assume that $w \Vdash A \rhd B$ and that there is a world x such that $wRx \Vdash \neg(A \rhd \neg C)$. We need to see that there is some world z such that $xS_wz \Vdash B \wedge \Box C$. From $x \Vdash \neg(A \rhd \neg C)$ we get a world y such that $xRy \Vdash A$ and $(\star)$ $\forall v(yS_xv \Rightarrow v \Vdash C)$. Since $w \Vdash A \rhd B$, and by transitivity we have wRy, it follows that there exists a world z such that $yS_wz \Vdash B$. To see that z is the desired world we first see that $z \Vdash \Box C$. Let u be such that zRu, then by (R) it follows that yS_xu and by $(\star)$ we get $u \Vdash C$. Finally, we have to see that xS_wz. Since $wRxRy$ we have that xS_wy and we have yS_wz from before, hence by transitivity of S_w we get xS_wz.

- $\boxed{\Leftarrow}$ Let $a, b, c \in \mathsf{Var}$ and assume that for some w, x, y, z we have $wRxRyS_wz$. Consider a model based on F that satisfies the following.

$$\begin{aligned}\llbracket a \rrbracket &= \{y\};\\ \llbracket b \rrbracket &= \{z\};\\ \llbracket c \rrbracket &= \{u : yS_xu\}.\end{aligned}$$

  By assumption we have that $w \Vdash a \rhd b \to (\neg(a \rhd \neg c) \rhd (b \wedge \Box c))$. Clearly $w \Vdash a \rhd b$ as we have $yS_wz \Vdash b$. Consequently it holds that $w \Vdash \neg(a \rhd \neg c) \rhd (b \wedge \Box c)$. In order to show that $x \Vdash \neg(a \rhd \neg c)$, considering that a is only forced in y, it suffices to observe that $\forall z(yS_xz \Rightarrow z \Vdash c)$, which clearly holds. Then there must exist some world v such that $xS_wv \Vdash b \wedge \Box c$ but $v = z$ since z is the only world that forces b, hence $xS_wz \Vdash \Box c$. Now to show $\forall v(zRv \Rightarrow yS_xv)$ consider some v such that zRv. From $z \Vdash \Box c$ we get $v \Vdash c$ and thus yS_xv.

The Agda proof can be found in Appendix B.29. □

17.2. Generalized semantics

In order to define the frame condition for the R principle we first need to introduce the concept of choice set.

Definition 17.2. If xRy we say that a set of worlds K is a choice set for $\langle x, y\rangle$ iff for any V such that yS_xV we have $V \cap K \neq \emptyset$. We denote the family of choice sets for $\langle x, y\rangle$ by $\mathcal{C}(x, y)$. Note that this definition depends on the frame, but it should always be clear from context.

The $(\mathsf{R})_{\text{gen}}$ condition reads as follows:

$$\begin{aligned}&\forall w, x, y, Y, K(wRxRyS_wY, K \in \mathcal{C}(x, y)\\ \Rightarrow &\exists Y' \subseteq Y(xS_wY', \forall y' \in Y' \forall z(y'Rz \Rightarrow z \in K))).\end{aligned}$$

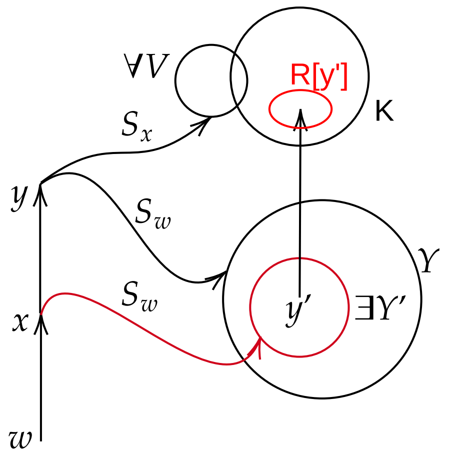


Figure 17.2.: Generalized frame condition for R.

Theorem 17.3. *For any generalized frame F, we have that F satisfies the $(\mathsf{R})_{gen}$ condition iff any model based on F forces every instantiation of the R principle. In symbols:*

$$F \vDash (\mathsf{R})_{gen} \Leftrightarrow F \Vdash \mathsf{R}.$$

Proof.

- $\boxed{\Rightarrow}$ Let M be a model based on F and assume there is a world w such that $w \Vdash A \rhd B$ and a world x such that wRx and $x \Vdash \neg(A \rhd \neg C)$. We need to show that there is a set Z such that $xS_wZ \Vdash B \wedge \Box C$. From $x \Vdash \neg(A \rhd \neg C)$ it follows that there is a world y such that $xRy \Vdash A$ and $(\star)$ $\forall V(yS_xV \Rightarrow \exists c \in V(c \Vdash C))$. Consider the set $K := \{c : c \Vdash C, \exists V(c \in V, yS_xV)\}$. Clearly by $(\star)$ it follows that K is a choice set for $\langle x, y\rangle$. By transitivity of R we get wRy and since $w \Vdash A \rhd B$ then there must exist some Y such that $yS_wY \Vdash B$. We can now apply the $(\mathsf{R})_{\text{gen}}$ condition and get a $Y' \subseteq Y$ such that xS_wY' and $(\dagger)$ $\forall y' \in Y' \forall z(y'Rz \Rightarrow z \in K)$. To show that Y' is the desired set it remains to see that $Y' \Vdash B \wedge \Box C$. From the fact that $Y' \subseteq Y \Vdash B$ it easily follows that $Y' \Vdash B$. Now, let $y' \in Y'$ and u such that $y'Ru$, from $(\dagger)$ we get $u \in K$ and by definition of K we have $u \Vdash C$.

- $\boxed{\Leftarrow}$ Let $a, b, c \in \mathsf{Var}$ and assume $F \Vdash a \rhd b \to \neg(a \rhd \neg c) \rhd (b \wedge \Box c)$. Assume also that for some w, x, y, Y, K we have $wRxRyS_wY, K \in \mathcal{C}(x, y)$. Now consider a model based on F that satisfies the following:

$$\begin{aligned} [\![a]\!] &= \{y\}; \\ [\![b]\!] &= Y; \\ [\![c]\!] &= K. \end{aligned}$$

  By assumption we have $w \Vdash a \rhd b \to \neg(a \rhd \neg c) \rhd (b \wedge \Box c)$. Observe that that $w \Vdash a \rhd b$ since $yS_wY \Vdash b$. Thus $w \Vdash \neg(a \rhd \neg c) \rhd (b \wedge \Box c)$. As y is the only world that forces a, in order to show $x \Vdash \neg(a \rhd \neg c)$ we need to see that $\forall V(yS_xV \Rightarrow \exists z \in V(z \Vdash c))$, which is equivalent to $\forall V(yS_xV \Rightarrow \exists z \in V \cap K)$ and this holds since $K \in \mathcal{C}(x, y)$. As a consequence of $x \Vdash \neg(a \rhd \neg c)$ we have that there exists a set Y' such that $xS_wY' \Vdash b \wedge \Box c$. From $Y' \Vdash b$ we get $Y' \subseteq Y$ and from $Y' \Vdash \Box c$ we get $\forall y' \in Y'(\forall z(y'Rz \to z \in K))$, hence Y' is the desired set.

The Agda proof can be found in Appendix B.12. □

18. The principle $\mathsf{R_1}$

The R_1 principle reads as follows:

$$A \rhd B \to (\neg(A \rhd \neg C) \wedge (D \rhd \Diamond E)) \rhd (B \wedge \Box C \wedge (D \rhd E)).$$

It is the second principle of the $\mathsf{R_n}$ series. The series of principles $\mathsf{R_n}$ is defined in [16] and it has been named *the slim series*.

18.1. Ordinary semantics

The $(\mathsf{R_1})$ frame condition reads as follows:

$$\forall w, x, y, z(wRxRyS_w z \Rightarrow \forall u(zRu \Rightarrow yS_x u, \forall v(uS_x v \Rightarrow \forall m(vRm \Rightarrow uS_z m))))$$

Theorem 18.1. *For any ordinary frame F, we have that F satisfies the $(\mathsf{R_1})$ condition iff any model based on F forces every instantiation of the $\mathsf{R_1}$ principle. In symbols:*

$$F \vDash (\mathsf{R_1}) \Leftrightarrow F \Vdash \mathsf{R_1}.$$

Proof. The details of the proof can be found in [16].

$\Box$

18.2. Generalized semantics

The results presented in this section are a joint work with Luka Mikec.

We start by giving some definitions:

1. $R^{-1}[E] := \{x : \exists y \in E . xRy\}$. E denotes a set.
2. $R_x^{-1}[E] := R^{-1}[E] \cap R[x]$. E denotes a set.

The $(\mathsf{R_1})_{\text{gen}}$ condition reads as follows:

$$\begin{aligned}
&\forall w, x, u, \mathbb{B}, \mathbb{C}, \mathbb{E}.\\
&\quad wRxRuS_w\mathbb{B}, \mathbb{C} \in \mathcal{C}(x,u)\\
&\Rightarrow \exists \mathbb{B}' \subseteq \mathbb{B}. xS_w\mathbb{B}', R[\mathbb{B}'] \subseteq \mathbb{C}, \forall v \in \mathbb{B}'. \forall c \in \mathbb{C}.(\exists U \subseteq R_x^{-1}[\mathbb{E}], vRcS_xU) \Rightarrow \exists \mathbb{E}' \subseteq \mathbb{E}. cS_v\mathbb{E}'\Big)
\end{aligned}$$

Theorem 18.2. *For any generalized frame F, we have that F satisfies the $(\mathsf{R_1})_{gen}$ condition iff any model based on F forces every instantiation of the $\mathsf{R_1}$ principle. In symbols:*

$$F \vDash (\mathsf{R_1})_{gen} \Leftrightarrow F \Vdash \mathsf{R_1}.$$

Proof.
- $\boxed{\Rightarrow}$ Let's fix the model and let $w \in W$ be arbitrary. Suppose $w \Vdash A \rhd B$, and let x be such that wRx and $x \Vdash \neg(A \rhd \neg C) \wedge (D \rhd \Diamond E)$. It follows from $x \Vdash \neg(A \rhd \neg C)$ that there exists u such that xRu, such that $u \Vdash A$, and for every Z such that uS_xZ there is some $c_Z \in Z$ such that $c_Z \Vdash C$. From wRu, $w \Vdash A \rhd B$ and $u \Vdash A$ follows in particular that there is a $\mathbb{B}$, $uS_w\mathbb{B} \Vdash B$. Let $\mathbb{C} := \{c_Z : uS_xZ\}$. It is easy to check that $\mathbb{C} \in \mathcal{C}(x,u)$.

Let $\mathbb{E} := [\![E]\!]$. For the selected $w, x, u, \mathbb{B}, \mathbb{C}, \mathbb{E}$ the property $(\mathsf{R}_1)_{\text{gen}}$ implies that there exists $\mathbb{B}' \subseteq \mathbb{B}$ such that:

$$xS_w\mathbb{B}', R[\mathbb{B}'] \subseteq \mathbb{C}, \forall v \in \mathbb{B}'.\forall c \in \mathbb{C}.(\exists U \subseteq R_x^{-1}[\mathbb{E}], vRcS_xU) \Rightarrow \exists \mathbb{E}' \subseteq \mathbb{E}.cS_v\mathbb{E}'$$

We have that $\mathbb{B}' \Vdash B$ since $\mathbb{B}' \subseteq \mathbb{B}$ and $\mathbb{B}' \Vdash \Box C$ since $R[\mathbb{B}'] \subseteq \mathbb{C}$. We now show that $\mathbb{B}' \Vdash D \rhd E$. Let $v \in B'$ and assume that for some c such that vRc we have $c \Vdash D$. From earlier we have $x \Vdash D \rhd \Diamond E$. Since $c \in R[\mathbb{B}'] \subseteq C \subseteq R[x]$, then xRc so it follows that there exists U such that cS_xU and $U \Vdash \Diamond E$. Clearly $U \subseteq R_x^{-1}[\mathbb{E}]$ so by the above property there exists $\mathbb{E}' \subseteq \mathbb{E}$ such that $cS_v\mathbb{E}'$. Because $\mathbb{E}' \subseteq \mathbb{E}$ we have $\mathbb{E}' \Vdash E$.

- $\boxed{\Leftarrow}$ Assume for a contradiction that $F \nVdash (\mathsf{R}_1)_{\text{gen}}$. It follows that there exist $w, x, u, \mathbb{B}, \mathbb{C}, \mathbb{E}$ such that $wRxRuS_w\mathbb{B}$, $\mathbb{C} \in \mathcal{C}(x, u)$ and:

  $$\forall \mathbb{B}' \subseteq \mathbb{B}.xS_w\mathbb{B}', R[\mathbb{B}'] \subseteq \mathbb{C} \Rightarrow \exists v \in \mathbb{B}'.\exists c \in \mathbb{C}.\exists Z \subseteq R_x^{-1}[\mathbb{E}].vRcS_xZ, \forall \mathbb{E}' \subseteq \mathbb{E}.c\not{S}_v\mathbb{E}'.$$

  Let $\mathcal{V}$ be a family of sets defined thus:

  $$\mathcal{V} := \{U : U \subseteq \mathbb{B}, xS_wU, R[U] \subseteq \mathbb{C}\}.$$

  From the condition it follows that for every $U \in \mathcal{V}$ the following is valid:

  $$\exists v_U \in U.\exists c_U \in \mathbb{C}.(\exists Z_U \subseteq R_x^{-1}[\mathbb{E}](v_URc_US_xZ_U, \forall \mathbb{E}' \subseteq \mathbb{E}.c_U\not{S}_{v_U}\mathbb{E}')).$$

  Let us fix such v_U and c_U and Z_U for all $U \in \mathcal{V}$.

  Define a valuation such that the following applies:

  $$\begin{aligned} [\![a]\!] &= \{u\}; \\ [\![b]\!] &= \mathbb{B}; \\ [\![c]\!] &= \mathbb{C}; \\ [\![d]\!] &= \{c_U : U \in \mathcal{V}\}; \\ [\![e]\!] &= \mathbb{E}. \end{aligned}$$

  By assumption we have $w \Vdash a \rhd b \to (\neg(a \rhd \neg c) \wedge (d \rhd \Diamond e)) \rhd (b \wedge \Box c \wedge (d \rhd e))$.

  It is easy to see that $w \Vdash a \rhd b$ and $x \Vdash \neg(a \rhd \neg c)$.

  Let us prove $x \Vdash d \rhd \Diamond e$. Let $xRc \Vdash D$. Then $c = c_U$ for some $U \in \mathcal{V}$. From the definition of c_U we have that $c_US_xZ_U$. The valuation is defined such that e is true exactly on the set $\mathbb{E}$. Hence $R_x^{-1}[\mathbb{E}] \Vdash \Diamond e$ and since $Z_U \subseteq R_x^{-1}[\mathbb{E}]$ it follows that $x \Vdash d \rhd \Diamond e$.

  We can also check that for every $U \in \mathcal{V}$ we have $U \Vdash b \wedge \Box c$. Furthermore, for any set U we have

  $$(\star)\ xS_wU \Vdash b \wedge \Box c \Rightarrow U \in \mathcal{V}.$$

  Since $w \Vdash a \rhd b$ and $wRx \Vdash \neg(a \rhd \neg c) \wedge (d \rhd \Diamond e)$ there must exist some set U such that $xS_wU \Vdash b \wedge \Box c \wedge (d \rhd e)$. From $(\star)$ follows that $U \in \mathcal{V}$ hence there exist v_U, c_U, Z_U such that $Z_U \subseteq R_x^{-1}[\mathbb{E}]$ and $v_URc_US_xZ_U, (\forall \mathbb{E}' \subseteq \mathbb{E})c_U\not{S}_{v_U}\mathbb{E}'$. Since $c_U \Vdash d$ there must exist some Y such that $c_US_{v_U}Y \Vdash e$, however, by the definition of the valuation it follows that $Y \subseteq \mathbb{E}$ and thus $c_U\not{S}_{v_U}Y$, which is a contradiction.

The Agda proof can be found in Appendix B.16. $\square$

19. The principle R^1

The R^1 principle reads as follows:

$$A \rhd B \to (\Diamond\neg(D \rhd \neg C) \wedge (D \rhd A)) \rhd (B \wedge \Box C).$$

R^1 is the second principle of the series R^n. The series of principles R_n is defined in [16] and it has been named *the broad series*. In Chapter 20 we comment on the series R^n and present the frame condition with respect to generalized semantics.

19.1. Generalized semantics

The $(\mathsf{R}^1)_{\text{gen}}$ condition reads as follows:

$$\begin{aligned}
&\forall w, x, y, z, \mathbb{A}, \mathbb{B}, \mathbb{C}, \mathbb{D}.\\
&\quad wRxRyRz,\\
&\quad (\forall u.wRu, u \in \mathbb{A} \Rightarrow \exists V.uS_wV, V \subseteq \mathbb{B}),\\
&\quad (\forall u.xRu, u \in \mathbb{D} \Rightarrow \exists V.uS_xV, V \subseteq \mathbb{A}),\\
&\quad (\forall V.zS_yV \Rightarrow \exists v \in V.v \in \mathbb{C}),\\
&\quad z \in \mathbb{D}\\
&\Rightarrow \exists V \subseteq \mathbb{B}(xS_wV, R[V] \subseteq \mathbb{C}).
\end{aligned}$$

Theorem 19.1. *For any generalized frame F, we have that F satisfies the $(\mathsf{R}^1)_{gen}$ condition iff any model based on F forces every instantiation of the R^1 principle. In symbols:*

$$F \vDash (\mathsf{R}^1)_{gen} \Leftrightarrow F \Vdash \mathsf{R}^1.$$

Proof.

- $\boxed{\Rightarrow}$ Fix a model M and a world w, we are to prove that $w \Vdash A \rhd B \to (\Diamond\neg(D \rhd \neg C) \wedge (D \rhd A)) \rhd (B \wedge \Box C)$. For that assume that $w \Vdash A \rhd B$ and that for some x, y, z we have $wRxRyRz$ and satisfy the conditions on the left hand side of the implication in the $(\mathsf{R}^1)_{\text{gen}}$ condition. From that we derive that $x \Vdash D \rhd A$, $y \Vdash \neg(D \rhd \neg C)$ and $z \Vdash D$. Now let $\mathbb{A} := \{w : w \Vdash A\}$. We define $\mathbb{B}, \mathbb{C}, \mathbb{D}$ likewise for formulas B, C, D respectively. It is routine to check that the left part of the implication of $(\mathsf{R}^1)_{\text{gen}}$ is met. Hence there exist a set $V \subseteq \mathbb{B}$ such that xS_wV and $R[V] \subseteq \mathbb{C}$. By the definition of the sets $\mathbb{B}$ and $\mathbb{C}$ it follows that $V \Vdash B \wedge \Box C$.

- $\boxed{\Leftarrow}$ Fix a frame F and let a, b, c, d be propositional variables and assume $F \Vdash a \rhd b \to (\Diamond\neg(d \rhd \neg c) \wedge (d \rhd a)) \rhd (b \wedge \Box c)$. Assume that the left part of the implication of $(\mathsf{R}^1)_{\text{gen}}$ holds. Now consider a model extending F such that:

$$\begin{aligned}
[\![a]\!] &= \mathbb{A},\\
[\![b]\!] &= \mathbb{B},\\
[\![c]\!] &= \mathbb{C},\\
[\![d]\!] &= \mathbb{D}.
\end{aligned}$$

  Now one can easily check that $w \Vdash A \rhd B$, $x \Vdash \Diamond\neg(D \rhd \neg C) \wedge (D \rhd A)$, hence there exists U such that xS_wU and $U \Vdash B \wedge \Box C$. From that we derive that $U \subseteq \mathbb{B}$ and $R[U] \subseteq \mathbb{C}$.

The Agda proof can be found in Appendix B.13. □

20. The series of principles $\mathsf{R^n}$

The series of principles $\mathsf{R^n}$ principle is defined thus:

$$\begin{aligned} U_0 &:= \Diamond\neg(D_0 \rhd \neg C) \\ U_{r+1} &:= \Diamond((D_r \rhd D_{r+1}) \land U_r) \end{aligned}$$

$$\begin{aligned} R^0 &:= A \rhd B \to \neg(A \rhd \neg C) \rhd B \land \Box C \\ R^{n+1} &:= A \rhd B \to ((D_n \rhd A) \land U_n) \rhd B \land \Box C \end{aligned}$$

The principle $\mathsf{R^0}$ is equivalent to R, which we already discussed in Chapter 17 and we already discussed the principle $\mathsf{R^1}$ in Chapter 19. In this chapter we deal with these and other principles from the series at once.

The $\mathsf{R^n}$ series is also referred to as *the broad series*.

20.1. Ordinary semantics

The frame condition for ordinary semantics ($\mathsf{R^n}$) can be found in [16].

20.2. Generalized semantics

The $(\mathsf{R^n})_{\text{gen}}$ condition reads as follows:

$$\begin{aligned} &\forall w, x_0, ..., x_{n-1}, y, z, \mathbb{A}, \mathbb{B}, \mathbb{C}, \mathbb{D}_0, ..., \mathbb{D}_{n-1}. \\ &\quad wRx_{n-1}R...Rx_0RyRz, \\ &\quad (\forall u.wRu, u \in \mathbb{A} \Rightarrow \exists V.uS_wV \subseteq \mathbb{B}), \\ &\quad (\forall u.x_{n-1}Ru \in \mathbb{D}_{n-1} \Rightarrow \exists V.uS_{x_{n-1}}V \subseteq \mathbb{A}), \\ &\quad (\forall i \in \{1, ..., n-2\} \forall u.x_{\mathrm{i}}Ru \in \mathbb{D}_i \Rightarrow \exists V.uS_{x_i}V \subseteq \mathbb{D}_{i+1}), \\ &\quad (\forall V.zS_yV \Rightarrow V \cap \mathbb{C} \neq 0), \\ &\quad z \in \mathbb{D}_0 \\ &\Rightarrow \exists V \subseteq \mathbb{B}.x_{n-1}S_wV, R[V] \subseteq \mathbb{C}. \end{aligned}$$

Lemma 20.1. Let M be a model, let x be a world of M and let $n \in \mathbb{N}$. For any $i \leq n$ we have that if $M, x \Vdash U_i$ then there exist some worlds $y, z, x_0, ..., x_i$ such that:

1. $x_{\mathrm{i}} = x$;
2. $x_iR...Rx_0RyRz$;
3. for all $j \leq i$ we have that $M, x_j \Vdash U_j$;
4. for all $j < i$ we have that $M, x_j \Vdash D_j \rhd D_{j+1}$;
5. for all V we have that if zS_yV then $V \cap \{w : M, w \Vdash C\} \neq \emptyset$;
6. $M, z \Vdash D_0$.

Proof. By induction on i.

- For $i = 0$ we have that $x \Vdash \Diamond\neg(D_0 \rhd \neg C)$. It follows that there exists some y such that $xRy \Vdash \neg(D_0 \rhd \neg C)$ and therefore there exists some z such that $yRz \Vdash D_0$ and for any V, if zS_yV, then $V \cap \{w : M, w \Vdash C\} \neq \emptyset$. It is clear that all claims are met.

- For $i+1$ we have that $x \Vdash \Diamond((D_i \rhd D_{i+1}) \wedge U_i)$. It follows that there exists some x_i such that $x_i \Vdash D_i \rhd D_{i+1} \wedge U_i$. By IH there exist $y, z, x_0, ..., x_i$ such that they satisfy claims $1, ..., 6$. We set $x_{i+1} := x$. It is trivial to observe that by using the IH all conditions are met for $i+1$.

$\square$

Theorem 20.2. *For any generalized frame F, we have that F satisfies the $(\mathsf{R^n})_{gen}$ condition iff any model based on F forces every instantiation of the $\mathsf{R^n}$ principle. In symbols:*

$$F \vDash (\mathsf{R^n})_{gen} \Leftrightarrow F \Vdash \mathsf{R^n}.$$

Proof. If $n = 0$ we refer to Theorem 17.3. For $n+1$ proceed as follows.

- $\boxed{\Rightarrow}$ Fix a model and assume that for some world w we have $w \Vdash A \rhd B$. Then assume also that $wRx \Vdash ((D_\mathrm{n} \rhd A) \wedge U_n)$. Our goal is to find a set V such that $xS_wV \Vdash B \wedge \Box C$. By Lemma 20.1 it follows that there exist $y, z, x_0, ..., x_n$ satisfying $1, ..., 6$. Then let $\mathbb{A} := [\![A]\!]$, $\mathbb{B} := [\![B]\!]$, $\mathbb{C} := [\![C]\!]$ and for $i \leq n$ let $\mathbb{D}_\mathrm{i} := [\![D_\mathrm{i}]\!]$.

  It is routine to check that the left part of the $(\mathsf{R^{n+1}})_{\mathrm{gen}}$ holds and thus we get that there exists some $V \subseteq \mathbb{B}$ such that x_nS_wV and $R[V] \subseteq \mathbb{C}$. Since $V \subseteq \mathbb{B}$ we have that $V \Vdash B$ and since $R[V] \subseteq \mathbb{C}$ we have $V \Vdash \Box C$. Finally, since $x_n = x$ we conclude $xS_wV \Vdash B \wedge \Box C$.

- $\boxed{\Leftarrow}$ Fix a frame F and let $a, b, c, d_0, ..., d_\mathrm{n}$ be propositional variables and assume $F \Vdash R^{n+1}$. Assume that the left part of the implication of $(\mathsf{R^{n+1}})_{\mathrm{gen}}$ holds. Now consider a model based on F that satisfies the following:

$$\begin{aligned}
[\![a]\!] &= \mathbb{A};\\
[\![b]\!] &= \mathbb{B};\\
[\![c]\!] &= \mathbb{C};\\
[\![d_\mathrm{i}]\!] &= \mathbb{D}_\mathrm{i}, \text{ for all } i \in \{0...n\}.
\end{aligned}$$

  Now one can routinely check that $w \Vdash A \rhd B$ and $x \Vdash ((D_n \rhd A) \wedge U_n)$, hence there exists U such that xS_wU and $U \Vdash B \wedge \Box C$. From that we derive that $U \subseteq \mathbb{B}$ and $R[U] \subseteq \mathbb{C}$.

The Agda proof can be found in Appendix B.15. $\square$

21. Generic frame condition

In this chapter we present a method that given a formula A, builds a second order formula that is a generalized frame condition for A.

Definition 21.1. Generic frame condition Given a generalized frame $F = \langle W, R, S \rangle$ and a formula A with $\mathsf{Var}(A) = \{x_1, ..., x_n\}$. Consider n second order variables $\mathbb{X}_1, ..., \mathbb{X}_n$ which range over sets of worlds. We write $\mathbb{X}_*$ instead of $\mathbb{X}_1, ..., \mathbb{X}_n$. Let $\mathcal{F}$ be defined by:

$$\mathcal{F} : \underbrace{\mathcal{P}(W) \times \cdots \times \mathcal{P}(W)}_{n} \times \mathsf{Fm} \to \mathcal{P}(W)$$

$$\begin{aligned}
\mathcal{F}(\mathbb{X}_*, x_i) &:= \mathbb{X}_i;\\
\mathcal{F}(\mathbb{X}_*, \bot) &:= \emptyset;\\
\mathcal{F}(\mathbb{X}_*, A \to B) &:= \{w : w \in \mathcal{F}(\mathbb{X}_*, A) \Rightarrow w \in \mathcal{F}(\mathbb{X}_*, B)\};\\
\mathcal{F}(\mathbb{X}_*, A \rhd B) &:= \{w : \forall u.(wRu, u \in \mathcal{F}(\mathbb{X}_*, A)) \Rightarrow \exists Y.uS_wY \subseteq \mathcal{F}(\mathbb{X}_*, B))\}.
\end{aligned}$$

Finally we define the condition $(\mathsf{A}^*)_{\text{gen}}$ thus:

$$(\mathsf{A}^*)_{\text{gen}} := \forall \mathbb{X}_* \forall w.w \in \mathcal{F}(\mathbb{X}_*, A).$$

The following theorem shows that the previous definition gives a valid frame condition.

Theorem 21.2. *Let A be a formula. For any generalized frame F, we have that F satisfies the $(\mathsf{A}^*)_{gen}$ condition iff any model based on F forces A. In symbols:*

$$F \vDash (\mathsf{A}^*)_{gen} \Leftrightarrow F \Vdash A.$$

Proof. The proof goes by an easy induction on the formula.
The Agda proof can be found in Appendix B.7. $\square$

Remark 21.3. For instance, if we want the frame condition for P_0 we would look at

$$((\mathsf{a} \rhd \Diamond \mathsf{b} \to \Box(\mathsf{a} \rhd \mathsf{b}))^*)_{\text{gen}}.$$

Where a, b are different propositional variables.

It is easy to see that the presented method can be adapted to generate a frame condition for ordinary semantics.

We suggest a future systematic study that lays down the relations between generic frame conditions and the ones used in practice.

Part IV.

The logic of Agda

The purpose of this part is to give a gradual introduction to Agda.

In Chapter 22 we will give a short introduction to untyped lambda calculus. Then we will proceed with simply typed lambda calculus. The important takeaway of this chapter is to observe how we can use types to rule out bogus terms.

In Chapter 23 we will explain Martin Löf's logical framework. This system features dependent types and thus is a suitable system to formalize propositions and proofs. This system is especially important because it is the basis of Agda's type theory. We will see how we can embed intuitionistic propositional logic into the system and how we can prove some simple properties about natural numbers.

In Chapter 24 we give an approximated specification of the Agda language. We must emphasize that it is an *approximation* because there is no formal specification of Agda. The one we present has been built from the official documentation ([3]), Ulf Norell's[1] PhD thesis ([29]) and personal experience. Furthermore, by no means we attempt to describe all the language. Rather, we focus on the parts which are more important to understand our implementation, which we will present in Part V.

In Chapter 25 we present an introductory tutorial to Agda. The spirit of this tutorial is to prioritize intuition over formalization and precision. The informal format is the one in which Agda is presented to the public in its official documentation ([3]) and some of the most popular learning resources ([7, 28, 42]).

[1]Original author of Agda.

22. Introduction to types

Type theory is a branch of mathematical symbolic logic. It formalizes mathematical concepts through terms, types and a typing relation between them. One could think of types as predicates on terms. We write $T : A$ to say that term T satisfies predicate A, or synonymously, that term T has type A. Later in this chapter we will see that types in *simply typed lambda calculus* provide a basic classification of lambda terms. For instance, a term representing a natural number will have a different type from a lambda term representing a Boolean value. In more expressive type theories which feature dependent types, such as *intuitionistic type theory*, we can express complex mathematical properties such as "$2 * n$ is always even" or that "any finite sequence of numbers can be sorted in lexicographical order".

Type theory has become especially relevant in the following areas.

- **Programming languages and proof assistants**. Simple (non-dependent) types are present in almost every modern programming language. Programming languages use types to classify its objects and functions with the goal of minimizing the amount of errors caused by misusing them. For instance, the term $1 + true$ does not make sense and types are used to rule out the validity of such term.

  Furthermore, the expressiveness of type theories with dependent types make them an adequate basis for modern proof assistants. Due to the constructive nature of the theory the proof assistants can be used as programming languages too. Agda and Coq are examples of that.

- **Foundations of mathematics**. (This paragraph is a paraphrase from [12]) A sufficiently expressive type theory such as Martin-Löf type theory is a formal logical system and philosophical foundation for constructive mathematics. It is a full-scale system which is based on the *propositions-as-types* principle and aims to play a similar role for constructive mathematics as Zermelo-Fraenkel set theory does for classical mathematics.

  (This paragraph is a quote from [35]) *Univalent Foundations of Mathematics* is Vladimir Voevodsky's new program for a comprehensive, computational foundation for mathematics based on the homotopical interpretation of type theory. The type theoretic univalence axiom relates propositional equality on the universe with homotopy equivalence of small types. The program is currently being implemented with the help of the automated proof assistant Coq. The Univalent Foundations program is closely tied to homotopy type theory.

22.1. The origins of types

([9]) Types were first introduced by Russel in 1903 in "Apendix B: The Doctrine of Types, from Principia Mathematica" while trying to avoid a contradiction in set theory, namely Russel's paradox. In Principia Mathematica types are defined as follows.

1. i is the type of individuals (elements of some fixed domain);
2. if $A_1, ..., A_n$ (for $n \geq 0$) are types then $(A_1, ..., A_n)$ is the type of n-ary relations over objects of respective types $A_1, ..., A_n$. Note that for $n = 0$ we have that $()$ is the type of propositions.

For instance, the type of binary relations over individuals is (i, i), the type of binary propositional connectives is $((), ())$. Observe that this formulation prevents a proposition of the form $R(R)$. Assume for a contradiction that $R(R)$ is a proposition, then we have that (by looking at the outer occurrence) R has type (A) for some type A and thus (by looking at the inner occurrence) R has type A but $A \neq (A)$. This observation is the key for avoiding Russel's paradox using types.

The more habitual definition of types is the one that stems from Church's formalization of lambda calculus which includes functions as primitive objects.

1. i is the type of individuals;
2. o is the type of propositions;
3. if A and B are types then $A \to B$ is the type of functions from A to B.

We may observe that $i \to o$ is the type of predicates on individuals, $i \to i$ is the type of functions on individuals and $\underbrace{i \to (i \to \dots \to (i}_{n} \to o))$ is an n-ary relation. Although this definition of types is relevant for historical reasons, it has become obsolete and we proceed by giving a short introduction to three (out of many) versions of lambda calculus available today.

22.2. Untyped lambda calculus

For the language of terms we present a refinement of Church's version due to Curry:

1. *variable*: every variable is a term;
2. *function application*: If A and B are terms then $A\ B$ is a term. Note that application associates to the left, thus $A\ B\ C = (A\ B)\ C$.
3. *lambda abstraction*: If x is a variable and A is a term then $\lambda x.A$ is a term. The body of a lambda abstraction (the expression after the .) extends to the rightmost part. Thus $\lambda x.\lambda y.x\ y = \lambda x.(\lambda y.x\ y)$.

This can be more succinctly expressed in the so-called Backus-Naur form (BNF for short):

$$T \coloneqq x \mid T\ T \mid \lambda x.T$$

In lambda calculus we have the following equation known as β-reduction.

$$(\lambda x.T)\ A = T[x \mapsto A]$$

This equation is often given as a reduction rule from left to right, giving computational value to lambda terms. In other words, β-reduction gives an algorithm based on a rewrite rule that *reduces*, *evaluates* or *computes* a lambda term until it can no longer be reduced. When a term cannot be reduced we say that it is in *normal form.* Note that not every term can be reduced to a normal form term as showcased by the following term, which reduces to itself:

$$(\lambda x.x\ x)\ (\lambda x.x\ x)$$

Notice how this term is of the form $R(R)$ (or $R\ R$, in the new syntax), which we were able to rule out before by using types. In fact, in the next section we will see how this term cannot be assigned a type.

Turing showed that untyped lambda calculus is equivalent in terms of computability to Turing machines ([34]), therefore any computable function has a lambda term that computes it.

It might be difficult to imagine how we could express every computable function in a lambda term. We believe that showing some practical examples will be enlightening, thus we will briefly introduce the *Church encoding* for Booleans and natural numbers.

- **Booleans**. We define *true* and *false* thus:

$$true := \lambda a.\lambda b.a; \quad false := \lambda a.\lambda b.b$$

As we can see, both *true* and *false* are defined as a function that takes two arguments. The former returns the first argument while the latter returns the second. Thus, this encoding of Booleans conveniently allows us to define an *if then else* expression.

$$ite := \lambda b.\lambda x.\lambda y.b\ x\ y$$

It is immediate to see by means of β-reduction that

$$ite\ b\ x\ y = b\ x\ y$$

hence, we will usually prefer to write $b\ x\ y$ instead of $ite\ b\ x\ y$. We can use the *if then else* concept to encode the *and* and *or* operators. It may help to read the *and* as "if the first argument is true return the second argument else return false". Likewise for the *or*.

$$and := \lambda a.\lambda b.a\ b\ false; \quad or := \lambda a.\lambda b.a\ true\ b$$

- **Natural numbers**. The natural number n is encoded as a lambda term that applies n times some parameter function f.

$$0 := \lambda f.\lambda a.a; \quad 1 := \lambda f.\lambda a.f\ a; \quad 2 := \lambda f.\lambda a.f\ (f\ a); \quad ...$$

Then we can define the successor function:

$$suc := \lambda n.\lambda f.\lambda a.f\ (n\ f\ a)$$

Let us show that the successor of 1 is indeed 2.

$$\begin{aligned} suc\ 1 &= (\lambda n.\lambda f.\lambda a.f\ (n\ f\ a))\ (\lambda f.\lambda a.f\ a) && \text{Def} \\ &= \lambda f.\lambda a.f\ ((\lambda f.\lambda a.f\ a)\ f\ a) && \beta\text{-reduction for } n \\ &= \lambda f.\lambda a.f\ (\lambda a.f\ a)\ a && \beta\text{-reduction for } f \\ &= \lambda f.\lambda a.f\ (f\ a) && \beta\text{-reduction for } a \\ &= 2 && \text{Def} \end{aligned}$$

It is also easy to define addition and multiplication. It may help to read *add n m* as "apply m times f, then apply n times f" and *mul n m* as "apply n times (apply m times)".

$$add := \lambda n.\lambda m.\lambda f.\lambda a.n\ f\ (m\ f\ a); \quad mul := \lambda n.\lambda m.\lambda f.\lambda a.n\ (m\ f)\ a$$

22.3. Simply typed lambda calculus

Let us introduce the idea of types in lambda calculus due to Curry. We view types as predicates on lambda terms. We write $T : A$ to say that the term T has type A.

We fix a set of base types $\mathbf{B}$ and a set of term constants

$$\Gamma = \{\langle c_0^0 : B_0\rangle, \langle c_0^1 : B_0\rangle, ..., \langle c_1^0 : B_1\rangle, \langle c_1^1 : B_1\rangle, ...\},$$

where each $B_i \in \mathbf{B}$.

The syntax of terms is defined thus.

$$T := x \mid T\ T \mid \lambda(x : S).T \mid c$$

Where S is a type, c a term constant and x a variable.

The syntax of types is defined thus.

$$S := B \mid S \to S$$

With $B \in \mathbf{B}$. The $\to$ symbol has right associativity, so $A \to B \to C = A \to (B \to C)$.

Then we define typing rules to assign a type to suitable terms. We define a context to be a set of tuples $\langle x : A \rangle$ where x is either a variable or a constant and A is a type. If Γ is a context write $\Gamma \vdash t : A$ to mean that y has type A in context Γ. When a term can be assigned a type in a context we say that it is well-typed in that context. Only well-typed terms are considered valid in simply typed lambda calculus.

$$\dfrac{c : A \in \Gamma}{\Gamma \vdash c : A}\,\textsc{Id} \qquad \dfrac{\Gamma \vdash f : A \to B \qquad \Gamma \vdash t : A}{\Gamma \vdash f\ t : B}\,\textsc{App} \qquad \dfrac{\Gamma \cup \{\langle x : A \rangle\} \vdash t : B}{\Gamma \vdash \lambda(x : S).t : A \to B}\,\textsc{Abstraction}$$

Figure 22.1.: Typing rules for simply typed lambda calculus.

Let us now see how we can give types to Booleans and natural numbers following the encoding given in the previous section. For that, we consider a singleton set of base types $B := \{\alpha\}$.

- **Booleans**. We define the type of Booleans thus:

  $$\mathbb{B} := \alpha \to \alpha \to \alpha$$

  $$\mathit{true} := \lambda(a : \alpha).\lambda(b : \alpha).a; \quad \mathit{false} := \lambda(a : \alpha).\lambda(b : \alpha).b$$

  We proceed by showing that $\emptyset \vdash \mathit{true} : \mathbb{B}$.

  $$\dfrac{\dfrac{\dfrac{\dfrac{}{\{\langle a : \alpha \rangle, \langle b : \alpha \rangle\} \vdash a : \alpha}\,\textsc{Id}}{\{\langle a : \alpha \rangle\} \vdash \lambda(b : \alpha).a : \alpha \to \alpha}\,\textsc{Abs}}{\emptyset \vdash \lambda(a : \alpha).\lambda(b : \alpha).a : \alpha \to \alpha \to \alpha}\,\textsc{Abs}}{\emptyset \vdash \mathit{true} : \mathbb{B}}\,\textsc{Def}$$

  Likewise we can show that $\emptyset \vdash \mathit{false} : \mathbb{B}$. It is routine to check that $\emptyset \vdash \mathit{and} : \mathbb{B} \to \mathbb{B} \to \mathbb{B}$ and that $\emptyset \vdash \mathit{or} : \mathbb{B} \to \mathbb{B} \to \mathbb{B}$. As we can see, types give us information about the nature of the term. For instance, $\mathit{and} : \mathbb{B} \to \mathbb{B} \to \mathbb{B}$ tells us that *and* is a lambda term that expects two Booleans as arguments and returns a Boolean.

- **Natural numbers**. We define the type of natural numbers thus:

  $$\mathbb{N} := (\alpha \to \alpha) \to \alpha \to \alpha$$

  $$0 := \lambda(f : \alpha \to \alpha).\lambda(a : \alpha).a; \quad 1 := \lambda(f : \alpha \to \alpha).\lambda(a : \alpha).f\ a; \quad ...$$

  We can routinely check that for any natural number n we have $\emptyset \vdash n : \mathbb{N}$ and $\emptyset \vdash \mathit{add} : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$ and $\emptyset \vdash \mathit{mul} : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$.

It is a well known property that simply typed lambda calculus is *strongly normalizing*, which means that every well-typed term can be reduced to a normal form. Thus it must be the case, and it is easy to observe, that the non-normalizing term we presented before cannot be typed for any choice of A.

$$(\lambda(x : A).x\ x)\ (\lambda(x : A).x\ x)$$

Strong normalization is a desirable property, but it comes at the price of losing equivalence to Turing machines as there are many computable functions that cannot be expressed in simply typed lambda calculus. To circumvent this, some extensions of simply typed lambda calculus are extended with Curry's Y combinator defined below. The Y combinator, also known as the fixed-point combinator, is a primitive lambda term that can be added to the language and be assigned the type $(A \to A) \to A$ for any type A.

$$Y \coloneqq \lambda g.(\lambda x.g\ (x\ x))\ (\lambda x.g\ (x\ x))$$

The Y combinator gives general recursion and thus the strong normalization property no longer holds.

22.4. Dependently typed lambda calculus

There is no way to briefly present a description of a type system with dependent types in an analogous form to untyped and simply typed lambda calculus. For this reason, we prefer to dedicate a whole chapter to it. In the upcoming Chapter 23 we present a full description of a theory which is based on dependent types and is the logical basis of Agda.

23. Martin Löf's logical framework

In this chapter we will present the intuitionistic type theory presented in [27]. We will refer to this system as *Martin Löf's logical framework* or LF for short. Even though Agda lacks a well defined specification, LF is considered the basis of Agda's type system ([12]). Furthermore, Peter Dybjer, who is a professor at the Chalmers University of Technology[1], suggested[2] the specific version of LF that we will present in this chapter. However, there are important differences between LF and Agda's theory that we will comment in Section 23.5.

System LF is complex and has a lot of cyclic dependencies in the definitions. As such, it is impossible to give a linear presentation. For that, we ask the reader to have some patience when following this chapter. It may be a good idea to skim through the chapter and then read it more thoroughly for a second time. Additionally, it is likely, specially for someone new to type theory, that everything will remain very abstract until we start introducing sets in Section 23.4.

23.1. Basic definitions

The system LF has four kinds of judgment. Each kind of judgment is bound to a *context*. We postpone momentarily the definition of context.

1. "a has type A in context Γ". We write $\Gamma \vdash a : A$. Crucially, in a type theory that follows the paradigm of *propositions as types* we may interpret the statement $a : A$ in several ways, all of them equivalent in such a paradigm:
   - The term a has type A;
   - the term a satisfies the proposition A;
   - the term a is a proof of the proposition A;
   - the term a is a program that satisfies the specification A.
2. "A is a type in context Γ". We write $\Gamma \vdash A : \mathsf{Type}$.
3. "A_1 and A_2 are equal types in context Γ". We write $\Gamma \vdash A_1 = A_2 : \mathsf{Type}$. It is important to remark that the $=$ sign is part of the judgment and is not part of the syntax of types and terms.
4. "a_1 and a_2 are equal elements of the type A in context Γ". We write $\Gamma \vdash a_1 = a_2 : A$. As before, the $=$ sign is part of the judgment.

Notice that we have not yet defined what is the syntax of terms nor types. We need to postpone the definition because in order to describe how we can build types and terms we first need to define several basic concepts. We will define the syntax of types and terms in Section 23.2.

The following definition characterizes what we consider propositions in the LF system.

Definition 23.1. Proposition. We say that A is a proposition (or a type) when we have:

$$\emptyset \vdash A : \mathsf{Type}$$

[1]Where most of the development of Agda takes place.
[2]via email correspondence.

The above is the formal definition of a type within the system. However, it might be helpful for the reader to consider a meta definition of what we understand as type in the system. For that, we quote the description given in [27]:

> What does it mean that something is a type? To know that A is a type is to know what it means to be an object of the type, as well as what it means for two objects to be the same. The identity between objects must be an equivalence relation and it must be decidable.

Since types are proposition in LF, we need a way to discern which propositions hold the status of theorem. Such a notion is given by the following definition.

Definition 23.2. Theorem. We say that some proposition (or type) A is true in LF if there exists some t such that:

$$\emptyset \vdash t : A$$

The following definition gives a precise statement on what it means for two terms to be indiscernible in LF.

Definition 23.3. Equality of terms. We say that two terms t and t' of type A are equal in LF if we have:

$$\emptyset \vdash t = t' : A$$

Similarly we define when two types are considered to be the same in LF.

Definition 23.4. Equality of types. We say that two types T and T' are equal in LF if we have:

$$\emptyset \vdash T = T' : \mathsf{Type}$$

We proceed by defining the concept of *context*.

Definition 23.5. Context. Intuitively, a context, is a sequence of typed variables, where each type may depend on the preceding variables in the sequence.

More precisely, a context is a finite sequence of the form

$$x_1 : S_1, ..., x_{\mathrm{n}} : S_{\mathrm{n}}$$

Where each x_{i} denotes a variable[3]. Furthermore, given arbitrary terms $a_1, ..., a_n$ it must hold that:

$$\begin{gathered}
\emptyset \vdash S_1 : \mathsf{Type} \text{ and } \emptyset \vdash a_1 : S_1; \\
\emptyset \vdash S_2[x_1 \mapsto a_1] : \mathsf{Type} \text{ and } \emptyset \vdash a_2 : S_2[x_1 \mapsto a_1]; \\
\vdots \\
\emptyset \vdash S_n[x_1 \mapsto a_1, ..., x_{n-1} \mapsto a_{n-1}] : \mathsf{Type} \text{ and } \emptyset \vdash a_n : S_n[x_1 \mapsto a_1, ..., x_{n-1} \to a_{n-1}].
\end{gathered}$$

We will now introduce a type for the first time. We introduce the type Set[4].

$$\frac{\textsc{Set}}{\Gamma \vdash \mathsf{Set} : \mathsf{Type}}$$

As we see the previous rule has no assumptions, hence we have that $\emptyset \vdash \mathsf{Set} : \mathsf{Type}$ is valid in LF.

[3]Every variable must be different than the rest, that is, for every i, j such that $i \neq j$ we have $x_{\mathrm{i}} \neq x_{\mathrm{j}}$.

[4]The concept of *set* used here differs from the one in set theory.

The nature of the type Set is given by the following meta definition.

Definition 23.6. Set. A *set* is an inductive description of how its *canonical elements* are built, plus a decidable equality relation between them. Two sets are equal if any canonical element of one set is a canonical element of the other set and moreover, if any two equal canonical elements in one set also are equal in the other set.

For instance, if we want to define the set of natural numbers, we have two canonical elements: *zero* and the *successor* of a natural number. The equality relation can be defined as expected. In Section 23.4 we will see how we can define the natural numbers, among other sets, in more detail.

If A is a set, then the elements of A, denoted with $\mathsf{El}(A)$, in conjunction with their equivalence relation form a type. As such, we add the following rules:

$$\textsc{El}\ \dfrac{\Gamma \vdash A : \mathsf{Set}}{\Gamma \vdash \mathsf{El}(A) : \mathsf{Type}} \qquad \textsc{El=}\ \dfrac{\Gamma \vdash A = B : \mathsf{Set}}{\Gamma \vdash \mathsf{El}(A) = \mathsf{El}(B) : \mathsf{Type}}$$

Note that $\mathsf{El}(.)$ is a primitive which takes a Set as an argument and returns a Type, which represents the elements of the set.

Later in the chapter (Section 23.3) we will define the concept of *family of types* and we will see (in Example 23.13) that $\mathsf{El}(.)$ is family of types over Set.

We are now ready to introduce the first structural rule. The following rule allows us to use assumptions in the context:

$$\textsc{Assum}\ \dfrac{\Gamma \vdash A : \mathsf{Type}}{\Gamma, x : A, \Gamma' \vdash x : A}$$

Finally we can build our first derivation:

Example 23.7. The derivation below shows that we can fully formalize in the system that we have an arbitrary set P as an assumption and show that $\mathsf{El}(P)$ is a type.

$$\textsc{El}\ \dfrac{\textsc{Assum}\ \dfrac{\textsc{Set}\ \dfrac{}{\Gamma \vdash \mathsf{Set} : \mathsf{Type}}}{\Gamma, P : \mathsf{Set}, \Gamma' \vdash P : \mathsf{Set}}}{\Gamma, P : \mathsf{Set}, \Gamma' \vdash \mathsf{El}(P) : \mathsf{Type}}$$

It turns out that this is a common pattern that we will use in the examples later in the chapter. Thus, it is convenient to define a shortcut rule:

$$\textsc{VarType}\ \dfrac{}{\Gamma, P : \mathsf{Set}, \Gamma' \vdash \mathsf{El}(P) : \mathsf{Type}}$$

We proceed by giving some rules that express that equality, both for types and terms, is indeed an equivalence relation.

For equality on terms we have:

$$\textsc{Refl}\ \dfrac{\Gamma \vdash a : A}{\Gamma \vdash a = a : A} \qquad \textsc{Sym}\ \dfrac{\Gamma \vdash a = b : A}{\Gamma \vdash b = a : A} \qquad \textsc{Trans}\ \dfrac{\Gamma \vdash a = b : A \qquad \Gamma \vdash b = c : A}{\Gamma \vdash a = c : A}$$

Analogously, for equality on types we have:

$$\textsc{ReflTy}\ \frac{\Gamma \vdash A : \mathsf{Type}}{\Gamma \vdash A = A : \mathsf{Type}} \qquad \textsc{SymTy}\ \frac{\Gamma \vdash A = B : \mathsf{Type}}{\Gamma \vdash B = A : \mathsf{Type}} \qquad \textsc{TransTy}\ \frac{\Gamma \vdash A = B : \mathsf{Type} \qquad \Gamma \vdash B = C : \mathsf{Type}}{\Gamma \vdash A = C : \mathsf{Type}}$$

Furthermore we can substitute equal types.

$$\textsc{SubsTy1}\ \frac{\Gamma \vdash A = B : \mathsf{Type} \qquad \Gamma \vdash a : A}{\Gamma \vdash a : B} \qquad \textsc{SubsTy2}\ \frac{\Gamma \vdash A = B : \mathsf{Type} \qquad \Gamma \vdash a = b : A}{\Gamma \vdash a = b : B}$$

Example 23.8. Let us show a trivial example. By using the shortcut rule VarType that we defined before we can show that type of the elements of an arbitrary set S is indeed equal to itself:

$$\textsc{ReflTy}\ \frac{\textsc{VarType}\ \frac{}{P : \mathsf{Set} \vdash \mathsf{El}(P) : \mathsf{Type}}}{P : \mathsf{Set} \vdash \mathsf{El}(P) = \mathsf{El}(P) : \mathsf{Type}}$$

With the rules that we have given so far it is not possible to prove a judgment of the kind $\Gamma \vdash a : A$ nor $\Gamma \vdash a = b : A$. We will be able to do so in the next section.

23.2. Rules for types and terms

We are finally ready to introduce the syntax of terms and types. The syntax is introduced by means of typing rules.

We first introduce the syntax of the dependent function type. The dependent function type is the type of functions from a type A to a type B, where B is a type that may depend on an arbitrary term of type A.

$$\textsc{Fun}\ \frac{\Gamma \vdash A : \mathsf{Type} \qquad \Gamma, x : A \vdash B : \mathsf{Type}}{\Gamma \vdash (x : A) \twoheadrightarrow B : \mathsf{Type}}$$

The subtle generalization to allow B to depend on a term of type A has incredible consequences for the system. So much so that it gives name the paradigm of *dependent types*.

When we have $(x : A) \twoheadrightarrow B$ we say that A is the type of the argument and that B is the *return type*. Also, we set $\twoheadrightarrow$ to have right associativity.

When B does not depend on a term of type A, or in other words, when x does not appear free in B, we will simply write $A \twoheadrightarrow B$ instead of $(x : A) \twoheadrightarrow B$. This notation convention motivates the following simplified rule which corresponds to the function type in simply typed lambda calculus:

$$\textsc{Fun'}\ \frac{\Gamma \vdash A : \mathsf{Type} \qquad \Gamma \vdash B : \mathsf{Type}}{\Gamma \vdash A \twoheadrightarrow B : \mathsf{Type}}$$

Example 23.9. As an easy example, we show how for arbitrary set P we can build the function type from elements of P to elements of P.

$$\textsc{Fun}\ \frac{\textsc{VarType}\ \frac{}{P : \mathsf{Set} \vdash \mathsf{El}(P) : \mathsf{Type}} \qquad \textsc{VarType}\ \frac{}{P : \mathsf{Set}, x : \mathsf{El}(P) \vdash \mathsf{El}(P) : \mathsf{Type}}}{P : \mathsf{Set} \vdash (x : \mathsf{El}(P)) \twoheadrightarrow \mathsf{El}(P) : \mathsf{Type}}$$

We have seen how we can build the dependent function type so the natural question is to ask how we can build terms of that type. The answer lies in the simple construction of a lambda abstraction, which is very reminiscent of the lambda abstraction of simply typed lambda calculus.

$$\textsc{Abs}\quad \frac{\Gamma, x : A \vdash b : B}{\Gamma \vdash \lambda x \rightarrow b : (x : A) \twoheadrightarrow B}$$

As a notational convention we have that expression on the right of the $\rightarrow$ extends to the rightmost part, without escaping parenthesis.

Example 23.10. To follow up on Example 23.9 we show how we can build a term that has the type $\mathsf{El}(P) \twoheadrightarrow \mathsf{El}(P)$ presented in that example. Unsirprisingly, the term that we are after is the lambda term $\lambda(x : \mathsf{El}(P)) \rightarrow x$. For the experienced reader it should be obvious that term represents the identity function. For the novel reader we will see it in Example 23.11. We will often refer to terms of a type of the form $(x : A) \twoheadrightarrow B$ as functions. Without further ado we show that $P : \mathsf{Set} \vdash \lambda x \rightarrow x : (x : \mathsf{El}(P)) \twoheadrightarrow \mathsf{El}(P)$.

$$\textsc{Abs}\ \dfrac{\textsc{Assum}\ \dfrac{\textsc{VarType}\ \dfrac{}{P : \mathsf{Set} \vdash \mathsf{El}(P) : \mathsf{Type}}}{P : \mathsf{Set}, x : \mathsf{El}(P) \vdash x : \mathsf{El}(P)}}{P : \mathsf{Set} \vdash \lambda x \rightarrow x : (x : \mathsf{El}(P)) \twoheadrightarrow \mathsf{El}(P)}$$

In order for a function to be useful we need to be able to apply it to an argument. The following rule allows us to do precisely that. More precisely, if we have a term f of type $(x : A) \twoheadrightarrow B$ and a term a of type A, then we can build a term $f\ a$ of type $B[x \mapsto a]$.

$$\textsc{App}\quad \frac{\Gamma \vdash f : (x : A) \twoheadrightarrow B \qquad \Gamma \vdash a : A}{\Gamma \vdash f\ a : B[x \mapsto a]}$$

The most important detail to notice is that in the consequence we have $f\ a : B[x \mapsto a]$. Thus, exhibiting the fact that indeed the type of an application depends not only on the type of the function but also on the argument itself.

Application has left associativity. Thus, $f\ a\ b = (f\ a)\ b$. Also, application binds stronger that abstraction, thus $\lambda x \rightarrow f\ b\ x$ is equivalent to $\lambda x \rightarrow (f\ b\ x)$.

In mathematics, when we have $f(x) := 2 + x$ we expect that $f(3) = 2 + 3$. The rule of β-reduction or simply β-$=$ tells us exactly that. More precisely, the β-$=$ rule tells us that if we have $(\lambda x \rightarrow b)\ a$, then we can substitute the variable x in the body b by the term a. It is important to notice that the term a is also substituted in the type B:

$$\beta\text{-}=\quad \frac{\Gamma \vdash a : A \qquad \Gamma, x : A \vdash b : B}{\Gamma \vdash (\lambda x \rightarrow b)\ a = b[x \mapsto a] : B[x \mapsto a]}$$

Example 23.11. Let us now argue why we identify the term $\lambda x \rightarrow x$ with the identity function. For that, we will show that $P : \mathsf{Set}, a : \mathsf{El}(P) \vdash (\lambda x \rightarrow x)\ a = a : \mathsf{El}(P)$.

$$\beta\text{-}=\ \dfrac{\textsc{Assum}\ \dfrac{\dots}{P : \mathsf{Set}, a : \mathsf{El}(P) \vdash a : \mathsf{El}(P)} \qquad \textsc{Assum}\ \dfrac{\dots}{P : \mathsf{Set}, a : \mathsf{El}(P), x : \mathsf{El}(P) \vdash x : \mathsf{El}(P)}}{P : \mathsf{Set}, a : \mathsf{El}(P) \vdash (\lambda x \rightarrow x)\ a = a : \mathsf{El}(P)}$$

At this point the reader should be able to fill in the ... gaps.

In mathematics, we are used to renaming variables without affecting, if we are careful, the meaning of the expression that we are operating on. For instance the definitions $f(x) \coloneqq 2 + x$ and $f(y) \coloneqq 2 + y$ are essentially the same in any conceivable practical system. This equivalence is usually known as α-equivalence. In the LF system, α-equivalence given by the α-= rule which we define below. Such rule tells us that we can rename the abstracted variable of lambda abstraction terms.

$$\alpha\text{-}{=}\quad \frac{\Gamma, x : A \vdash b : B}{\Gamma \vdash \lambda x \to b = \lambda y \to (b[x \mapsto y]) : (z : A) \twoheadrightarrow B}$$

It is important to remark that y must not occur free in b.

In mathematics, if we have $f(x) \coloneqq g(x)$ we expect that $f = g$. The η-= rule expresses such equivalence. In other words, the rule tells us that abstraction and application cancel each other out.

$$\eta\text{-}{=}\quad \frac{\Gamma \vdash g : (x : A) \twoheadrightarrow B}{\Gamma \vdash \lambda x \to g\ x = g : (x : A) \twoheadrightarrow B}$$

The following allows us to replace equal terms and types inside other terms and types.

The ξ-= rule tells us that we can replace equal bodies of lambda terms.

$$\xi\text{-}{=}\quad \frac{\Gamma, x : A \vdash b = b' : B}{\Gamma \vdash \lambda x \to b = \lambda x \to b' : (x : A) \twoheadrightarrow B}$$

We can perform substitution in a function application.

$$\textsc{SubsApp}\quad \frac{\Gamma \vdash f = f' : (x : A) \twoheadrightarrow B \qquad \Gamma \vdash a = a' : A}{\Gamma \vdash f\ a = f'\ a' : B[x \mapsto a]}$$

Likewise, we can perform substitution in function types:

$$\textsc{SubsFun}\quad \frac{\Gamma \vdash A = A' : \mathsf{Type} \qquad \Gamma, x : A \vdash B = B' : \mathsf{Type}}{\Gamma \vdash (x : A) \twoheadrightarrow B = (x : A') \twoheadrightarrow B'}$$

At this point, we have introduced most of the system but still we cannot show any judgment of the form $\emptyset \vdash a : A$. For that, we will need to wait until Section 23.4, where we introduce concrete sets to the language.

23.3. Families of types

In this section we present the concept of *family of types*. This is a technical concept that does not add much to the intuition of the system. As such, we suggest the reader to skip it on a first read and go back to it later.

Definition 23.12. Family of types. We say that A is a family of types in the context Γ iff $\Gamma \vdash A : \mathsf{Type}$ and is extensional with respect to the context. To be extensional with respect to the context means the following. If $x_1 : S_1, ..., x_\mathrm{n} : S_\mathrm{n} \vdash A$ and the following holds:

$$\begin{aligned}
&\emptyset \vdash a_1 = b_1 : S_1;\\
&\emptyset \vdash a_2 = b_2 : S_1[x_1 \mapsto a_1];\\
&\qquad\vdots\\
&\emptyset \vdash a_\mathrm{n} = b_\mathrm{n} : S_\mathrm{n}[x_1 \mapsto a_1, ..., x_{n-1} \mapsto a_{n-1}].
\end{aligned}$$

Then it must be that:

$$\emptyset \vdash A[x_1 \mapsto a_1, ..., x_{n-1} \mapsto a_{n-1}] = A[x_1 \mapsto b_1, ..., x_{n-1} \mapsto b_{n-1}].$$

Note that in the case where $\Gamma = \emptyset$, while A is still a family of types by definition, we will usually simply say that A is a type.

If we have that A is a family of types in the context $x : B$ we will say that A is a family of types over B. Likewise, if A is a family of types in the context $x : B, y : C$ we will say that is a family of types over B and C. And so on.

Example 23.13. As we have previously commented, $\mathsf{El}(P)$ is a family of types over Set. To check that, we observe that $P : \mathsf{Set} \vdash \mathsf{El}(P)$ follows from rule VarType. Moreover, extensionality is given by the El-= rule.

We proceed by giving some rules for type families. Note that in the upcoming rules we have two assumptions, one on top the other. This is due to spacing and readability reasons. The intended meaning is the same as if the two hypotheses were side by side.

Instantiation of a type family:

TF1
$$\frac{\begin{array}{c} x_1 : A_1, ..., x_\mathrm{n} : A_\mathrm{n} \vdash C : \mathsf{Type} \\ \emptyset \vdash a_1 : A_1 \qquad ... \qquad \emptyset \vdash a_\mathrm{n} : A_\mathrm{n}[x_1 \mapsto a_1, ..., x_{n-1} \mapsto a_{n-1}] \end{array}}{\emptyset \vdash C[x_1 \mapsto a_1, ..., x_\mathrm{n} \mapsto a_\mathrm{n}] : \mathsf{Type}}$$

Substitution in a type family:

TF2
$$\frac{\begin{array}{c} x_1 : A_1, ..., x_\mathrm{n} : A_\mathrm{n} \vdash C : \mathsf{Type} \\ \emptyset \vdash a_1 = b_1 : A_1 \qquad ... \qquad \emptyset \vdash a_\mathrm{n} = b_\mathrm{n} : A_\mathrm{n}[x_1 \mapsto a_1, ..., x_{n-1} \mapsto a_{n-1}] \end{array}}{\emptyset \vdash C[x_1 \mapsto a_1, ..., x_\mathrm{n} \mapsto a_\mathrm{n}] = C[x_1 \mapsto b_1, ..., x_\mathrm{n} \mapsto b_\mathrm{n}]}$$

Instantiation of a term of a type family:

TF3
$$\frac{\begin{array}{c} x_1 : A_1, ..., x_\mathrm{n} : A_\mathrm{n} \vdash c : C \\ \emptyset \vdash a_1 : A_1 \qquad ... \qquad \emptyset \vdash a_\mathrm{n} : A_\mathrm{n}[x_1 \mapsto a_1, ..., x_{n-1} \mapsto a_{n-1}] \end{array}}{\emptyset \vdash c[x_1 \mapsto a_1, ..., x_\mathrm{n} \mapsto a_\mathrm{n}] : C[x_1 \mapsto a_1, ..., x_\mathrm{n} \mapsto a_\mathrm{n}]}$$

Substitution in term of a type family:

TF4
$$\frac{\begin{array}{c} x_1 : A_1, ..., x_\mathrm{n} : A_\mathrm{n} \vdash c : C \\ \emptyset \vdash a_1 = b_1 : A_1 \qquad ... \qquad \emptyset \vdash a_\mathrm{n} = b_\mathrm{n} : A_\mathrm{n}[x_1 \mapsto a_1, ..., x_{n-1} \mapsto a_{n-1}] \end{array}}{\emptyset \vdash c[x_1 \mapsto a_1, ..., x_\mathrm{n} \mapsto a_\mathrm{n}] = c[x_1 \mapsto b_1, ..., x_\mathrm{n} \mapsto b_\mathrm{n}] : C[x_1 \mapsto a_1, ..., x_\mathrm{n} \mapsto a_\mathrm{n}]}$$

Substitution of a type family.

TF5
$$\frac{\begin{array}{c} x_1 : A_1, ..., x_\mathrm{n} : A_\mathrm{n} \vdash B = C : \mathsf{Type} \\ \emptyset \vdash a_1 : A_1 \qquad ... \qquad \emptyset \vdash a_\mathrm{n} : A_\mathrm{n}[x_1 \mapsto a_1, ..., x_{n-1} \mapsto a_{n-1}] \end{array}}{\emptyset \vdash B[x_1 \mapsto a_1, ..., x_\mathrm{n} \mapsto a_\mathrm{n}] = C[x_1 \mapsto a_1, ..., x_\mathrm{n} \mapsto a_\mathrm{n}] : B[x_1 \mapsto a_1, ..., x_\mathrm{n} \mapsto a_\mathrm{n}]}$$

Substitution of a type family term.

TF6
$$\frac{\begin{array}{c} x_1 : A_1, ..., x_\mathrm{n} : A_\mathrm{n} \vdash b = c : C \\ \emptyset \vdash a_1 : A_1 \qquad ... \qquad \emptyset \vdash a_\mathrm{n} : A_\mathrm{n}[x_1 \mapsto a_1, ..., x_{n-1} \mapsto a_{n-1}] \end{array}}{\emptyset \vdash b[x_1 \mapsto a_1, ..., x_\mathrm{n} \mapsto a_\mathrm{n}] = c[x_1 \mapsto a_1, ..., x_\mathrm{n} \mapsto a_\mathrm{n}] : C[x_1 \mapsto a_1, ..., x_\mathrm{n} \mapsto a_\mathrm{n}]}$$

23.4. Introducing sets

In this section we will define some sets that will allow us to represent different mathematical objects, such as pairs or natural numbers, in the system LF.

For each set we will introduce the following (we will be purposely vague in this summary. A more precise list is given a few lines below).

1. A constant that denotes the set.
2. Some (zero or more) constants to build elements in the set.
3. A way to interact with the elements of the set.

The subsequent sections are going to be structured as follows: We define a new set, including all the parts mentioned before. then we give some examples. During the introduction of the first few sets we show how we can embed propositional intuitionistic logic into LF. After defining the natural numbers and the identity set we show how we can prove a property on addition by (formalized) induction.

We believe that the summary above is enough to continue to Section 23.4.1. In fact, we recommend the reader to continue to Section 23.4.1 and come back to read the remainder of this section once they have seen some examples of sets.

We proceed by repeating the summary above with more precision. We want to stress that the following details do not add much to the intuition of the system and we add them just for reference.

For each set we will introduce the following:

1. A typed constant $\mathsf{T} : S$ that represents the set. By that we mean that we add a rule of the form $\emptyset \vdash \mathsf{T} : S$ to the system. Moreover the following must hold:
   - T is a fresh constant;
   - S is of the form $(x_1 : A_1) \twoheadrightarrow \ldots \twoheadrightarrow (x_\mathrm{n} : A_\mathrm{n}) \twoheadrightarrow \mathsf{Set}$;
   - we have $\emptyset \vdash S : \mathsf{Type}$.
2. Zero or more[5] typed constants $\mathsf{c}_1 : C_1, \ldots, \mathsf{c}_\mathrm{n} : C_\mathrm{n}$ to introduce elements of the set. We call these constants *constructors* of the set. Furthermore each C_i must be of the form

   $$(y_1 : A_1) \twoheadrightarrow \ldots \twoheadrightarrow (y_\mathrm{n} : A_\mathrm{n}) \twoheadrightarrow (z_1 : B_1) \twoheadrightarrow \ldots \twoheadrightarrow (z_\mathrm{m} : B_\mathrm{m}) \twoheadrightarrow \mathsf{T}\ y_1\ \ldots\ y_\mathrm{n}$$

   The reader should notice the following:
   - The types of the first n arguments of each constructor must coincide with the types of the arguments of S.
   - Furthermore, the return type of the constructor (the type after the rightmost arrow) must be T applied to the first n arguments of the constructor. In this case $\mathsf{T}\ y_1\ \ldots\ y_\mathrm{n}$.
3. A way to interact with the elements of the set. This is done via an induction principle. The induction principle is defined by giving a typed constant to represent it. Then for each of the constructors we add an equality that describes the behavior of the induction principle with that constructor.

   The concept of *principle of induction* that we use here should be taken in the wide sense of *structural induction* since every set is defined in an inductive way. Of course, not every set that we will define has a recursive nature and thus the introduced induction principle

[5]The empty set is the only set which has no constants to introduce new elements.

for those sets is not going to be reminiscent of a principle of induction in the traditional sense.

We use the following notational convention: If X is the constant that denotes the set, then we will define the constant that represents the induction principle as $\mathsf{case}_{\mathsf{X}}$.

The rest of this chapter is going to be structured as follows: We define a new set, including all the parts mentioned before. then we give some examples. During the introduction of the first few sets we show how we can embed propositional intuitionistic logic into LF. After defining the natural numbers and the identity set we show how we can prove a property on addition by (formalized) induction.

23.4.1. Function set

We introduce the typed constant that denotes the set of functions from elements of a set A to elements of a set B:

$$\rightarrowtail : \mathsf{Set} \twoheadrightarrow \mathsf{Set} \twoheadrightarrow \mathsf{Set}$$

When we say that we introduce a typed constant to the language is shorthand for saying that we introduce a rule to assert that the new constant can be proven to have the given type from the empty set. In our case, it corresponds to the following rule:

$$\frac{}{\emptyset \vdash \rightarrowtail : \mathsf{Set} \twoheadrightarrow \mathsf{Set} \twoheadrightarrow \mathsf{Set}}$$

Notation: We will use infix notation and write $A \rightarrowtail B$ instead of $\rightarrowtail A\ B$. Also, $\rightarrowtail$ has right associativity.

We add a constructor to introduce elements in this set:

$$\Lambda : (A : \mathsf{Set}) \twoheadrightarrow (B : \mathsf{Set}) \twoheadrightarrow (\mathsf{El}(A) \twoheadrightarrow \mathsf{El}(B)) \twoheadrightarrow \mathsf{El}(A \rightarrowtail B)$$

As we can see, the return type of Λ is $\mathsf{El}(A \rightarrowtail B)$. As such, if we apply Λ to a set A, a set B and a function from elements of A to elements of B we get an element of the set $A \rightarrowtail B$. In this case we will not introduce more constructors, thus, applying Λ to the corresponding arguments will be the only way to build an element of the function set.

We proceed by adding an induction principle:

$$\mathsf{case}_\Lambda : (A : \mathsf{Set}) \twoheadrightarrow (B : \mathsf{Set}) \twoheadrightarrow (\mathsf{El}(A \rightarrowtail B)) \twoheadrightarrow \mathsf{El}(A) \twoheadrightarrow \mathsf{El}(B)$$

Let us clarify why we use the words *induction principle* here. As already pointed out in the introduction of this section, every set must be defined in an inductive way. Some sets only have non-recursive constructors, like pairs or the function set being defined here. In the case of the function set we only have the Λ constructor. From the type of Λ we see that in order to build a term of type $\mathsf{El}(A \rightarrowtail B)$ we need to provide a term of type $\mathsf{El}(A) \twoheadrightarrow \mathsf{El}(B)$. We see that the inductive principle case_Λ does works in the opposite direction: from a term of type $\mathsf{El}(A \rightarrowtail B)$ returns a term of type $\mathsf{El}(A) \twoheadrightarrow \mathsf{El}(B)$. Because of this, the functionality of the case_Λ constant is often referred to as *inversion*, *deconstruction* or *unwrapping* principle. We believe that in this introduction to sets it helps to use a homogeneous language so we will always use the general term *induction principle*.

And the equality:

$$\textsc{case}\Lambda\quad \frac{f : \mathsf{El}(A) \twoheadrightarrow \mathsf{El}(B)}{\mathsf{case}_\Lambda\ A\ B\ (\Lambda\ A\ B\ f) = f : \mathsf{El}(A) \twoheadrightarrow \mathsf{El}(B)}$$

The following rules are redundant but they help keeping proofs shorter.

$$\textsc{Intro}\Lambda\ \dfrac{f : \mathsf{El}(A) \twoheadrightarrow \mathsf{El}(B)}{\Lambda\ A\ B\ f : \mathsf{El}(A \rightarrowtail B)} \qquad \textsc{Abs}\Lambda\ \dfrac{\Gamma, x : A \vdash b : B}{\Lambda\ A\ B\ (\lambda x \to b) : \mathsf{El}(A \rightarrowtail B)} \qquad \textsc{App}\Lambda\ \dfrac{f : \mathsf{El}(A \rightarrowtail B) \qquad a : \mathsf{El}(A)}{\mathsf{case}_\Lambda\ f\ a : \mathsf{El}(B)}$$

Notation: When we have $f : \mathsf{El}(A \rightarrowtail B)$ and $a : \mathsf{El}(A)$ we will write $f\ a$ instead of $\mathsf{case}_\Lambda\ f\ a$. Note that this notational convention is very convenient and adds no ambiguity since $f\ a$ on its own would never be a valid term.

In each of the sections where we define a set we will draw a horizontal line such as the one below to separate the definition of the set from the corresponding examples.

Consider the fragment of intuitionistic logic with only implication. If A is a propositional formula then A^* is the type that identifies the formula A in LF, where $*$ is a map from propositional formulas to types as defined below.

$$A^* := \mathsf{El}([\![A]\!])$$

$$[\![p]\!] := P \qquad \text{where } P : \mathsf{Set}$$
$$[\![A \to B]\!] := [\![A]\!] \rightarrowtail [\![B]\!]$$

We map lowercase propositional variables to the same uppercase variable of type Set, that is, $[\![p]\!] = P$ with $P : \mathsf{Set}$, $[\![q]\!] = Q$ with $Q : \mathsf{Set}$, and so on.

Let us now show that $(p \to p)^*$ is a theorem in LF. In order to prove that, we need to give a term with type $\mathsf{El}(P \rightarrowtail P)$. This term is in fact the identity function wrapped in the function set, as shown below.

$$\textsc{Abs}\Lambda\ \dfrac{\textsc{Assum}\ \dfrac{\textsc{VarType}\ \dfrac{}{P : \mathsf{Set} \vdash \mathsf{El}(P) : \mathsf{Type}}}{P : \mathsf{Set}, x : \mathsf{El}(P) \vdash x : \mathsf{El}(P)}}{P : \mathsf{Set} \vdash \Lambda\ P\ P\ (\lambda x \to x) : \mathsf{El}(P \rightarrowtail P)}$$

As another example we show that $(p \to (q \to p))^*$ is a theorem in LF:

$$\textsc{Abs}\Lambda\ \dfrac{\textsc{Abs}\Lambda\ \dfrac{\textsc{Assum}\ \dfrac{\textsc{VarType}\ \dfrac{}{P : \mathsf{Set}, Q : \mathsf{Set}, y : \mathsf{EL}(Q) \vdash \mathsf{El}(P) : \mathsf{Type}}}{P : \mathsf{Set}, Q : \mathsf{Set}, x : \mathsf{EL}(P), y : \mathsf{EL}(Q) \vdash x : \mathsf{El}(P)}}{P : \mathsf{Set}, Q : \mathsf{Set}, x : \mathsf{EL}(P) \vdash \Lambda\ Q\ P\ (\lambda y \to x) : Q \rightarrowtail P}}{P : \mathsf{Set}, Q : \mathsf{Set} \vdash \Lambda\ P\ (Q \rightarrowtail P)\ (\lambda x \to \Lambda\ Q\ P\ (\lambda y \to x)) : \mathsf{El}(P \rightarrowtail Q \rightarrowtail P)}$$

Notation. In order to improve readability, from now on, where we would have $\Lambda\ A\ B\ (\lambda x \to b)$ and A and B are clear from the context, we will write $\Lambda x \to b$ instead. For instance, the last step of the previous proof would be written as

$$\textsc{Abs}\Lambda\ \dfrac{\cdots}{P : \mathsf{Set}, Q : \mathsf{Set} \vdash \Lambda x \to \Lambda y \to x : \mathsf{El}(P \rightarrowtail Q \rightarrowtail P)}$$

We now show the last step of the proof for $((p \to (q \to r)) \to ((p \to q) \to (p \to r))^*$. It should not be hard for the reader to see how the proof can be finished.

$$\textsc{Abs}\Lambda\ \dfrac{\cdots}{\begin{array}{c}P : \mathsf{Set}, Q : \mathsf{Set}, R : \mathsf{Set} \vdash \Lambda f \to (\Lambda g \to (\Lambda x \to f\ x\ (g\ x))) : \\ \mathsf{El}((P \rightarrowtail Q \rightarrowtail R) \rightarrowtail (P \rightarrowtail Q) \rightarrowtail P \rightarrowtail R)\end{array}}$$

Observe now that the modus ponens rule is valid, that is, if $(p \to q)^*$ and p^* are theorems in LF, then q^* is also a theorem. From the assumption it follows that there are terms $f : \mathsf{El}(P \rightarrowtail Q)$ and $a : \mathsf{EL}(P)$. Then we have that $\mathsf{case}_\wedge\ f\ a$, which we write as $f\ a$, is the desired term.

$$\textsc{App}\ \frac{\overline{P : \mathsf{Set}, Q : \mathsf{Set} \vdash f : \mathsf{El}(P \rightarrowtail Q)} \qquad \overline{P : \mathsf{Set}, Q : \mathsf{Set} \vdash a : \mathsf{El}(P)}}{P : \mathsf{Set}, Q : \mathsf{Set} \vdash f\ a : \mathsf{El}(Q)}$$

Theorem 23.14.

If A is a theorem in the fragment of propositional intuitionistic logic with only implication, then A^* is a theorem in LF.

Proof. By induction on A and using the previous facts. □

23.4.2. Top set

We now define the singleton set. We name it $\top$. We say that it is a singleton set because it only has one element, namely tt. The $\top$ set represents "true" in this logic.

We add the constant that represents the set:

$$\top : \mathsf{Set}$$

We introduce the constant that is the only element in the set:

$$\mathsf{tt} : \mathsf{El}(\top)$$

An induction principle:

$$\mathsf{case}_\top : (P : \mathsf{El}(\top) \twoheadrightarrow \mathsf{Set}) \twoheadrightarrow \mathsf{El}(P\ \mathsf{tt}) \twoheadrightarrow (a : \mathsf{El}(\top)) \twoheadrightarrow \mathsf{El}(P\ a)$$

The above type should read as follows. Given a predicate P, a proof that P holds for the base case, that is, an element of the set $P\ \mathsf{tt}$, and an arbitrary element a in the set $\top$, we build a term in the set $P\ a$, which is a proof that a satisfies P.

We also add the following equality.

$$\overset{\textsc{case}\top}{\frac{P : \mathsf{El}(\top) \twoheadrightarrow \mathsf{Set} \qquad p : \mathsf{El}(P\ \mathsf{tt})}{\mathsf{case}_\top\ P\ p\ \mathsf{tt} = p\ \mathsf{tt} : \mathsf{El}(P\ \mathsf{tt})}}$$

As expected from a set representing *true*, it is trivial to provide a term of type $\mathsf{El}(\top)$.

$$\overline{\mathsf{tt} : \mathsf{El}(\top)}$$

23.4.3. Bottom set

We define the empty set, $\bot$, as we call it. We say that it is the empty set because there is no term t such that $t : \mathsf{El}(\bot)$. The $\bot$ type represents "false" in this theory.

We introduce the constant that represents the type:

$$\bot : \mathsf{Set}$$

We do not introduce any constructors for this set. We introduce an induction principle:

$$\mathsf{case}_\bot : (P : \mathsf{El}(\bot) \twoheadrightarrow \mathsf{Set}) \twoheadrightarrow (a : \mathsf{El}(\bot)) \twoheadrightarrow \mathsf{El}(P\ a)$$

The justification of this induction principle is that since it is impossible to build a term of type $\bot$ by definition, then from the assumption that we have a term of type $\bot$, follows anything.

We add the following equation to the map $[\![.]\!]$:

$[\![\bot]\!] := \bot$

The $\bot$ on the right of $:=$ is the constant introduced in this section.

Let us now show that the principle $(\bot \to p)^*$ known as *ex falso quodlibet* or *principle of explosion* holds in LF. For that, we provide a term typed $(\bot \to p)^*$.

$$\textsc{Abs}\Lambda\,\dfrac{\textsc{App}\,\dfrac{\textsc{App}\,\dfrac{\dfrac{}{\mathsf{case}_\bot : \ldots}\quad \textsc{Abs}\,\dfrac{\textsc{Assum}\,\dfrac{}{P : \mathsf{Set}, x : \mathsf{El}(\bot) \vdash P : \mathsf{Set}}}{P : \mathsf{Set}(\lambda x \to P) : \mathsf{El}(\bot) \twoheadrightarrow \mathsf{Set}}}{P : \mathsf{Set}, y : \mathsf{El}(\bot) \vdash \mathsf{case}_\bot\ (\lambda x \to P) : \mathsf{El}(\bot) \twoheadrightarrow \mathsf{El}(P)}\quad \textsc{Assum}\,\dfrac{}{y : \mathsf{El}(\bot) \vdash y : \mathsf{El}(\bot)}}{P : \mathsf{Set}, y : \mathsf{El}(\bot) \vdash \mathsf{case}_\bot\ (\lambda x \to P)\ y : \mathsf{El}(P)}}{P : \mathsf{Set} \vdash \Lambda y \to \mathsf{case}_\bot\ (\lambda x \to P)\ y : \mathsf{El}(\bot \rightarrowtail P)}$$

23.4.4. Disjoint unions set

We introduce the constant:

$$\uplus : \mathsf{Set} \twoheadrightarrow \mathsf{Set} \twoheadrightarrow \mathsf{Set}$$

Notation.We will use infix notation for $\uplus$, hence we will write $A\ \uplus\ B$ instead of $\uplus\ A\ B$. The disjoint union set represents two options, hence, when we have a term of type $A\ \uplus\ B$ it means that we either have a term of type A or a term of type B (and we know which of the two we have).

Since we want to represent two options, we introduce two constructors:

$$\begin{aligned}\mathsf{inj}_1 &: (A : \mathsf{Set}) \twoheadrightarrow (B : \mathsf{Set}) \twoheadrightarrow \mathsf{El}(A) \twoheadrightarrow \mathsf{El}(A\ \uplus\ B)\\ \mathsf{inj}_2 &: (A : \mathsf{Set}) \twoheadrightarrow (B : \mathsf{Set}) \twoheadrightarrow \mathsf{El}(B) \twoheadrightarrow \mathsf{El}(A\ \uplus\ B)\end{aligned}$$

We introduce the following induction principle:

$$\begin{aligned}&\mathsf{case}_\uplus : (A : \mathsf{Set}) \twoheadrightarrow (B : \mathsf{Set}) \twoheadrightarrow (P : \mathsf{El}(A\ \uplus\ B) \twoheadrightarrow \mathsf{Set})\\ &\qquad \twoheadrightarrow ((a : \mathsf{El}(A)) \twoheadrightarrow \mathsf{El}(P\ (\mathsf{inj}_1\ A\ B\ a))) && \text{case A}\\ &\qquad \twoheadrightarrow ((b : \mathsf{El}(B)) \twoheadrightarrow \mathsf{El}(P\ (\mathsf{inj}_2\ A\ B\ b))) && \text{case B}\\ &\qquad \twoheadrightarrow (u : \mathsf{El}(A\ \uplus\ B))\\ &\qquad \twoheadrightarrow \mathsf{El}(P\ u)\end{aligned}$$

As we can easily identify from the type of $\mathsf{case}_\uplus$, this induction principle corresponds to a proof by cases. Note that we need to provide a proof for P given an element of A and also a proof of P given an element of B.

We also introduce the rules below. These rules tells us that when we apply the proof by cases to a disjoint set built with the inj_1 constructor we apply the proof which corresponds to the first case, whereas if we apply proof by cases to a disjoint set built with the inj_2 constructor we apply the proof which corresponds to the second case.

$$\textsc{Case}\uplus_1 \quad \frac{\begin{array}{c} P : \mathsf{El}(A \ \uplus \ B) \twoheadrightarrow \mathsf{Set} \\ p_1 : (a : \mathsf{El}(A)) \twoheadrightarrow \mathsf{El}(P\ (\mathsf{inj}_1\ A\ B\ a)) \\ p_2 : (b : \mathsf{El}(B)) \twoheadrightarrow \mathsf{El}(P\ (\mathsf{inj}_2\ A\ B\ b)) \\ a : \mathsf{El}(A) \end{array}}{\mathsf{case}_{\uplus}\ A\ B\ P\ p_1\ p_2\ (\mathsf{inj}_1\ A\ B\ a) = p_1\ a : \mathsf{El}(P\ (\mathsf{inj}_1\ A\ B\ a))}$$

$$\textsc{Case}\uplus_2 \quad \frac{\begin{array}{c} P : \mathsf{El}(A \ \uplus \ B) \twoheadrightarrow \mathsf{Set} \\ p_1 : (a : \mathsf{El}(A)) \twoheadrightarrow \mathsf{El}(P\ (\mathsf{inj}_1\ A\ B\ a)) \\ p_2 : (b : \mathsf{El}(B)) \twoheadrightarrow \mathsf{El}(P\ (\mathsf{inj}_2\ A\ B\ b)) \\ b : \mathsf{El}(B) \end{array}}{\mathsf{case}_{\uplus}\ A\ B\ P\ p_1\ p_2\ (\mathsf{inj}_2\ A\ B\ b) = p_2\ b : \mathsf{El}(P\ (\mathsf{inj}_2\ A\ B\ b))}$$

We extend the map $[\![.]\!]$ we defined with the following equation:

$[\![A \vee B]\!] := [\![A]\!] \uplus [\![B]\!]$

We show that $(p \to (p \vee q))^*$ is a theorem in LF.

$$\frac{\cdots}{P : \mathsf{Set}, Q : \mathsf{Set} \vdash \Lambda x \to \mathsf{inj}_1\ P\ Q\ x : \mathsf{El}(P \rightarrowtail (P \ \uplus \ Q))}$$

The proof of $(q \to (p \vee q))^*$ is analogous.

$$\frac{\cdots}{P : \mathsf{Set}, Q : \mathsf{Set} \vdash \Lambda x \to \mathsf{inj}_2\ P\ Q\ x : \mathsf{El}(Q \rightarrowtail (P \ \uplus \ Q))}$$

Finally we have that $((p \to r) \to ((q \to r) \to ((p \vee q) \to r)))^*$ is a theorem in LF. Finishing the proof is straightforward.

$$\frac{\cdots}{\begin{array}{c} P : \mathsf{Set}, Q : \mathsf{Set}, R : \mathsf{Set} \vdash \Lambda f \to \Lambda g \to \Lambda x \to \mathsf{case}_{\uplus}\ P\ Q\ (\lambda x \to R)\ (\mathsf{case}_{\wedge}\ f)\ (\mathsf{case}_{\wedge}\ g)\ x : \\ \mathsf{El}((P \rightarrowtail R) \rightarrowtail (Q \rightarrowtail R) \rightarrowtail (P \uplus Q) \rightarrowtail R) \end{array}}$$

23.4.5. Pairs set

The following constant which represents the set of pairs:

$$\times : \mathsf{Set} \twoheadrightarrow \mathsf{Set} \twoheadrightarrow \mathsf{Set}$$

Notation. We will use infix notation for $\times$, hence we will write $A \ \times B$ instead of $\times A\ B$.

We introduce one constructor:

$$\mathsf{pair} : (A : \mathsf{Set}) \twoheadrightarrow (B : \mathsf{Set}) \twoheadrightarrow \mathsf{El}(A) \twoheadrightarrow \mathsf{El}(B) \twoheadrightarrow \mathsf{El}(A \ \times \ B)$$

We introduce the following induction principle:

$$\begin{aligned} \mathsf{case}_{\times} : & (A : \mathsf{Set}) \twoheadrightarrow (B : \mathsf{Set}) \twoheadrightarrow (P : \mathsf{El}(A \ \times \ B) \twoheadrightarrow \mathsf{Set}) \\ & \twoheadrightarrow ((a : \mathsf{El}(A)) \twoheadrightarrow (b : \mathsf{El}(B)) \twoheadrightarrow \mathsf{El}(P\ (\mathsf{pair}\ A\ B\ a\ b))) \\ & \twoheadrightarrow (u : \mathsf{El}(A \ \times B)) \\ & \twoheadrightarrow \mathsf{El}(P\ u) \end{aligned}$$

The functionality of this induction principle is to unwrap a term of type $\mathsf{El}(A \ \times \ B)$ to access its components.

We add the following rule, which tells us that the constructor pair and the induction principle $\mathsf{case}_\times$ cancel each other.

$$\textsc{case}\times \quad \dfrac{\begin{array}{c} P : \mathsf{El}(A \ \times \ B) \twoheadrightarrow \mathsf{Set} \\ p : (a' : \mathsf{El}(A)) \twoheadrightarrow (b' : \mathsf{El}(B)) \twoheadrightarrow \mathsf{El}(P \ (\mathsf{pair} \ A \ B \ a' \ b')) \\ a : \mathsf{El}(A) \qquad b : \mathsf{El}(B) \end{array}}{\mathsf{case}_\times \ A \ B \ P \ p \ (\mathsf{pair} \ A \ B \ a \ b) = p \ a \ b : P \ (\mathsf{pair} \ A \ B \ a \ b)}$$

We extend the $[\![.]\!]$ map with the following clause:

$[\![A \wedge B]\!] := [\![A]\!] \times [\![B]\!]$

We show that $(p \wedge q \to p)^*$ is a theorem in LF.

$$\dfrac{\dots}{P : \mathsf{Set}, Q : \mathsf{Set} \vdash \Lambda x \to \mathsf{case}_\times \ P \ Q \ (\lambda v \to P) \ (\lambda y \to \lambda z \to y) \ x : \mathsf{El}((P \times Q) \rightarrowtail P)}$$

Analogously, the proof for $(p \wedge q \to q)^*$.

$$\dfrac{\dots}{P : \mathsf{Set}, Q : \mathsf{Set} \vdash \Lambda x \to \mathsf{case}_\times \ P \ Q \ (\lambda v \to Q) \ (\lambda y \to \lambda z \to z) \ x : \mathsf{El}((P \times Q) \rightarrowtail Q)}$$

Finally we show that $p \to (q \to (p \wedge q))^*$ is a theorem in LF.

$$\dfrac{\dots}{P : \mathsf{Set}, Q : \mathsf{Set} \vdash \Lambda x \to \Lambda y \to \mathsf{pair} \ P \ Q \ x \ y : \mathsf{El}(P \rightarrowtail Q \rightarrowtail (P \times Q))}$$

23.4.6. Dependent functions set

Dependent functions generalize regular functions in the way that the type of the return type depends on the argument value. More precisely, a dependent function is a map from elements of a set A to elements of a set B indexed by elements of A. Thus we introduce the following constant:

$$\rightsquigarrow : (A : \mathsf{Set}) \twoheadrightarrow (B : \mathsf{El}(A) \twoheadrightarrow \mathsf{Set}) \twoheadrightarrow \mathsf{Set}$$

Notation: We will use infix notation and write $A \ \rightsquigarrow \ B$ instead of $\rightsquigarrow \ A \ B$. Also, $\rightsquigarrow$ has right associativity.

We add a constant to introduce elements in this set:

$$\Lambda' : (A : \mathsf{Set}) \twoheadrightarrow (B : \mathsf{El}(A) \twoheadrightarrow \mathsf{Set}) \twoheadrightarrow ((a : \mathsf{El}(A)) \twoheadrightarrow \mathsf{El}(B \ a)) \twoheadrightarrow \mathsf{El}(A \ \rightsquigarrow \ B)$$

We add an induction principle:

$$\mathsf{case}_\Lambda{}' : (A : \mathsf{Set}) \twoheadrightarrow (B : \mathsf{El}(A) \twoheadrightarrow \mathsf{Set}) \twoheadrightarrow (\mathsf{El}(A \rightsquigarrow B)) \twoheadrightarrow (a : \mathsf{El}(A)) \twoheadrightarrow \mathsf{El}(B \ a)$$

And the equality:

$$\textsc{case}\Lambda\text{'} \quad \dfrac{f : \mathsf{El}(A) \twoheadrightarrow \mathsf{El}(B)}{\mathsf{case}_\Lambda{}' \ A \ B \ (\Lambda \ A \ B \ f) = f : (a : \mathsf{El}(A)) \twoheadrightarrow \mathsf{El}(B \ a)}$$

The following rules are redundant but they help keeping proofs shorter.

$$\textsc{Intro}\Lambda\text{'} \ \dfrac{f : (a : \mathsf{El}(A)) \twoheadrightarrow \mathsf{El}(B \ a)}{\Lambda \ A \ B \ f : \mathsf{El}(A \rightsquigarrow B)} \qquad \textsc{Abs}\Lambda\text{'} \ \dfrac{x : A \vdash b : B \ x}{\Lambda \ A \ B \ (\lambda x \to b) : \mathsf{El}(A \rightsquigarrow B)} \qquad \textsc{App}\Lambda\text{'} \ \dfrac{f : \mathsf{El}(A \rightsquigarrow B) \qquad a : \mathsf{El}(A)}{\mathsf{case}_\Lambda \ f \ a : \mathsf{El}(B \ a)}$$

Notation: When we have $f : \mathsf{El}(A \rightsquigarrow B)$ and $a : \mathsf{El}(A)$ we will write $f \ a$ instead of $\mathsf{case}'_\Lambda \ f \ a$.

23.4.7. Σ Pairs set

A Σ *pair* generalizes the concept of a regular pair. In a Σ pair the type of the second component depends on the value of the first component.

We introduce the constant for the set:

$$\Sigma : (A : \mathsf{Set}) \twoheadrightarrow (B : \mathsf{El}(A) \twoheadrightarrow \mathsf{Set}) \twoheadrightarrow \mathsf{Set}$$

We introduce a constant to build dependent pairs:

$$\mathsf{pair}_\Sigma : (A : \mathsf{Set}) \twoheadrightarrow (B : A \twoheadrightarrow \mathsf{Set}) \twoheadrightarrow (a : \mathsf{El}(A)) \twoheadrightarrow (b : \mathsf{El}(B\ a)) \twoheadrightarrow \Sigma\ A\ B$$

The above constructor takes four arguments: The type of the first component, the type of the second component, the term for the first component, and the term for the second component. We add the following induction principle:

$$\begin{aligned}\mathsf{case}_\Sigma :& (A : \mathsf{Set}) \twoheadrightarrow (B : \mathsf{El}(A) \twoheadrightarrow \mathsf{Set}) \twoheadrightarrow (P : \Sigma\ A\ B \twoheadrightarrow \mathsf{Set})\\ &\twoheadrightarrow ((a : \mathsf{El}(A)) \twoheadrightarrow (b : \mathsf{El}(B\ a)) \twoheadrightarrow P\ (\mathsf{pair}_\Sigma\ A\ B\ a\ b))\\ &\twoheadrightarrow (u : \mathsf{El}(\Sigma\ A\ B))\\ &\twoheadrightarrow \mathsf{El}(P\ u)\end{aligned}$$

We also add this equality.

$$\frac{\begin{array}{c}P : \mathsf{El}(\Sigma\ A\ B) \twoheadrightarrow \mathsf{Set}\\ p : (a' : \mathsf{El}(A)) \twoheadrightarrow (b' : \mathsf{El}(B\ a')) \twoheadrightarrow \mathsf{El}(P\ (\mathsf{pair}_\Sigma\ A\ B\ a'\ b'))\\ a : \mathsf{El}(A) \qquad b : \mathsf{El}(B\ a)\end{array}}{\mathsf{case}_\Sigma\ A\ B\ P\ p\ (\mathsf{pair}_\Sigma\ A\ B\ a\ b) = p\ a\ b : P\ (\mathsf{pair}_\Sigma\ A\ B\ a\ b)}$$

23.4.8. Natural numbers set

We represent the set of natural numbers with the constant $\mathbb{N}$:

$$\mathbb{N} : \mathsf{Set}$$

We add two constants to build elements in $\mathbb{N}$.

$$\begin{gathered}\mathbf{0} : \mathbb{N}\\ \mathsf{suc} : \mathbb{N} \twoheadrightarrow \mathbb{N}\end{gathered}$$

We add an induction principle:

$$\begin{aligned}\mathsf{case}_\mathbb{N} :& (P : \mathsf{El}(\mathbb{N}) \twoheadrightarrow \mathsf{Set})\\ &\twoheadrightarrow \mathsf{El}(P\ \mathbf{0})\\ &\twoheadrightarrow ((n : \mathsf{El}(\mathbb{N})) \twoheadrightarrow \mathsf{El}(P\ n) \twoheadrightarrow \mathsf{El}(P\ (\mathsf{suc}\ n)))\\ &\twoheadrightarrow (a : \mathsf{El}(\mathbb{N}))\\ &\twoheadrightarrow \mathsf{El}(P\ a)\end{aligned}$$

We add two equalities for the induction principle:

$$\textsc{case 0}\ \frac{\begin{array}{c}P : \mathsf{El}(\mathbb{N}) \twoheadrightarrow \mathsf{Set} \qquad B : \mathsf{El}(P\ \mathbf{0})\\ R : (n : \mathsf{El}(\mathbb{N})) \twoheadrightarrow \mathsf{El}(P\ n) \twoheadrightarrow \mathsf{El}(P\ (\mathsf{suc}\ n))\end{array}}{\mathsf{case}_\mathbb{N}\ P\ B\ R\ \mathbf{0} = B : \mathsf{El}(P\ \mathbf{0})}$$

$$\textsc{case suc}\ \frac{\begin{array}{c}P : \mathsf{El}(\mathbb{N}) \twoheadrightarrow \mathsf{Set} \qquad B : \mathsf{El}(P\ \mathbf{0}) \qquad n : \mathsf{El}(\mathbb{N})\\ R : (n : \mathsf{El}(\mathbb{N})) \twoheadrightarrow \mathsf{El}(P\ n) \twoheadrightarrow \mathsf{El}(P\ (\mathsf{suc}\ n))\end{array}}{\mathsf{case}_\mathbb{N}\ P\ B\ R\ (\mathsf{suc}\ n) = R\ n\ (\mathsf{case}_\mathbb{N}\ P\ B\ R\ n) : \mathsf{El}(P\ (\mathsf{suc}\ n))}$$

Let us see how we can define addition:

$$\mathsf{add} := \lambda n \to \lambda m \to \mathsf{case}_{\mathbb{N}}\ (\lambda x \to \mathbb{N})\ m\ (\lambda x \to \lambda y \to \mathsf{suc}\ y)\ n$$

It is easy to show that indeed

$$\mathsf{add} : \mathsf{El}(\mathbb{N}) \twoheadrightarrow \mathsf{El}(\mathbb{N}) \twoheadrightarrow \mathsf{El}(\mathbb{N})$$

Let us sketch the proof of $\mathsf{add}\ 1\ 2 = 3$.

$$\begin{aligned}
\mathsf{add}\ 1\ 2 &= (\lambda n \to \lambda m \to \mathsf{case}_{\mathbb{N}}\ (\lambda x \to \mathbb{N})\ m\ (\lambda x \to \lambda y \to \mathsf{suc}\ y)\ m)\ (\mathsf{suc}\ \mathbf{0})\ (\mathsf{suc}\ (\mathsf{suc}\ \mathbf{0})) \\
&= \mathsf{case}_{\mathbb{N}}\ (\lambda x \to \mathbb{N})\ (\mathsf{suc}\ (\mathsf{suc}\ \mathbf{0}))\ (\lambda x \to \lambda y \to \mathsf{suc}\ y)\ (\mathsf{suc}\ \mathbf{0}) && \textsc{Case suc} \\
&= (\lambda x \to \lambda y \to \mathsf{suc}\ y)\ \mathbf{0}\ (\mathsf{case}_{\mathbb{N}}\ (\lambda x \to \mathbb{N})\ (\mathsf{suc}\ (\mathsf{suc}\ \mathbf{0}))\ (\lambda x \to \lambda y \to \mathsf{suc}\ y)\ \mathbf{0}) && \beta\text{-}{=} \\
&= \mathsf{suc}\ (\mathsf{case}_{\mathbb{N}}\ (\lambda x \to \mathbb{N})\ (\mathsf{suc}\ (\mathsf{suc}\ \mathbf{0}))\ (\lambda x \to \lambda y \to \mathsf{suc}\ y)\ \mathbf{0}) && \textsc{Case}\ \mathbf{0} \\
&= \mathsf{suc}\ (\mathsf{suc}\ (\mathsf{suc}\ \mathbf{0})) \\
&= 3
\end{aligned}$$

In the next section we will see how we can use induction on the natural numbers to prove properties of addition.

23.4.9. Identity set

We denote the identity set (or equality set) with the $\equiv$ constant:

$$\equiv\ : (S : \mathsf{Set}) \twoheadrightarrow (x : \mathsf{El}(S)) \twoheadrightarrow \mathsf{El}(S) \twoheadrightarrow \mathsf{Set}$$

Notation. We will write $A\ \equiv_S\ B$ instead of $\equiv\ S\ A\ B$.
We add the constant refl to build elements of this set.

$$\mathsf{refl}\ : (S : \mathsf{Set}) \twoheadrightarrow (x : \mathsf{El}(S)) \twoheadrightarrow x \equiv_S x$$

We add an induction principle:

$$\begin{aligned}
\mathsf{case}_{\equiv} : {} & (S : \mathsf{Set}) \twoheadrightarrow (x : S) \twoheadrightarrow (y : \mathsf{El}(S)) \\
& \twoheadrightarrow (P : (x' : \mathsf{El}(S)) \twoheadrightarrow (y' : \mathsf{El}(S)) \twoheadrightarrow \mathsf{El}(x' \equiv_S y') \twoheadrightarrow \mathsf{Set}) \\
& \twoheadrightarrow ((z : \mathsf{El}(S)) \twoheadrightarrow \mathsf{El}(P\ z\ z\ (\mathsf{refl}\ S\ z))) \\
& \twoheadrightarrow (e : \mathsf{El}(x \equiv_S y)) \\
& \twoheadrightarrow \mathsf{El}(P\ x\ y\ e)
\end{aligned}$$

We add one equality:

$$\textsc{Case}\equiv\ \frac{\begin{array}{c} S : \mathsf{Set} \qquad a : \mathsf{El}(S) \qquad y : \mathsf{El}(S) \\ P : (x' : \mathsf{El}(S)) \twoheadrightarrow (y' : \mathsf{El}(S)) \twoheadrightarrow \mathsf{El}(x \equiv_S y) \twoheadrightarrow \mathsf{Set} \\ i : (z : \mathsf{El}(S)) \twoheadrightarrow \mathsf{El}(P\ z\ z\ (\mathsf{refl}\ S\ z)) \end{array}}{\mathsf{case}_{\equiv}\ S\ x\ y\ P\ i\ (\mathsf{refl}\ S\ x) = i\ a : \mathsf{El}(P\ x\ x\ (\mathsf{refl}\ S\ x))}$$

With this set we can build a type which states "for every natural n, we have $n + 0 = n$". Such type is as follows:

$$(n : \mathsf{El}(\mathbb{N})) \twoheadrightarrow \mathsf{El}(\mathsf{add}\ n\ \mathbf{0} \equiv_{\mathbb{N}} n)$$

If we were to prove this informally we would proceed as follows. Perform induction on n, for the base case is trivial, for the successor case $n = \mathsf{suc} m$ we have by IH that $\mathsf{add}\ m\ \mathbf{0} = m$, then we apply suc on both sides of the IH, so we have $\mathsf{suc}\ (\mathsf{add}\ m\ \mathbf{0}) = \mathsf{suc}\ m$ which is equivalent to $(\mathsf{add}\ (\mathsf{suc}\ m)\ \mathbf{0}) = \mathsf{suc}\ m$.

In order to formalize the previous proof we will first show this lemma:

$$(n : \mathsf{El}(\mathbb{N})) \twoheadrightarrow (m : \mathsf{El}(\mathbb{N})) \twoheadrightarrow \mathsf{El}(n \equiv_{\mathbb{N}} m) \twoheadrightarrow \mathsf{El}(\mathsf{suc}\ n \equiv_{\mathbb{N}} \mathsf{suc}\ m)$$

In order to show that we build a term of the above type:

$$\begin{aligned}\mathsf{lemma} := \lambda n \to \lambda m \to \lambda e \to \mathsf{case}_{\equiv}\ & \mathbb{N}\ n\ m \\ & (\lambda n' \to \lambda m' \to \lambda e' \to \mathsf{suc}\ n' \equiv_{\mathbb{N}} \mathsf{suc}\ m') \\ & (\lambda z \to \mathsf{refl}\ \mathbb{N}\ (\mathsf{suc}\ z)) \\ & e\end{aligned}$$

Let us explain the proof step by step. We start with three lambda abstractions as we have three arguments: two natural numbers and a proof of equality between them. Then we use the $\mathsf{case}_{\equiv}$ primitive to build the desired proof. We now inspect each of the applied arguments. First we have $\mathbb{N}$ because the equalities in the proof are between naturals. Second and third we have n and m because they are the two involved naturals. Then we have the property that we want to show, namely $(\lambda n' \to \lambda m' \to \lambda e' \to \mathsf{suc}\ n' \equiv_{\mathbb{N}} \mathsf{suc}\ m')$, we will refer to this term as P. After that we need to provide a term typed $(z : \mathsf{El}(S)) \twoheadrightarrow \mathsf{El}(P\ z\ z\ (\mathsf{refl}\ S\ z))$. In our case we need to replace S by $\mathbb{N}$ and P by the fourth argument. After simplifying with the β-reduction rule we get $(z : \mathsf{El}(\mathbb{N})) \twoheadrightarrow \mathsf{El}(\mathsf{suc}\ z \equiv_{\mathbb{N}} \mathsf{suc}\ z)$. We can easily see that $(\lambda z \to \mathsf{refl}\ \mathbb{N}\ (\mathsf{suc}\ z))$ has the desired type. The last argument, e, which has type $\mathsf{El}(n \equiv_{\mathbb{N}} m)$ is the equality that we will use. Finally we see that the return type is $\mathsf{El}(P\ x\ y\ e)$. After replacing P with its definition, x by and n and y by m and simplifying via the β-reduction rule we get $\mathsf{El}(\mathsf{suc}\ n \equiv_{\mathbb{N}} \mathsf{suc}\ m)$, which is what we wanted.

We are now ready to prove the theorem. The term that serves as a proof of the theorem, or in other words the term that has the specified type, is provided below.

$$\begin{aligned}\mathsf{thm} := \lambda n \to \mathsf{case}_{\mathbb{N}}\ & (\lambda n' \to \mathsf{add}\ n'\ \mathbf{0} \equiv_{\mathbb{N}} n') \\ & (\mathsf{refl}\ \mathbb{N}\ \mathbf{0}) && \text{base case} \\ & (\lambda m \to \lambda p \to \mathsf{lemma}\ (\mathsf{add}\ m\ \mathbf{0})\ m\ p) && \text{inductive case} \\ & n\end{aligned}$$

Let us break the term thm into smaller pieces and analyze them. Since we are proving that a property holds for any natural number, we start with a lambda abstraction $\lambda n \to \dots$ and then we proceed by induction on n by using the $\mathsf{case}_{\mathbb{N}}$ constant.

The first argument of $\mathsf{case}_{\mathbb{N}}$, namely $(\lambda n' \to \mathsf{add}\ n'\ \mathbf{0} \equiv_{\mathbb{N}} n')$, expresses the property that we want to prove abstracted over the term on which we perform the induction, in our case we abstract n with the variable n'. We will refer to this argument as P. It is easy to observe that P has type $\mathsf{El}(\mathbb{N}) \twoheadrightarrow \mathsf{Set}$ as required by the type of $\mathsf{case}_{\mathbb{N}}$.

Then, the second argument, which corresponds to the proof for the base case has type $\mathsf{El}(P\ \mathbf{0})$, which in our case translates into $(\lambda n' \to \mathsf{add}\ n'\ \mathbf{0} \equiv_{\mathbb{N}} n')\ \mathbf{0}$ which is the same (by β-reduction) as $\mathsf{El}(\mathsf{add}\ \mathbf{0}\ \mathbf{0} \equiv_{\mathbb{N}} \mathbf{0})$ which is the same (by CASE 0) as $\mathsf{El}(\mathbf{0} \equiv_{\mathbb{N}} \mathbf{0})$. Thus it suffices to provide the term $\mathsf{refl}\ \mathbb{N}\ \mathbf{0}$, which has the desired type.

The third argument corresponds to the proof of the inductive case. The type for this argument is[6] $((m : \mathsf{El}(\mathbb{N})) \twoheadrightarrow \mathsf{El}(P\ m) \twoheadrightarrow \mathsf{El}(P\ (\mathsf{suc}\ m)))$ which in our case translates into $((m : \mathsf{El}(\mathbb{N})) \twoheadrightarrow \mathsf{El}(\mathsf{add}\ m\ \mathbf{0} \equiv_{\mathbb{N}} n) \twoheadrightarrow \mathsf{El}(\mathsf{add}\ (\mathsf{suc}\ m)\ \mathbf{0} \equiv_{\mathbb{N}} \mathsf{suc}\ m))$. It is easy to see that the lemma we proved before will come in handy. In fact, it suffices to observe that the term $(\lambda m \to \lambda p \to \mathsf{lemma}\ (\mathsf{add}\ m\ \mathbf{0})\ m\ p)$ has the desired type. This concludes the proof of the inductive case.

[6]We renamed the variable n in this argument to m to avoid confusion with the previously bound variable n. Recall that this can be done thanks to the α-= rule.

Finally, the fourth argument is the specific natural number for which we want to perform the induction and prove the property. We just give it n, which is bound by the initial $\lambda n \to \ldots$. This concludes our proof.

23.5. Main differences with Agda's type system

In this section we briefly highlight some of the differences between LF and Agda.

Hierarchy of sets. The most important difference between LF and Agda is that Agda does not has the concepts of Type and Set, where Set : Type. For instance, in LF we do not have an answer to "what is the type of Type?". Agda solves this problem an infinite hierarchy of sets Set_0 : Set_1 : .... It is important to note that the word Type does not exist in Agda, instead, there is only the hierarchy of sets. For instance, small types such as the type of natural numbers reside in Set_0. Predicates on natural numbers reside in Set_1, and so on. For a more detailed description we refer the reader to Section 25.4 and Appendix A.7.

Extensible language. Agda allows the introduction of new types within the language. Of course, in order to avoid bogus types that could cause the soundness of the system to be compromised, the types defined by the user are subject to some restrictions. For instance, positivity checking (Section 25.9).

Pattern matching. Agda has a generic way, called pattern matching, to deconstruct terms and do case splits based on the constructors of the corresponding type. We refer the reader to Chapter 24 for more information.

Induction by proof of termination. Agda does not have explicit induction principles. In Agda, recursive calls and induction hypotheses are the same. In order to ensure that the system remains sound Agda checks for termination on the recursive calls, which is the same as saying that we use the induction hypotheses on something provably smaller.

24. Basic Agda

This part of the thesis is not meant to be an exhaustive analysis of the inner workings of Agda, as this falls out of the scope of this thesis. The original author of Agda, Ulf Norell, has suggested[1] [8] as a good reference for that purpose.

In this chapter we precisely define a moderate subset of Agda. We have tried to remain faithful to the semantics of the real Agda language, however, this is an incomplete simplification and thus is not meant to be a reference for the real Agda language.

We introduce several concepts that have cyclic dependencies and thus a linear presentation is not possible.

Definition 24.1. Identifier. An identifier is a sequence of characters which do not contain any white space or parentheses (normal `()` or curly `{}`) and furthermore it is different than all reserved keywords. Some identifier examples are `a`, `x`, `¬¬x`, `A▷B`, `A→B`, `Some-Long-Word`. For all practical purposes, we can assume we have an infinite set of identifiers.

Some of the Agda reserved keywords are `λ`, `∀`, `→`, `=`, `data`, `where`, `:`.

It is worth noting that syntactically constructors and identifiers are subject to the same rules. Agda detects constructors by using the datatype definitions in scope. For more information on constructors see the definition of a datatype in Definition 24.4.

Definition 24.2. Term/Type. An Agda term is recursively defined as shown in the figure below.

We use `x` to denote an arbitrary identifier, we use $\texttt{p}_1$, …, $\texttt{p}_n$ to denote arbitrary patterns (see Definition 24.5), we use `A`, , `B`, $\texttt{A}_1$, …, $\texttt{A}_n$ to denote arbitrary terms and we use `c` to denote an arbitrary constructor (see Definition 24.4).

term	:=	`x`	identifier;
	|	`(x : A) → B`	function type;
	|	`λ x → A`	lambda abstraction;
	|	`λ {p₁ → A₁; … ; pₙ → Aₙ}`	lambda abstraction with pattern matching;
	|	`A B`	function application;
	|	`Set ℓ`	universe ($\ell \in \omega$).

We say that an Agda term `A` is a **type** if we have `A : Set ℓ` for some `ℓ`. In Definition 24.8 we give a description of the typing relation `:`. We want to emphasize, and it is clear from the definition, that all types are also terms.

The `→` in the function type has right associativity, hence `(a : A) → (b : B) → C` is the same as `(a : A) → ((b : B) → C)`. The `→` in the lambda abstraction extends to the rightmost part, hence `λ x → λ y → A B` is the same as `(λ x → (λ y → A B))`. Function application has left associativity, hence `a b c` is the same as `(a b) c`. Also note that the `i` in `Set i` can be an identifier but for simplicity in this chapter we restrict the `i` to be an arbitrary constant natural number.

An example term:

```
λ (A : Set 0) → λ (B : Set 0) → (f : A → B) → (a : A) → B
```

[1]In correspondence via email.

Definition 24.3. Function definition. A function definition is used to bind a new[2] identifier to a term.

Note: Maybe the name "term definition" or "term binding" would be more appropriate for the concept defined here. We have decided to use the name "Function definition" since it is widely used in the field of computer science.

Below we present two schemas of function definitions:

```
x : T
x = A

y : T'
y = A'
```

The above code should read as: The identifier `x` is bound to `A`, which is a term of type `T`. Likewise, the identifier `y` is bound to `A'` which is a term of type `T'`.

Note that `A : T` and `A' : T'` must be valid according to the typing rules (see Section 24.1).

Function definitions are evaluated in order, thus, in `T'` and in `A'` we can refer to `x`. However, neither in `A` or `T` we can refer to `y`.

Recursive references are allowed in the term, thus we can refer to `x` in `A`. Likewise we can refer to `y` in `A'`.

Also note that we cannot bind the same identifier twice.

Definition 24.4. Datatype definition. Datatype definitions are used to introduce new terms/types to the language. We call datatypes the types which have been defined using a datatype definition. For instance, we would use a datatype definition to define a type representing the natural numbers.

The general form of the definition of a datatype `D` is the following:

```
data D (x₁ : P₁) … (xₖ : Pₖ) : (y₁ : Q₁) → … → (yₗ : Qₗ) → Set ℓ where
  c₁ : T₁
  …
  cₙ : Tₙ
```

Note that $k \geq 0$, $l \geq 0$ and $n \geq 0$. We distinguish the following parts of the declaration:

1. *Name*. `D` is an identifier, which is the name of the newly introduced datatype. `D` is assigned the following type and is brought into scope:

   ```
   (x₁ : P₁) → … → (xₖ : Pₖ) → (y₁ : Q₁) → … → (yₗ : Qₗ) → Set ℓ
   ```

   By bringing `D` into scope we mean that `D` can be referenced in the constructor types T_1, …, T_n, also in subsequent datatypes definitions and in terms defined after the definition of the datatype `D`.

2. *Indices*. $(y_1 : Q_1)$ … $(y_l : Q_l)$ are the indices of the datatype. For any $i \in \{1, ..., l\}$ we have that:

   a) y_i is an identifier with associated type Q_i;

   b) the type Q_i can reference x_j for any $j \in \{1, ..., k\}$;

   c) if $i > 1$ we have that the type Q_i can reference any y_j for $j < i$.

3. *Parameters*. $(x_1 : P_1)$ … $(x_k : P_k)$ are the parameters of the datatype. For every $i \in \{1, ..., k\}$ we have that:

[2] By new we mean that is has not yet been bound by another definition.

   a) `x`$_i$ is an identifier with associated type `P`$_i$;
   b) if $i > 1$ we have that the type `P`$_i$ can reference any `x`$_j$ for $j < i$.

4. *Constructors.* `c`$_1$ … `c`$_n$ are identifiers, which we call the constructors of the datatype. For every $i \in \{1, ..., n\}$ we have that:
   a) `T`$_i$ is the type of the constructor `c`$_i$.
   b) `T`$_i$ has to be of the form

      `(z`$_1$ `: B`$_1$`) → ... → (z`$_m$ `: B`$_m$`) → D x`$_1$ … `x`$_k$ `t`$_1$ … `t`$_l$

      Where for every $i \in \{1, ..., l\}$ we have that `t`$_i$ `: Q`$_i$, furthermore `t`$_i$ can refer to `z`$_j$ for any $j \in \{1, ..., m\}$.

      If we focus on the return type[3] of `c`$_i$, namely `D x`$_1$ … `x`$_k$ `t`$_1$ … `t`$_l$, we see that the first k arguments to `D` are required to be precisely the parameters of `D`, while the remaining l arguments, the indices, can be any terms `t`$_1$, …, `t`$_l$ of type `Q`$_1$, …, `Q`$_l$ respectively and may vary for each constructor. For that reason, we say that parameters are shared among all constructors, while indices are specified on a constructor basis. Refer to Appendix A.4 for a meaningful example.

   The following is fundamental: the only way to build a term of type `D x`$_1$ … `x`$_k$ `t`$_1$ … `t`$_l$ is to build a term of the form `c`$_i$ `w`$_1$ `...` `w`$_m$, for some $i \in \{1, ..., n\}$, assuming `c`$_i$ is declared to have the type `(z`$_1$ `: B`$_1$`) → ... → (z`$_m$ `: B`$_m$`) → D x`$_1$ … `x`$_k$ `t`$_1$ … `t`$_l$ and `w`$_1$, … `w`$_m$ are terms of type `B`$_1$, … `B`$_m$ respectively.

   There are no datatypes or constructors which are inherent to the language, thus, every datatype and constructor will be defined by the user in a datatype definition.

Definition 24.5. Pattern. A pattern is recursively defined as follows. We use `p`$_1$, …, `p`$_n$ to denote patterns.

pattern	:=	`c p`$_1$ … `p`$_n$	constructor of arity `n ≥ 0`;
	|	`x`	identifier.

A pattern cannot contain repeated identifiers. We define *ids*(p) to be the set of identifiers that appear in a pattern.

Note that patterns of the form `c p`$_1$ … `p`$_n$ for $n \geq 1$ must be surrounded by parentheses.

Definition 24.6. Module. A module is a sequence of function definitions and datatype definitions. Each function definition exposes the bound identifier to the subsequent definitions. Each datatype definition exposes the name of the datatype and its constructors to the subsequent definitions.

module	:=	`fundef ↵ module`	function definition;
	|	`datadef ↵ module`	datatype definition;
	|		empty.

The ↵ symbol represents a line break.

An example module which contains a definition of the `Bool` datatype and the `not` function:

```
data Bool : Set 0 where
  true : Bool
  false : Bool

not : (b : Bool) → Bool
not = λ { false → true; true → false}
```

[3]The rightmost term which is not a function type.

24.1. Contexts, and typing rules

Definition 24.7. Context. A context is a pair of (finite) sets $\langle \mathrm{T}, \Delta \rangle$. The set T consists of pairs $\langle \mathit{identifier} : \mathit{type} \rangle$. It is used to keep track of what identifiers are bound and what is their type. The set Δ consists of pairs $\langle \mathit{identifier} = \mathit{term} \rangle$ which are the identifiers which have been assigned a term through a function definition.

In order to simplify things we assume that there is no *shadowing*, which means that an identifier which is already bound in the current context cannot be bound again. Thus, the term `λ x → λ x → x` would be invalidated by this assumption since the identifier `x` is rebound by the second lambda function. This restriction is not a limiting as we can always rename our identifiers to De Bruijn indices ([11]), which guarantee this assumption.

Notation. We refer to contexts by a single letter, thus if $\Gamma = \langle \mathrm{T}, \Delta \rangle$ we abuse notation and write $a : t \in \Gamma$ instead of $\langle a : t \rangle \in \mathrm{T}$ and $a = t \in \Gamma$ instead of $\langle a = t \rangle \in \Delta$. Also, we use $\Gamma; t : A$ as short for $\langle \mathrm{T} \cup \{\langle t : A \rangle\}, \Delta \rangle$.

Definition 24.8. Well-typed term (and patterns). We say that a term t is well-typed in context Γ if $\Gamma \vdash t : A$ for some type A. The rules for $\vdash$ are presented below. We also define the relation $\vdash_{\mathrm{P}}$, which is for typing patterns. For that purpose we need an auxiliary function definition, τ, which is defined afterwards.

The structure of a module implicitly assigns a context to each term in it. We usually use the concept of well-typed term in the context of a module, in that case, we implicitly refer to the context assigned by the structure of the module. See Definition 24.9 for a thorough explanation.

$$\textsc{Id}\quad \frac{t : A \in \Gamma}{\Gamma \vdash t : A} \qquad \textsc{Level}\quad \frac{}{\Gamma \vdash \mathit{Set}\ i : \mathit{Set}\ (i+1)} \qquad \textsc{Arrow}\quad \frac{\Gamma \vdash A : \mathit{Set}\ i \qquad \Gamma; x : A \vdash B : \mathit{Set}\ j}{\Gamma \vdash (x : A) \to B : \mathit{Set}\ (i \sqcup j)}$$

$$\textsc{Abstraction}\quad \frac{x : A; \Gamma \vdash t : B}{\Gamma \vdash \lambda x \to t : (x : A) \to B} \qquad \textsc{Application}\quad \frac{\Gamma \vdash f : (x : A) \to B \qquad \Gamma \vdash a : A}{\Gamma \vdash f\ a : B[x \mapsto t]}$$

$$\textsc{Pattern abstraction}\quad \frac{\begin{array}{c} \text{Let } \overline{D} := D\ x_1\ \dots\ x_{\mathrm{n}}\ t_1\ \dots\ t_{\mathrm{n}} \\ \Gamma \vdash_{\mathrm{P}} p_1 : \overline{D} \qquad \dots \qquad \Gamma \vdash_{\mathrm{P}} p_{\mathrm{n}} : \overline{D} \\ \Gamma \cup \tau(\Gamma, \overline{D}, p_1) \vdash s_1 : B[x \mapsto p_1] \qquad \dots \qquad \Gamma \cup \tau(\Gamma, \overline{D}, p_{\mathrm{n}}) \vdash s_{\mathrm{n}} : B[x \mapsto p_{\mathrm{n}}] \end{array}}{\Gamma \vdash \lambda\ \{p_1 \to s_1;\ \dots\ ;\ p_{\mathrm{n}} \to s_{\mathrm{n}}\} : (x : \overline{D}) \to B}$$

Figure 24.1.: Typing rules for terms.

Identifier

$$\frac{}{\Gamma \vdash_{\mathrm{P}} x : A}$$

Constructor

$$\frac{c : (b_1 : B_1) \to ... \to (b_{\mathrm{n}} : B_{\mathrm{n}}) \to D\ x_1\ ...\ x_{\mathrm{n}}\ t_1\ ...\ t_{\mathrm{n}} \in \Gamma \qquad \forall i \in \{1, ..., n-1\}.\ \Gamma \vdash_{\mathrm{P}} p_{\mathrm{i}} : B_{\mathrm{i}}[b_1 \mapsto p_1, ..., b_{i-1} \mapsto p_{i-1}]}{\Gamma \vdash_{\mathrm{P}} c\ p_1\ ...\ p_{\mathrm{n}} : D\ x_1\ ...\ x_{\mathrm{n}}\ t_1\ ...\ t_{\mathrm{n}}}$$

Figure 24.2.: Typing rules for patterns.

Let $\overline{D} := D\ x_1\ ...\ x_{\mathrm{n}}\ t_1\ ...\ t_{\mathrm{n}}$. We now define $\tau(\Gamma, \overline{D}, p)$, which is the set of identifiers bound by pattern p paired with their respective types. Assume that $\Gamma \vdash_{\mathrm{P}} p : \overline{D}$. Finally, let τ be defined as follows:

$\tau(\Gamma, \overline{D}, x) := \{x : \overline{D}\}$ Identifier

$\tau(\Gamma, \overline{D}, c\ p_1\ ...\ p_{\mathrm{n}}) := \tau(\Gamma, p_1, B_1) \cup ... \cup \tau(\Gamma, p_{\mathrm{n}}, B_{\mathrm{n}})$ Constructor

assuming $c : (b_1 : B_1) \to ... \to (b_{\mathrm{n}} : B_{\mathrm{n}}) \to \overline{D} \in \Gamma$

Definition 24.9. Scoping and type checking a module. We understand by *scoping* the process of implicitly assigning a context to each part of the module. We understand by *type checking* the process of checking that all terms in a module are well-typed (in their corresponding context) and respect the typing annotations. These processes are tightly tied and thus we describe them together.

It may be worth noticing that sometimes the annotation `x : A` may be redundant since the type of `t` can be automatically inferred to be `A` from the rules. Type inference is widely used in the real Agda language, however, in this presentation we skip it for simplicity.

However, for simplicity we do not differentiate between type inference and type checking. assume that the annotation is always required.

The process of type-checking a module is as follows:

1. At the beginning of a module we start with an empty context $\Gamma := \emptyset$.

2. We look at the next definition.

   - If it is a **function definition**, it is of the form

     ```
     x : A
     x = t
     ```

     Check there is some ℓ such that $\Gamma \vdash A : \mathit{Set}\ \ell$. Then let $\Gamma' := \Gamma; x : A$ and check $\Gamma' \vdash t : A$.

     We repeat step 2 with context Γ'.

   - If it is a **datatype definition**, it is the form

     ```
     data D (x₁ : P₁) … (xₖ : Pₖ) : (y₁ : Q₁) → … → (yₗ : Qₗ) → Set ℓ where
       c₁ : T₁
       …
       cₙ : Tₙ
     ```

     where each $\mathtt{T_i}$ is of the form

     ```
     (z₁ : B₁) → … → (zₘ : Bₘ) → D x₁ … xₖ t₁ … tₗ
     ```

     We check that:

a) For each parameter `xi : Pi` check that for some $\varepsilon \leq \ell$ we have

$$\Gamma; x_1 : P_1; ...; x_{i-1} : P_{i-1} \vdash P_{\mathrm{i}} : Set\ \varepsilon.$$

b) Then let

$$\begin{aligned}\Gamma' :=& \Gamma; x_1 : P_1; ...; x_{\mathrm{k}} : P_{\mathrm{k}};\\ & D : (x_1 : P_1) \to ... \to (x_{\mathrm{k}} : P_{\mathrm{k}}) \to (y_1 : Q_1) \to ... \to (y_{\mathrm{l}} : Q_{\mathrm{l}}) \to Set\ \ell.\end{aligned}$$

For each $i \in \{1, ..., n\}$ check that

$$\Gamma' \vdash (z_1 : B_1) \to ... \to (z_{\mathrm{m}} : B_{\mathrm{m}}) \to D\ x_1\ ...\ x_{\mathrm{k}}\ t_1\ ...\ t_{\mathrm{l}} : Set\ \ell.$$

Then let

$$\begin{aligned}\Gamma'' :=& \Gamma;\\ & D : (x_1 : P_1) \to ... \to (x_{\mathrm{k}} : P_{\mathrm{k}}) \to (y_1 : Q_1) \to ... \to (y_{\mathrm{l}} : Q_{\mathrm{l}}) \to Set\ \ell;\\ & c_1 : (z_1^1 : B_1^1) \to ... \to ({z^1}_{\mathrm{m}} : {B^1}_{\mathrm{m}}) \to D\ x_1\ ...\ x_{\mathrm{k}}\ t_1^1\ ...\ {t^1}_{\mathrm{l}};\\ & \vdots\\ & c_{\mathrm{n}} : (z_1^n : B_1^n) \to ... \to ({z^n}_{\mathrm{m}} : {B^n}_{\mathrm{m}}) \to D\ x_1\ ...\ x_{\mathrm{k}}\ t_1^n\ ...\ {t^n}_{\mathrm{l}}\end{aligned}$$

and continue to step 2 with context Γ''.

24.2. Normalization

Normalization is refers to the process of simplifying or evaluating a term via rewrite rules.

In real Agda normalization is done at the same time as type-checking via an involved algorithm (see Section 3.3.2 of [29]). Here we present a collection normalization rules which are detached from the type-checking process. Our aim is to provide an intuition of how well-typed terms are simplified automatically in Agda rather than giving details of the algorithm.

The reader may be already acquainted with the β-REDUCTION rule, which is present in untyped lambda calculus, the most basic form of lambda calculus. Here we present the mentioned rule among others. We use the notation $\Delta \vdash_N t \downarrow t'$ to say that term t normalizes to term t' in context Δ. We extend the definition of context to also contain all the pairs $\langle identifier = term \rangle$ which are defined in a function definition above in the module. To be more precise, now a context consists of two sets: One contains the typing relations $\langle identifier : term \rangle$ and the other contains the binding relations $\langle identifier = term \rangle$.

Normalization can happen in nested terms. For instance, if we have that $t \downarrow t'$ then $\lambda x \to t \downarrow \lambda x \to t'$. Likewise for the other kinds of terms.

In the rule MATCH we use the notation $\exists^{min} i$ to mean that in case there exist multiple values for i such that the function $match(p_{\mathrm{i}}, b)$ succeeds and returns a substitution σ, then we must take the minimum of such is. In other words, we check patterns in order and we use the first that succeeds.

We use a, b, c, t to denote arbitrary terms and x, f to denote arbitrary identifiers.

$$\frac{}{\Delta \vdash_N (\lambda x \to t)\ a \downarrow t[x \mapsto a]}\ \beta\text{-REDUCTION} \qquad \frac{\Delta \vdash_N a \downarrow b \qquad \Delta \vdash_N b \downarrow c}{\Delta \vdash_N a \downarrow c}\ \text{TRANSITIVITY} \qquad \frac{f = t \in \Delta}{\Delta \vdash_N f \downarrow t}\ \text{DEFINITION}$$

$$\frac{\Delta \vdash_N a \downarrow b \qquad \exists^{min} i.match(p_\text{i}, b) = \sigma}{\Delta \vdash_N \lambda\ \{p_1 \to t_1; ...; p_\text{n} \to t_\text{n}\}\ a \downarrow \sigma(t_\text{i})}\ \text{MATCH}$$

Figure 24.3.: Normalization rules for terms.

Here we define the partial function *match*, which takes a pattern and a term, then either fails or returns a substitution. We represent a substitution by a set of pairs of the form $\langle x \mapsto t\rangle$, which means "replace identifier x by term t". Keep in mind that a pattern does not contain repeated identifiers, so the result (if it succeeds) of *match* is a proper function.

$$match(x,t) := \{\langle x \mapsto t\rangle\}$$

$$match(c\ p_1\ ...\ p_\text{n}, c'\ t_1\ ...\ t_\text{n}) := \begin{cases} \bigcup_i (match(p_\text{i}, t_\text{i})) & \text{if } c = c' \text{ and all recursive calls succed;} \\ \text{fail} & \text{otherwise.} \end{cases}$$

Some examples for the β-REDUCTION rule:

$\lambda y \to (\lambda x \to x)\ y \downarrow \lambda y \to y$

$(\lambda x \to x)\ a \downarrow a$

To see examples for the MATCH rule assume we have the following definition in scope:

```
data Nat : Set 0 where
  zero : Nat
  suc : Nat → Nat

plus : Nat → Nat → Nat
plus = λ { zero → (λ b → b); (suc a) → (λ b → suc (plus a b) ) }
```

Then see that the term `plus zero` normalizes to the identity function:

$plus\ zero \downarrow \lambda b \to b$

As another example, see that the term `plus (suc zero)` normalizes to `λ b → suc b`. We present this example step by step.

$$\begin{array}{rr} plus\ (suc\ zero) \downarrow & \text{DEF} \\ \lambda\ \{zero \to (\lambda b \to b);\ (suc\ a) \to (\lambda b \to suc\ (plus\ a\ b))\}\ (suc\ zero) \downarrow & \text{MATCH} \\ \lambda b \to suc\ (plus\ zero\ b) \downarrow & \text{DEF} \\ \lambda b \to suc\ ((\lambda b' \to b')\ b) \downarrow & \beta\text{-REDUCTION} \\ \lambda b \to suc\ b & \end{array}$$

Theorem 24.10. *Normalization is type-preserving.*

$$\frac{\Gamma \vdash_N t \downarrow t' \qquad \Gamma \vdash t : A}{\Gamma \vdash t' : A}$$

Proof. By an easy proof by induction. ☐

24.3. Totality

In order to be a sound system, Agda requires all of its terms to be total. Thus, it needs to assure that:

1. Every lambda abstraction with pattern matching of the form `λ {p₁ → A₁; … ; pₙ → Aₙ}` must have a set of exhaustive patterns `p₁, …, pₙ`. For instance if we have the following:

   ```
   data Bool : Set 0 where
     true : Bool
     false : Bool

   wrong : (b : Bool) → Bool
   wrong = λ { true → true }
   ```

   Then the definition of `wrong` is rejected since it does not have the `false` pattern. The coverage algorithm (the algorithm which checks the exhaustivity of patterns) is described in [29].

2. There are no infinite loops. For instance, the following definition is rejected.

   ```
   loop : (A : Set 0) → A
   loop = λ A → loop A
   ```

   In order to do so Agda analyses every recursive call and tries to find a well-founded order on its arguments. If it succeeds the recursive call is considered save, otherwise it is rejected. The Agda online documentation ([3]) references [1] as the basis of the termination checker implemented in Agda. For instance, the checker is sophisticated enough to accept the definition of the Ackermann function ([3]), which is non-primitive recursive.

25. Agda tutorial

25.1. BHK interpretation of propositional logic

In this section we present an informal tutorial of Agda. We introduce new syntax by means of example and we introduce new concepts appealing to the reader intuition. We guide the reader to Chapter 24 for a more precise definition of the language. In the official Agda reference ([3]), the informal approach is always preferred.

As we have already mentioned, Agda is based on an intuitionistic type theory with dependent types that extends Per Martin-Löf's type theory ([23]). Agda's constructive nature suggests that a fitting way to start the introduction is through the BHK interpretation of intuitionistic logic ([4]). We start with propositional logic. During this part, the reader will notice that the first steps in this tutorial are going to be reminiscent of the embedding of propositional logic into LF we presented in Section 23.4. Later in the chapter (Section 25.5) we will continue with first-order logic and explain how to represent relations and quantifiers in Agda.

The BHK interpretation states that:

1. A proof of $A \to B$ is an algorithm that transforms an arbitrary proof of A into a proof of B;
2. A proof of $A \land B$ is a proof of A and a proof of B;
3. A proof of $A \lor B$ is an algorithm telling to which of A or B we commit to and according to that, a proof of A or a proof of B;
4. Nothing is a proof of $\bot$;
5. $\top$ is always true and provable.

According to the BHK interpretation a proof of $A \to A$ is an algorithm implementing the identity function. We can implement this algorithm in Agda by writing the identity lambda term:

```
λ a → a
```

More generally, in order to build lambda terms we will use the syntax `λ arg₁ arg₂ … → term`. The term `λ a → a` is well-formed. However, in Agda we must give a name to all the terms that we define so we can refer to them in other parts of our code. We give a name to a term by prepending `name =` to the term. We name our term `id`:

```
id = λ a → a
```

When the term that we are defining is a lambda term, we are allowed to move the arguments to the left of the = sign. Furthermore, we should write the type of the function using the `:` symbol. The `:` symbol denotes the typing relation between terms and types, hence `a : A` reads as "term `a` has type `A`". Sometimes we also say that "the term `a` is a proof of `A`".

```
id : (A : Set) → A → A
id A a = a
```

The definition `id A a = a` should be read as $\mathsf{id}(A)(a) := a$. The definition `id : (A : Set) → A → A` should be read as follows: Given an arbitrary type `A` and a term of type `A`, it returns a term of type `A`. It may be helpful to the reader to identify the Agda type `(A : Set) → A → A` with the LF type $(A : \mathsf{Set}) \twoheadrightarrow \mathsf{El}(A) \twoheadrightarrow \mathsf{El}(A)$. The Agda type checking algorithm is in charge of checking the implementation of `id` matches the specified type. For instance, if we wrote the following:

```
id-bad : (A : Set) → A → A
id-bad A a = A
```

Agda would complain that `A` is of type `Set` and not of type `A`.

Since the argument named `A` will always be exactly the type of the second argument of `id`, Agda can infer it and thus it is recommended to use an implicit argument (see Appendix A.3).

```
id2 : {A : Set} → A → A
id2 a = a
```

For instance, assume we have a term `t` of type `B` and we want to apply the identity function to it. If we use `id` we would write `id B t`. On the other hand, using `id2` we would write `id2 t`. We observe that the argument `{A : Set}` is inferred from `t` and is not needed to be explicitly given.

Let us identify propositional formulas which contain only the implication connective with Agda types. We define the map $*$ as follows:

- If p is a variable, then $p^* :=$ `P`, where `P : Set`.
- $(A \to B)^* := A^*$ `→` B^*. Note that the `→` on the right of the symbol $:=$ is Agda's arrow type.

Then, if F is a propositional formula with variables $p_1, ..., p_n$ we define

$$[\![F]\!] := \texttt{\{P}_1 \texttt{ : Set\} → … → \{P}_n \texttt{ : Set\} → F*}$$

Note that in Agda we can write `{P₁ … Pₙ : Set} → F*` instead of `{P₁ : Set} → … → {Pₙ : Set} → F*`.

We will say that an Agda type `T` is a theorem if we can provide a function definition of type `T`. In the simplest form, the definition will be of the following form:

```
thm : T
thm = proof
```

Where `thm` is the name of the theorem. For the general form we refer the reader to the Agda documentation in Appendix A.1.

Let us now see that $[\![A \to A]\!]$ is a theorem in Agda. It suffices to observe that $[\![A \to A]\!]$ is equal to `{A : Set} → A → A`, which is the type of `id2`. Hence `id2` is indeed a proof that $[\![A \to A]\!]$ is an Agda theorem. However, it is more than that. Agda is a programming language, so the programs that we define in Agda are executable. For instance, we can ask the system to evaluate the expression `id2 t`, which evaluates to `t`, as expected. The capability of evaluating expressions reflects the fact that the `id2` function we defined is a perfect candidate to be a proof of $A \to A$ according to the BHK interpretation. How the evaluation of expression works falls out of the scope of this tutorial. We refer the reader to the Agda documentation ([3]) for that.

Let us see another example involving functions. The following Agda definition shows that $[\![(A \to (B \to C)) \to (B \to (A \to C))]\!]$ is an Agda theorem.

```
commute : {A B C : Set} → (A → B → C) → B → A → C
commute f b a = f a b
```

We see that `commute` has three explicit arguments, namely `f : A → B → C`, `b : B` and `a : A`. Then in the right hand side of the = sign we have `f a b`, which should read as "term `f` applied to `a` and `b`". Notice how function application is denoted by juxtaposition and has left associativity.

Let us now put our attention on proofs that revolve around conjunction. First, we need to define a new type. In Agda we can introduce new types by means of a datatype definition (we refer the reader to Definition 24.4 for a precise definition). Recall from Chapter 24 that we call datatypes the types which have been defined by the user by means of a datatype definition.

The definition below defines the pair datatype (the semantics of this datatype are analogous to the pair set we defined for LF in Section 23.4.5).

```
data _×_ (A B : Set) : Set where
  _,_ : A → B → A × B
```

A pair type is in rough terms the type of a tuple. More precisely, the definition defines a new datatype called `_×_`. It has two parameters: `(A B : Set)`. These parameters are the types of the components of the pair. It has a single constructor named `_,_`. Underscores are used to denote infix operators. Thus we have that `_,_ A B` is the same as `A , B`. We observe that the type of the constructor `_,_` is `A → B → A × B`. This tells us that `_,_` is a constructor that takes an argument of type `A`, an argument of type `B`, and then returns a term of type `A × B`. Recall that when we *pattern match* (or deconstruct) a term we will have a case for each possible constructor of the type of that term.

We proceed by extending the map $*$ with the following clause:

$$(A \wedge B)^* := A^* \ \times \ B^*$$

We can show that $[\![A \to (B \to A \wedge B)]\!]$ is a theorem in Agda by simply applying the constructor `_,_`:

```
p1 : {A B : Set} → A → B → A × B
p1 a b = a , b
```

In order to show that $[\![A \wedge B \to B \wedge A]\!]$ is a theorem we need to access to components of the argument. We can do so by deconstructing the argument with pattern matching as shown below. Note that when we use pattern matching on an argument we need to put parentheses around it.

```
swap : {A B : Set} → A × B → B × A
swap (a , b) = b , a
```

We proceed by giving some more examples involving conjunction.

We show that we can prove $[\![A \wedge B \to A]\!]$ and $[\![A \wedge B \to B]\!]$. It suffices to pattern match on the argument to access to corresponding component and return it.

```
proj₁ : {A B : Set} → A × B → A
proj₁ (a , b) = a

proj₂ : {A B : Set} → A × B → B
proj₂ (a , b) = b
```

If we ask Agda to evaluate an expression of the form `proj₁ (a , b)` the result will be `a`.

Observe how we can use `proj₁` and `proj₂` to provide an alternative proof of $[\![A \wedge B \to B \wedge A]\!]$:

```
swap' : {A B : Set} → A × B → B × A
swap' ab = (proj₁ ab) , (proj₂ ab)
```

We show that we can prove $[\![(A \to B \to C) \to ((A \wedge B) \to C)]\!]$.

```
p3 : {A B C : Set} → (A → B → C) → (A × B) → C
p3 f (a , b) = f a b
```

As we can see, we use pattern matching to extract the components of the pair and then apply `f` to them.

Now that we are familiarized with the pair type we can proceed by exploring disjunction.

In order to express options we can define a new datatype called sum type thus (the semantics of this datatype are analogous to the disjoint union set we defined for LF in Section 23.4.4):

```
data _⊎_ (A B : Set) : Set where
  inj₁ : A → A ⊎ B
  inj₂ : B → A ⊎ B
```

We see that the main difference with respect to the pair type is that now we have two constructors which we named `inj₁` and `inj₂`. The constructor `inj₁` is used to build a term of type `A ⊎ B` by providing a term with type `A`. The constructor `inj₂` builds a term of type `A ⊎ B` given a term of type `B`.

We extend the $*$ map with the following clause:

$$(A \vee B)^* := A^* \uplus B^*$$

We show how we can prove $[\![A \vee B \to B \vee A]\!]$.

```
p4: {A B : Set} → A ⊎ B → B ⊎ A
p4 (inj₁ a) = inj₂ a
p4 (inj₂ b) = inj₁ b
```

We observe that since we have the `⊎` type has two constructors, when pattern matching against an argument of type `A ⊎ B` we need to define two cases, one for each constructor. In the first case, when we have `inj₁ a` on the left of the equal sign we know that `a` is of type `A` by the definition of the `inj₁` constructor. Hence, we can build a term of type `B ⊎ A` by applying `inj₂` to `a`. The second case is symmetric to the first case.

Let us see another example, we show $[\![(A \vee B) \to (A \to C) \to (B \to C) \to C]\!]$.

```
p5 : {A B : Set} → A ⊎ B → (A → C) → (B → C) → C
p5 (inj₁ a) f g = f a
p5 (inj₂ b) f g = g b
```

The reader may have raised the following question in their mind: when we pattern match an argument of type `A ⊎ B`, what if we do not include one of the cases? In other words, what happens if we try to prove $[\![A \vee B \to A]\!]$?

```
wrong : {A B : Set} → A ⊎ B → A
wrong : (inj₁ a) = a
```

The answer is that Agda rejects the above definition. As explained in Section 24.3 Agda requires all the definitions to be total. As such, when we split an argument into cases using pattern matching, it checks that the cases are exhaustive.

We proceed with $\perp$. We can define a datatype which we call *bottom type* or *empty type*.

```
data ⊥ : Set where
```

Notice that ⊥ has no constructors and hence it is impossible to construct a term with type ⊥. The bottom type is specially useful to define negation, which we define in the following way:

```
¬ : Set → Set
¬ A = A → ⊥
```

We extend the $*$ map with the following clauses (note that the ⊥ and ¬ on the left of the symbol := refer to the Agda symbols):

$$\bot^* := \bot$$
$$(\neg A)^* := \neg(A^*)$$

The principle of explosion (*ex falso quodlibet*) $[\![\bot \to A]\!]$ can be proved as shown below:

```
explosion : {A : Set} → ⊥ → A
explosion ()
```

We see that we have a new pattern: `()`. Agda uses the pattern `()` to denote the impossible pattern. When one of the arguments matches the impossible pattern then there is no need to provide a definition of the function. In this case we match the impossible pattern because the type ⊥ has not constructors. For more information on the absurd pattern we refer the reader to Appendix A.2.

As we are in an intuitionistic logic then we cannot show $[\![\neg\neg A \to A]\!]$ nor $[\![A \vee \neg A]\!]$[1]:

```
¬¬elim : {A : Set} → ¬ (¬ A) → A
¬¬elim = ?              -- not provable

excluded-middle : {A : Set} → A ⊎ (¬ A)
excluded-middle = ?     -- not provable
```

However, we can show that $[\![A \to \neg\neg A]\!]$ and $[\![\neg\neg\neg A \to \neg A]\!]$:

```
¬¬intro : {A : Set} → A → ¬ (¬ A)
¬¬intro a ¬a = ¬a a

¬¬¬elim : {A : Set} → ¬ (¬ (¬ A)) → ¬ A
¬¬¬elim ¬¬¬a a = ¬¬¬a (¬¬intro a)
```

We can build an alternative proof of $[\![\neg\neg\neg A \to \neg A]\!]$ that does not use `¬¬intro` by replacing it with a lambda term:

```
¬¬¬elim' : {A : Set} → ¬ (¬ (¬ A)) → ¬ A
¬¬¬elim' ¬¬¬a a = ¬¬¬a (λ ¬a → ¬a a)
```

Consider the proof of $[\![A \to \neg A \to B]\!]$.

```
p6 : {A B : Set} A → ¬ A → B
p6 a ¬a = explosion (¬a a)
```

[1]Observe that in the Agda code we use the symbol ¬ in the name of the term `¬¬elim`. Here the ¬ symbol is just part of the name and serves the same purpose of any other character. Hence it is important to note that `¬a` (just a name with the ¬ character) is different from `¬ a` (the negation of `a`).

We have two arguments, `a : A` and `¬a : ¬ A`. It is easy to see that the type of `¬a a` is ⊥. Also, recall that the type of `explosion` is[2] `{C : Set} → ⊥ → C`. Then, in the definition above, since we expect a term of type `B`, Agda can infer that implicit argument `{C : Set}` is equal to `B` and thus `explosion (¬a a)` has type `B`. For illustrative purposes, let us rewrite the same proof but with the implicit arguments made explicit. We can make implicit arguments explicit by surrounding them with `{}` as shown below:

```
p6' : {A B : Set} A → ¬ A → B
p6' {A} {B} a ¬a = explosion {B} (¬a a)
```

This concludes the first part of the introduction.

25.2. Booleans and case analysis

The *true or false* concept is ubiquitous in computer science and in logic. In this section we show how we can define a datatype that represents this dichotomy and we give a small introduction to case analysis through pattern matching.

In Agda we can define the `Bool` type in a similar fashion to the disjunction type we defined before.

```
data Bool : Set where
  true : Bool
  false : Bool
```

As a simple example, see how we can define the `not` and `and` Boolean operators using pattern matching:

```
not : Bool → Bool
not false = true
not true = false

and : Bool → Bool → Bool
and false b = false
and true b = b
```

We proceed by defining equality for the `Bool` type[3]. We use the symbol ≡ because = is reserved for Agda.

```
data _≡_ : Bool → Bool → Set where
  t≡t : true ≡ true
  f≡f : false ≡ false
```

We see that the type of _≡_ is `Bool → Bool → Set`. We say that _≡_ is an *indexed datatype*, in this case with two `Bool` indices. In contrast to parameters (recall the definitions of _×_ and _⊎_) which are shared among all constructors, indices are specified on a constructor basis.

Let us prove the following property:

```
notnot : (b : Bool) → not (not b) ≡ b
notnot true = t≡t
notnot false = f≡f
```

[2]we have renamed the bound variable `A` to `C` to avoid confusion with the `A` in `A → ¬ B → B`.

[3]This definition is just for illustrative purposes. It is possible to define generic equality as described in Section 25.6.

There are a number of things that are worth mentioning. First, we see that we refer to `b` on the returning result `not (not b) ≡ b`, which is possible in virtue of dependent types. Then we see that we pattern match on `b` and thus we need to fill out two cases. We could make an analogy with a hand written proof by cases. The case split with pattern matching allows Agda to know via *normalization* that in the `true` case we must provide a term (proof) of type `true ≡ true`, which we can provide using the `t≡t` constructor. We proceed analogously in the `false` case. Agda normalizes the terms when possible, for instance the term `not (not b)` is already normalized because we cannot apply any rule. On the other hand the term `not (not true)` can be normalized to `not false` and further normalized to `true` by using the definition of `not`. For further information on normalization refer to [29]. Another thing to notice is that we use the same construction (i.e. an Agda function definition) to provide function definitions, like `not`, and theorems, like `notnot`.

Pattern matching (case analysis) is ubiquitous in Agda, be it in definitions or in proofs. We show some more examples below:

```
p1 : (b : Bool) → and false b ≡ false
p1 b = f≡f

p2 : (b : Bool) → and b false ≡ false
p2 true = f≡f
p2 false = f≡f
```

See that in the first case we did not need to do pattern matching while in the second we had to. This is due to how the definition of `and` is written which in our case performs pattern matching on the first argument.

25.3. Naturals and induction

In this section we will have a look at the simplest possible recursive structure, the natural numbers. In Agda natural numbers can be defined in the following way:

```
data Nat : Set where
  zero : Nat
  suc : Nat → Nat
```

The definition should be intuitive enough for the reader at this point. We can represent the number 1 with the term `suc zero`, the number 2 with `suc (suc zero)` and so on.

Let us continue by defining the equality relation for natural numbers[4].

```
data _≡_ : Nat → Nat → Set where
  z≡z : zero ≡ zero
  s≡s : {a b : Nat} → a ≡ b → suc a ≡ suc b
```

We see that it has a similar structure to the datatype for Boolean equality. The only difference is that the `s≡s` constructor requires a proof of `a ≡ b` as an argument. Let us show that every natural number is equal to itself.

```
refl : (n : Nat) → n ≡ n
refl zero = z≡z
refl (suc n) = s≡s (refl n)
```

[4]Agda does not allow overloading of symbols so we would need to use a different name other than `_≡_` to avoid the clash with the equality relation of Booleans that we defined before.

We see that we pattern match on `n`, for the `zero` case we give the `z≡z` constructor. For the `suc` case we need to provide a proof of `suc n ≡ suc n`. By performing a recursive call with `n` as argument we get a proof of `n ≡ n`, then we can use the constructor `s≡s` to build a term of type `suc n ≡ suc n`. It can be enlightening to observe that in a proof by induction, such as the previous one, a recursive call plays the role of an induction hypothesis.

An inexperienced Agda user might try the following:

```
refl' : (n : Nat) → n ≡ n
refl' zero = z≡z
refl' (suc n) = refl' (suc n)
```

While the types match we see that in the inductive case we perform a recursive call on the same argument and thus we get an infinite loop. Agda has a termination checker that rejects proofs where termination cannot be assured and thus rejects the previous definition. We know that termination is an undecidable problem hence it is inevitable that Agda will reject some programs that in fact would always terminate. For more information on Agda's termination checker refer to [3, 29].

We now define addition on natural numbers:

```
_+_ : Nat → Nat → Nat
zero + b = b
(suc a) + b = suc (a + b)
```

Proving associativity can be achieved by means of an inductive proof following a similar structure as before. For the base case we use the `refl` property proved above.

```
assoc : (a b c : Nat) → (a + b) + c ≡ a + (b + c)
assoc zero b c = refl (b + c)
assoc (suc a) b c = s≡s (assoc a b c)
```

Consider the following example involving negation. Keep in mind that `¬ (n ≡ suc n) = n ≡ suc n → ⊥`.

```
p1 : (n : Nat) → ¬ (n ≡ suc n)
p1 zero ()
p1 (suc n) (s≡s x) = p1 n x
```

For the base case we have the impossible pattern because when `n = zero` the second argument is supposed to have the type `zero ≡ suc zero` which is not unifiable with any type of a constructor and thus we get the empty pattern. For more information on Agda unification refer to [29].

Finally, let us focus on proving commutativity of addition, which is a more involved example. We first prove transitivity of equality, which is proved by an easy induction.

```
trans : {a b c : Nat} → a ≡ b → b ≡ c → a ≡ c
trans z≡z z≡z = z≡z
trans (s≡s x) (s≡s y) = s≡s (trans x y)
```

Notice how there are two missing cases, that is, `z≡z` with `s≡s` and vice versa. We are allowed to do that because Agda was able to detect the empty pattern. We could have also omitted the `p1 zero ()` case in the theorem above.

We proceed by proving two lemmas by an easy induction:

```
zero-r : (a : Nat) → a ≡ (a + zero)
zero-r zero = z≡z
zero-r (suc a) = s≡s (zero-r a)
```

```
suc-r : (a b : Nat) → suc (a + b) ≡ (a + suc b)
suc-r zero b = refl (suc b)
suc-r (suc a) b = s≡s (suc-r a  b)
```

At last, we put all the pieces together to prove our theorem:

```
+commut : (a b : Nat) → (a + b) ≡ (b + a)
+commut zero b = zero-r b
+commut (suc a) b = trans (s≡s (+commut a b)) (suc-r b a)
```

For the base case we must prove `0 + b ≡ b + 0` which normalizes to `b ≡ b + 0` and then we can use our `zero-r` lemma. For the inductive case we must prove `suc a + b ≡ b + suc a` which normalizes to `suc (a + b) ≡ b + suc a`. By IH we know that `a + b ≡ b + a` so by `s≡s` we get `suc (a + b) ≡ suc (b + a)`. Then by our lemma `suc-r` we get `suc (b + a) ≡ b + suc a`. Finally by transitivity we get the desired `suc (a + b) ≡ b + suc a`.

We hope that at this point the user has a grasp of how properties can be proved in Agda.

25.4. Universe hierarchy

In Agda every well-typed term is assigned a type. For instance, the type of `true` is `Bool` and the type of `0` is `Nat`. As we have seen before, in a dependent type theory we are allowed to mix types and terms, hence `Nat` is a term in itself and must be assigned a type. Agda calls the type of (small) types `Set`, hence we have that `Nat` has type `Set`. But then `Set` is also a term an must be assigned a type as well. Could we have that the type of `Set` is `Set`? No. The first version of Martin-Löf's type theory ([22]) had an axiom stating that there is a type of all types and thus we would have that the type of `Set` is `Set`. However Girard showed ([12]) that having `Set : Set` allowed the Burali-Forti paradox[5] to be encoded in the theory, and thus the relation `Set : Set` needs to be rejected. In order to avoid such inconsistency Agda builds a hierarchy of universes where small types such as `Nat` and `Bool` are assigned the type `Set 0` and then for every $i \in \omega$ we have that `Set i : Set i+1`. Notice however, that `Set i : Set i+1` is true while `Set i : Set i+n` does not hold for $n > 1$. In Agda we write `Set` instead of `Set 0`. When the level is a constant natural number we can also write `Set₁`, `Set₂`, etc. instead of `Set 1`, `Set 2`, etc.

It is possible to combine types of different universe levels. The biggest type is the one that counts. For instance:

```
function : Set₃ → Set₁ → Set₃
function A B = A → B
```

The typing rule is analogous for product and sum types.

Agda provides a primitive[6] type for universe levels called `Level`. Essentially it is the same as `Nat` (we have `lzero` for the base level and `lsuc` for the successor level), but it is designed to work as a universe index. Having the `Level` type allows us to write universe polymorphic functions. See the same function as before, but now with universe polymorphism.

```
function' : {a b : Level} → Set a → Set b → Set (a ⊔ b)
function' A B = A → B
```

[5]The assumption that there is a set of all ordinal numbers leads to a contradiction.
[6]*primitive* means that it is built in the language and it cannot be defined by the user.

The `_⊔_` operator is a primitive operator of type `Level → Level → Level` that normalizes to the the maximum of its two operands.

Most of the functions that we have defined before should be rewritten to be universe polymorphic if possible. For instance, we can now rewrite the identity function thus:

```
id : {a : Level} {A : Set a} → A → A
id a = a
```

In the most recent version of Agda (2.6.1) there is an option to enable *universe cumulativity* ([3]). This extension adds the typing rule $Set_i : Set_j$ for $i < j$. Hence it allows us to write the following:

```
a : Set              -- always allowed
a = Nat
b : Set₁             -- only with cumulativity
b = Nat
c : {i : Level} → Set i    -- only with cumulativity
c = Nat
```

In our thesis we have not used this extension.

25.5. BHK interpretation of first order logic

We extend the interpretation that we gave before to include the universal and existential quantifiers ([4]):

1. A proof of $\forall x.P(x)$ is a function that given an arbitrary element c in the domain, builds a proof that c satisfies P.
2. A proof of $\exists x.P(x)$ is a witness c in the domain and a proof that c satisfies P.

Before diving into quantifiers we first discuss how to represent relations in Agda. Recall the equality relation for natural numbers `_≡_` that we defined before. Its type is `Nat → Nat → Set`. Let us say that we want to define a generic type `REL` for relations on any type. A first attempt could be:

```
REL : Set → Set → Set₁
REL A B = A → B → Set
```

This first definition is somewhat limited. Recall that `Set = Set 0`, thus we restrict `A` and `B` to be small types, furthermore, we require the relation to be a small type as well. If we make our `REL` definition universe polymorphic it turns out like this.

```
REL : {a b : Level} → Set a → Set b → (ℓ : Level) → Set (a ⊔ b ⊔ lsuc ℓ)
REL A B ℓ = A → B → Set ℓ
```

This is the definition used in the Agda standard library ([10]) and is the one that we use in our thesis.

For homogeneous relations we use the name `Rel`:

```
Rel : {a : Level} → Set a → (ℓ : Level) → Set (a ⊔ lsuc ℓ)
Rel A ℓ = REL A A ℓ
```

By using these new definitions we could have defined the type of `_≡_` thus (observe that `Rel Nat lzero` normalizes to `Nat → Nat → Set`):

```
data _≡_ : Rel Nat lzero where
  ... -- same as before
```

In a similar way we can define predicates:

```
Pred : {a : Level} → Set a → (ℓ : Level) → Set (a ⊔ lsuc ℓ)
Pred A ℓ = A → Set ℓ
```

We proceed by giving a representation of the universal quantifier. In the following definition the parameter `D` represents the domain and `P` the predicate.

```
data ∀[_] {a ℓ : Level} (D : Set a) (P : Pred D ℓ) : Set (a ⊔ ℓ) where
  proof∀ : ((e : D) → P e) → ∀[ D ] P
```

In fact this datatype is just a wrapper for a function of type `(e : D) → P e`.

For instance, let us prove that every successor of a natural number is different than zero:

```
aux : (n : Nat) → ¬ (suc n ≡ zero)
aux n ()

s≠z : ∀[ Nat ] (λ n → ¬ (suc n ≡ zero))
s≠z = proof∀ aux
```

Alternatively we could have written a shorter version that does not use an auxiliary lemma.

```
s≠z : ∀[ Nat ] (λ n → ¬ (suc n ≡ zero))
s≠z = proof∀ λ {n ()}
```

Proving ∀-elimination (if $\forall x.P(x)$ and c is in the domain then $P(c)$) is straightforward:

```
∀-elim : {a ℓ : Level} {A : Set a} {P : Pred A ℓ} → ∀[ A ] P → (a : A) → P a
∀-elim (proof∀ f) a = f a
```

We now continue with the existential quantifier. Recall that according to the BHK interpretation a proof $\exists x.P(x)$ is an element c of the domain and a proof that $P(c)$. The first plan could be to use the pair type we defined before to contain the needed elements. We repeat the definition here:

```
data _×_ (A B : Set) : Set where
  _,_ : A → B → A × B
```

The problem is that the type of the second component, `B`, is independent of the first component and thus we cannot express what we need. Here is where the Σ type (or dependent pair) comes into play. A dependent pair is a structure where the type of the second component depends on the value of the first component. This concept is defined by the following datatype.

```
data Σ {ℓ ℓ' : Level} (A : Set ℓ) (B : A → Set ℓ') : Set (ℓ ⊔ ℓ') where
  _,_ : (a : A) → B a → Σ A B
```

Proving ∃-introduction is trivial:

```
∃-intro : {ℓ ℓ' : Level} {A : Set ℓ} {P : A → Set ℓ'}
  → (a : A) → P a → Σ A P
∃-intro a p = a , p
```

We now show that $\forall x(P(x)) \Rightarrow \neg\exists x(\neg P(x))$.

```
p1 : {ℓ ℓ' : Level} {A : Set ℓ} {P : A → Set ℓ'}
  → ∀[ A ] P → ¬ (Σ A (λ x → ¬ (P x)))
p1 (proof∀ f) (c , b) = b (f c)
```

Our goal is to give a term of type ⊥. We have that `f` has type `(a : A) → P a` so `f c` has type `P c`, then `b` has type `¬ (P c)` which is the same as `P c → ⊥`, thus by applying `b` to `(f c)` we get a term of type ⊥.

Of course, as we are in an intuitionistic logic we cannot show the other direction, namely $\neg\exists x(\neg P(x)) \Rightarrow \forall x(P(x))$.

A note on syntax. The reader may find the Σ syntax a bit too different from the usual existential notation: $\exists x(Px)$. We can fix that thanks to Agda's syntax versatility. Agda provides a tool to define custom syntax. Below we show how we can use that tool to improve the syntax of Σ pairs. We will not go into more detail since this feature is of shallow mathematical interest.

```
Σ-syntax : {ℓ ℓ' : Level} → (A : Set ℓ) → (A → Set ℓ') → Set (ℓ ⊔ ℓ')
Σ-syntax = Σ
syntax Σ-syntax A (λ x → B) = Σ[ x ∈ A ] B
```

With this syntax enhancement we can replace `Σ A (λ c → P c)` by `Σ[ c ∈ A ] (P c)`. The previous theorem becomes:

```
∃-intro : {ℓ ℓ' : Level} {A : Set ℓ} {P : A → Set ℓ'}
  → (a : A) → P a → Σ[ c ∈ A ] (P c)
```

There is a variation of this notation which omits the type of the variable as it can be inferred in many cases. That is, instead of `Σ[ c ∈ A ] (P c)` we would have `∃[ c ] (P c)`. We prefer this notation when possible in the thesis[7]. See again the `∃-intro` theorem type using this notation:

```
∃-intro : {ℓ ℓ' : Level} {A : Set ℓ} {P : A → Set ℓ'}
  → (a : A) → P a → ∃[ c ] (P c)
```

25.6. Equality

In Sections 25.2 and 25.3 we have seen how we can define equality for Booleans and naturals. In this section we explore a generic equality and some of its properties and limitations. Of course, the main advantage of generic equality is that we can use it for every type and thus there is no need to redefine it for every new datatype that we define.

Below we present a slight simplification of generic equality as defined in [10]. The semantics of this generic equality type defined in this section are analogous to the identity set for the system LF presented in Section 23.4.9.

```
data _≡_ {a : Level} {A : Set a} (x : A) : A → Set a where
  refl : x ≡ x
```

It may help the reader to read the following description of the generic equality datatype defined above taken from [42]:

> For any type `A` and for any `x` of type `A`, the constructor `refl` provides evidence that `x ≡ x`. Hence, every value is equal to itself, and we have no other way of showing values equal. The definition features an asymmetry, in that the first argument to `_≡_` is given by the parameter `x : A`, while the second is given by an index in `A → Set a`. The first argument to `_≡_` can be a parameter because it does not vary, while the second must be an index, so it can be required to be equal to the first.

[7]For instance, in the definition of Veltman semantics in Chapter 28.

It is easy to see that equality is a reflexive relation.

```
reflexive : {ℓ : Level} {A : Set ℓ} {a : A} → a ≡ a
reflexive = refl
```

We can also show that it is symmetric.

```
symmetric : {ℓ : Level} {A : Set ℓ} {a b : A} → a ≡ b → b ≡ a
symmetric a≡b = ?
```

The argument `a≡b` has type `a ≡ b` and our goal is to give a term of type `b ≡ a`. However, the only way to give a term of type `b ≡ a` is by unifying `a` and `b` since we only have the `refl` constructor. We achieve that by pattern matching against the argument `a≡b`. Then the goal becomes `a ≡ a` and we can use the `refl` constructor.

```
symmetric : {ℓ : Level} {A : Set ℓ} {a b : A} → a ≡ b → b ≡ a
symmetric refl = refl
```

We can prove transitivity in an analogous way.

```
transitivity : {ℓ : Level} {A : Set ℓ} {a b c : A} → a ≡ b → b ≡ c → a ≡ c
transitivity refl refl = refl
```

We can also show that if `x ≡ y` then for any `f` we have `f x ≡ f y`.

```
cong : {ℓA ℓB : Level} {A : Set ℓA} {B : Set ℓB} {x y : A}
  → (f : A → B) → x ≡ y → f x ≡ f y
cong f refl = refl
```

We now see that this new equality datatype is equivalent to the previously defined equality for naturals. To avoid a name clash, we redefine equality for naturals with the `_ℕ≡_` symbol. We also rename reflexivity for `_ℕ≡_` as `ℕrefl`.

```
data _ℕ≡_ : Nat → Nat → Set where
  z≡z : zero ℕ≡ zero
  s≡s : {a b : Nat} → a ℕ≡ b → suc a ℕ≡ suc b

ℕrefl : (n : Nat) → n ℕ≡ n
ℕrefl zero = z≡z
ℕrefl (suc n) = s≡s (ℕrefl n)
```

We show that the new equality implies the old equality.

```
≡→ℕ≡ : {a b : Nat} → a ≡ b → a ℕ≡ b
≡→ℕ≡ {a} refl = ℕrefl a
```

We see that the old equality implies the new equality.

```
ℕ≡→≡ : {a b : Nat} → a ℕ≡ b → a ≡ b
ℕ≡→≡ z≡z = refl
ℕ≡→≡ (s≡s aℕ≡b) = cong suc (ℕ≡→≡ aℕ≡b)
```

See Section 25.8 for a note on extensionality.

25.7. Predicates as mathematical sets

In this section when we say *set* we refer to a subset of an Agda type. The most natural way to represent subsets in Agda is to use predicates. See 25.5 for an introduction. A predicate represents the characteristic function of the associated subset. For instance consider the predicate:

```
Pos : Pred Nat lzero
Pos n = ¬ (n ≡ 0)
```

It represents the subset of strictly positive natural numbers.

We proceed by defining the usual concepts related to mathematical sets. In order to make the types less verbose we assume that we already have `A : Set ℓ` in scope.

1. $\boxed{\in}$ A proof of membership is a simple function application.

   ```
   _∈_ : REL A (Pred A)
   a ∈ X = X a
   ```

   This definition is mostly superfluous but it helps to have a syntax closer to regular mathematics.

2. $\boxed{\notin}$ A proof of non membership is function from a proof of membership to ⊥.

   ```
   _∉_ : REL A (Pred A)
   a ∉ X = ¬ (a ∈ X)
   ```

3. $\boxed{\subseteq}$ A proof of inclusion `X ⊆ Y` is a function that maps a proof of membership to `X` to a proof of membership to `Y`.

   ```
   _⊆_ : Rel (Pred A)
   X ⊆ Y = ∀ {x} → x ∈ X → x ∈ Y
   ```

4. $\boxed{\cap}$ We use pairs to represent the intersection. Each component is a proof of membership to `X` and `Y` respectively.

   ```
   _∩_ : Pred A → Pred A → Pred A
   X ∩ Y = λ x → x ∈ X × x ∈ Y
   ```

5. $\boxed{\cup}$ We use a sum type to represent the union.

   ```
   _∪_ : Pred A → Pred A → Pred A
   X ∪ Y = λ x → x ∈ X ⊎ x ∈ Y
   ```

6. $\boxed{\emptyset}$ The empty set is represented by a characteristic constant function to ⊥.

   ```
   ∅ : Pred A
   ∅ = λ x → ⊥
   ```

7. $\boxed{\mathbf{1}}$ Similarly, the universe set is represented by a characteristic constant function to ⊤.

   ```
   U : Pred A
   U = λ x → ⊤
   ```

8. $\boxed{\{x\}}$ A singleton set is defined using equality (assuming we have equality defined for that type).

   ```
   {_} : A → Pred A
   { x } = λ y → x ≡ y
   ```

25.8. Extensionality

In Set theory we call axiom of extensionality the property that if two sets have the same elements, then they are equal. As in Agda we represent sets as functions we reword extensionality for functions: if two functions have the same domain and coincide for every element in their domain then they are equal. In symbols:

$$\forall f \forall g.(\forall x.f(x) = g(x)) \Rightarrow f = g.$$

In Agda we can represent this concept thus ([10]):

```
Extensionality : (a b : Level) → Set _
Extensionality a b =
  {A : Set a} {B : A → Set b} {f g : (x : A) → B x} →
  ((x : A) → f x ≡ g x) → f ≡ g
```

It is usually the case that we accept the axiom of extensionality as part of our metalogic as it is part of the most popular logical framework ZF, however, in Agda the property of extensionality is not an axiom nor a provable theorem.

An direct consequence of the lack of extensionality is that we cannot show equality of sets by double inclusion.

```
⊆⊇→≡ : {ℓS ℓA : Level} {A : Set ℓA} {X Y : Pred A ℓS} → X ⊆ Y → Y ⊆ X → X ≡ Y
⊆⊇→≡ = ? -- not provable
```

We will see that this has a small effect on the definition of generalized Veltman frames in Chapter 29.

25.9. Positivity

In this section we present a technicality as described in the Agda documentation ([3]) regarding datatype definitions that will become relevant in Chapter 28 where we define ordinary Veltman semantics.

When defining a datatype `D`, Agda poses an additional requirement on the types of the constructors of `D`, namely that `D` may only occur strictly positively in the types of their arguments. Concretely, for a datatype with constructors $\mathtt{c_1 : A_1}$, …, $\mathtt{c_n : A_n}$, Agda checks that each $\mathtt{A_i}$ has the form

```
(y₁ : B₁) → ... → (yₗ : Bₗ) → D
```

where an argument of type $\mathtt{B_i}$ of the constructors does not mention `D` or has the form

```
(z₁ : C₁) → ... → (zₖ : Cₖ) → D
```

The following example showcases the possibility to build a term of type ⊥ by defining a non strictly positive type `Bad`. As mentioned above, Agda rejects the definition of `Bad`.

```
data ⊥ : Set where

data Bad : Set where
  bad : (Bad → ⊥) → Bad

self-app : Bad → ⊥
self-app (bad f) = f (bad f)

absurd : ⊥
absurd = self-app (bad self-app)
```

Part V.

Agda in the thesis

The goal of this part is to guide the reader through some key parts of the code that we have implemented. It is worth noting that we have started from scratch as we believe that no other previous work in interpretability logics has been done in Agda.

In Chapter 26 we give the inductive definition of modal formulas in Agda. We also show how we can define the non-primitive operators.

In Chapter 27 we show how we have define Noetherian relations in Agda. We introduce the concepts of coinductive type and proof by coinduction in Agda.

In Chapter 28 we describe how we defined ordinary frames. Moreover, we explain how we implemented the forcing relation, as it is non-trivial to implement in Agda due to the restriction of positivity on datatype definitions.

In Chapter 28 we comment how we adapted the definitions for generalized Veltman semantics. Furthermore, we include a guided Agda proof which shows that Löb's axiom is forced in every world and model.

In Chapter 30 we give an Agda implementation of the logic IL. We also show how we can use to prove some theorems of IL.

In Chapter 30 we present a language, that we implemented, for writing verified Hilbert style proofs in Agda. All of that, with paper-like syntax.

The implementation relies on Agda 2.6.1 and the Agda standard library ([10]).

26. Modal formulas

Here we present the Agda type that represents a formula as defined in Chapter 4.
First we define variables to be natural numbers:

```
Var : Set
Var = Nat
```

We proceed by inductively defining the formula type: `Fm`. We add a constructor for variables and one for each primitive operator.

```
data Fm : Set where
  var : Var → Fm
  ⊥' : Fm
  _↝_ : Fm → Fm → Fm
  _▷_ : Fm → Fm → Fm
```

We have named the bottom constructor `⊥'` since the symbol ⊥ is commonly used in Agda as the empty type. We have used the ↝ to denote a implication since → is a reserved symbol for the Agda function type.
We finally add definable operators as Agda functions. For instance, we define ¬ and □ thus:

```
infix 60 ¬'_
¬'_ : Fm → Fm
¬' a = a ↝ ⊥'

infix 70 □_
□_ : Fm → Fm
□_ a = ¬' a ▷ ⊥'
```

We use the symbol `¬'` instead of ¬ for the same reason we used `⊥'` instead of ⊥.
It is often the case that we define priority and associativity for our infix operators in order to minimize the amount of needed parentheses. The following code defines the *infixity* (level or priority) of _↝_ and _▷_.

```
infixr 20 _↝_
infixr 50 _▷_
```

A greater number means higher priority. Then we can drop the parentheses from the previous formula `var 1 ▷ var 0 ↝ ⊥'`. The *r* in `infixr` stands for right associativity.

27. Noetherian relations

We say that a relation is Noetherian if it is conversely well-founded. We begin by formalizing the concept of infinite ascending chain in Agda. In order to do that, we define a coinductive record datatype ([3, 29]). A coinductive record is allowed to be infinite. In other words, it does not need to have a non-recursive constructor.

```
record InfiniteChain {ℓW ℓR} {W : Set ℓW} (_<_ : Rel W ℓR) (a : W)
  : Set (ℓR ⊔ ℓW)where
  coinductive
  constructor infiniteChain
  field
    b : W
    a<b : a < b
    tail : InfiniteChain _<_ b
```

We see that the previous record datatype represents an infinite ascending chain starting at `a` of some relation `_<_`. It has three fields. `b`: The next element in the chain. `a<b`: A proof that $a < b$ and `tail`: an infinite chain starting at `b`.

Then we can define being Noetherian as the negation of the existence of an infinite chain:

```
Noetherian : ∀ {ℓR ℓW} {W : Set ℓW} → Rel W ℓR → Set (ℓR ⊔ ℓW)
Noetherian _<_ = ∀ {a} → ¬ (InfiniteChain _<_ a)
```

For instance, we can prove that a Noetherian relation is irreflexive. First we show that from a proof that xRx we can build an infinite chain:

```
infiniteRefl : ∀ {ℓ} {R : Rel A ℓ} {x} → R x x → InfiniteChain R x
InfiniteChain.b (infiniteRefl {x = x} Rxx) = x
InfiniteChain.a<b (infiniteRefl {x = x} Rxx) = Rxx
InfiniteChain.tail (infiniteRefl {x = x} Rxx) = infiniteRefl Rxx
```

We see that each equation corresponds to a different field in the record datatype. This construction is known as *copattern*. Coinductive datatypes must be constructed in this way. Copatterns are for coinductive types what patterns are for inductive (finite) types. In [2] copatterns are described in detail.

And then we can apply the Noetherian definition.

```
Noetherian⇒Irreflexive : ∀ {ℓR ℓW} {W : Set ℓW} {R : Rel W ℓR}
     → Noetherian R → ∀ {x} → ¬ R x x
Noetherian⇒Irreflexive noetherian Rxx = noetherian (infiniteRefl Rxx)
```

To see another example we refer the reader to Section 29.1.

28. Ordinary Veltman semantics

In this chapter we explain how we have represented ordinary Veltman semantics in Agda.

To represent ordinary Veltman semantics in Agda, the first step is to define the type of an ordinary Veltman frame:

```
record Frame {ℓW ℓR ℓS : Level} (W : Set ℓW) (R : Rel W ℓR) (S : Rel₃ W ℓS)
  : Set (ℓW ⊔ ℓR ⊔ ℓS) where
  constructor frame
  field
    witness : W
    R-trans : Transitive R
    R-noetherian : Noetherian R
    Sw⊆R[w]² : ∀ {w u v} → S w u v → R w u × R w v
    Sw-refl : ∀ {w u} → R w u → S w u u
    Sw-trans : ∀ {w} → Transitive (S w)
    R-Sw-trans : ∀ {w u v} → R w u → R u v → S w u v
```

The keyword `record` is used to define a new product type (a tuple) in which each component (or field) has a name that we can use to access it.

We see that the datatype is parameterized by the universe `W`, the `R` relation, the `S` relation and their respective universe levels `ℓW, ℓR, ℓS`.

The first component, `witness`, is required to make sure that the set of worlds is not empty. The remaining components are the properties that must be satisfied according to Definition 6.1.

We define a valuation on a frame thus:

```
Valuation : Frame {ℓW} {ℓR} {ℓS} W R S → Set (lsuc lzero ⊔ ℓW)
Valuation {W = W} F = REL W Var lzero
```

And then we define a model to be a frame parameterized with a valuation on that frame.

```
record Model (W : Set ℓW) (R : Rel W ℓR) (S : Rel₃ W ℓS) (V : REL W Var lzero)
  : Set (ℓW ⊔ ℓR ⊔ ℓS) where
  constructor model
  field
    F : Frame {ℓW} {ℓR} {ℓS} W R S
```

Our next step is to define the forcing relation.

```
data _,_⊩_ (M : Model {ℓW} {ℓR} {ℓS} W R S V) (w : W)
  : Fm → Set (ℓW ⊔ ℓR ⊔ ℓS)
```

We set a model and a world of that model as parameters as they should be shared by all constructors. We leave the formula as an index as it may vary depending on the constructor. We should introduce a constructor for each case in Definition 6.3:

1. We do not need a constructor for `⊥'` as its absence implicitly implies that we can never build an instance of `M , w ⊩ ⊥'` regardless of `M` and `w`.

2. If $x \in \mathsf{Var}$, then $w \Vdash x$ iff $\langle w, x \rangle \in V$:

```
var : {p : Var} → p ∈ V w → M , w ⊩ var p
```

3. If $A, B \in \mathsf{Fm}$, then $w \Vdash A \rightarrow B$ iff if $w \Vdash A$ then $w \Vdash B$:

   ```
   impl : {A B : Fm} → ((M , w ⊩ A) → (M , w ⊩ B)) → M , w ⊩ (A ⇝ B)
   ```

4. If $A, B \in \mathsf{Fm}$, then $w \Vdash A \rhd B$ iff if wRu and $u \Vdash A$ then there exists v such that $v \Vdash B$ and $uS_w v$.

   ```
   rhd : {A B : Fm} →
     ({u : W} → R w u → M , u ⊩ A → (∃[ v ] (S w u v × (M , v ⊩ B))))
     → M , w ⊩ A ▷ B
   ```

Unfortunately the definition above is not accepted by Agda. The reason is that constructors `rhd` and `impl` both fail the positivity check (see Section 25.9). For instance, observe that in the `impl` constructor type we have `(M , w ⊩ A)` on the left of an arrow `→`.

We have circumvented this problem by providing mutually recursive definitions for *forcing* (`_,_⊩_`) and *not forcing* (`_,_⊮_`). Agda allows the definition of mutually recursive datatypes (and functions) by first providing the type of both[1] definitions and after those giving the rest of the definition, that is, the constructors for datatypes and the equations for functions.

The type of the two datatypes that we want to define are as follows.

```
data _,_⊩_ (M : Model {ℓW} {ℓR} {ℓS} W R S V) (w : W)
  : Fm → Set (ℓW ⊔ ℓR ⊔ ℓS) -- forcing relation

data _,_⊮_ (M : Model {ℓW} {ℓR} {ℓS} W R S V) (w : W)
  : Fm → Set (ℓW ⊔ ℓR ⊔ ℓS) -- not forcing relation
```

Next we provide the strictly positive types of each constructor of the `_,_⊩_` and `_,_⊮_` relations.

1. For the `⊥'` constant.
   a) Forcing (`_,_⊩_`). No constructor is required.
   b) Not forcing (`_,_⊮_`).

      ```
      bot : M , w ⊮ ⊥'
      ```

2. For variables.
   a) Forcing (`_,_⊩_`).

      ```
      var : {p : Var} → p ∈ V w → M , w ⊩ var p
      ```

   b) Not forcing (`_,_⊮_`).

      ```
      var : {p : Var} → p ∉ V w → M , w ⊮ var p
      ```

3. For implication (`⇝`).
   a) Forcing (`_,_⊩_`).

      ```
      impl : {A B : Fm} → M , w ⊮ A ⊎ M , w ⊩ B → M , w ⊩ A ⇝ B
      ```

   b) Not forcing (`_,_⊮_`).

      ```
      impl : {A B : Fm} → M , w ⊩ A → M , w ⊮ B → M , w ⊮ A ⇝ B
      ```

[1] Or more if it is the case.

4. For interpretability (▷).
   a) Forcing (_,_⊩_).
      ```
      rhd : {A B : Fm} →
        (∀ {u} → R w u → M , u ⊮ A ⊎ (∃[ v ] (S w u v × M , v ⊩ B)))
        → M , w ⊩ A ▷ B
      ```
   b) Not forcing (_,_⊮_).
      ```
      rhd : {A B : Fm} →
       ∃[ u ] (R w u × M , u ⊩ A × ((v : W) → (¬ S w u v) ⊎ M , v ⊮ B))
        → M , w ⊮ A ▷ B
      ```

Putting it all together results in the following definitions:

```
data _,_⊩_ M w where
  var : {x : Var} → V w x → M , w ⊩ var x
  impl : {A B : Fm} → M , w ⊮ A ⊎ M , w ⊩ B → M , w ⊩ A ↝ B
  rhd : {A B : Fm} →
    (∀ {u} → R w u → M , u ⊮ A ⊎ (∃[ v ] (S w u v × M , v ⊩ B)))
    → M , w ⊩ A ▷ B

data _,_⊮_ M w where
  var : {x : Var} → ¬ (V w x) → M , w ⊮ var a
  impl : {A B : Fm} → M , w ⊩ A → M , w ⊮ B → M , w ⊮ A ↝ B
  rhd : {A B : Fm} →
    ∃[ u ] (R w u × M , u ⊩ A × ((v : W) → (¬ S w u v) ⊎ M , v ⊮ B))
    → M , w ⊮ A ▷ B
  bot : M , w ⊮ ⊥'
```

To prove that `_,_⊩` and `_,_⊮` are indeed the negation of each other we should prove two lemmas. We define `A ⇔ B ≔ A → B × B → A`. Then the lemma in Agda types is as follows.

Lemma 28.1.

1. `∀ {M w A} → M , w ⊩ A ⇔ ¬ (M , w ⊮ A)`.
2. `∀ {M w A} → ¬ (M , w ⊩ A) ⇔ M , w ⊮ A`.

For part 1 we can prove ⇒ and for part 2 we can prove ⇐ (see Lemma 28.3). However, it is not possible to prove the remaining directions. In general terms, this is due to the fact that in Agda (and in intuitionistic logic in general) we can prove that `(¬ A ⊎ B) → A → B` but we cannot prove `A → B → (¬ A ⊎ B)`. The reason being that we lack the law of excluded middle, as it is a non-constructive axiom. In order to prove the remaining directions we need to assume that the forcing relation is decidable.

Definition 28.2.
We say that `M` is decidable model if for any world `w` and formula `A` we have that either `M , w ⊩ A` or `M , w ⊮ A`.
In Agda terms:

```
DecidableModel : Model → Set
DecidableModel M = ∀ w A → M , w ⊩ A ⊎ M , w ⊮ A
```

Proof. Under the assumption that we restrict ourselves to decidable models we can prove Lemma 28.1. □

Lemma 28.3. The following properties on the forcing relation are true:

1. `⊩⊥ : ∀ {M w} → ¬ (M , w ⊩ ⊥');`
2. `⊮→¬⊩ : ∀ {M w A} → M , w ⊮ A → ¬ (M , w ⊩ A);`
3. `⊩→¬⊮ : ∀ {M w A} → M , w ⊩ A → ¬ (M , w ⊮ A);`
4. `⊩MP : ∀ {M w A B} → M , w ⊩ A ↝ B → M , w ⊩ A → M , w ⊩ B;`
5. `⊩¬ : ∀ {M w A} → (M , w ⊩ ¬' A) ⇔ (M , w ⊮ A);`
6. `⊮¬ : ∀ {M w A} → M , w ⊮ ¬' A ⇔ M , w ⊩ A;`
7. `⊩¬¬ : ∀ {M w A} → M , w ⊩ ¬' ¬' A ⇔ M , w ⊩ A;`
8. `⊮¬¬ : ∀ {M w A} → M , w ⊮ ¬' ¬' A ⇔ M , w ⊮ A;`
9. `⊩∧ : ∀ {M w A B} → M , w ⊩ A ∧ B ⇔ (M , w ⊩ A × M , w ⊩ B);`
10. `⊮∧ : ∀ {M w A B} → M , w ⊮ A ∧ B ⇔ (M , w ⊮ A ⊎ M , w ⊮ B);`
11. `⊩∨ : ∀ {M w A B} → M , w ⊩ A ∨ B ⇔ (M , w ⊩ A ⊎ M , w ⊩ B);`
12. `⊩□ : ∀ {M w A} → M , w ⊩ □ A ⇔ (∀ {v} → R w v → M , v ⊩ A);`
13. `⊮□ : ∀ {M w A} → M , w ⊮ □ A ⇔ (∃[ u ] (R w u × M , u ⊮ A));`
14. `⊩◇ : ∀ {M w A} → M , w ⊩ ◇ A ⇔ (∃[ u ] (R w u × M , u ⊩ A));`
15. `⊮◇ : ∀ {M w A} → M , w ⊮ ◇ A ⇔ (∀ {u} → R w u → M , u ⊮ A);`
16. `⊩↝⇒ : ∀ {M w A B} → M , w ⊩ A ↝ B → M , w ⊩ A → M , w ⊩ B;`
17. `⊩▷⇒ : ∀ {M w A B} → M , w ⊩ A ▷ B → (∀ {u} → R w u → M , u ⊩ A → ∃[ v ] (S w u v × M , v ⊩ B).`

Proof. The above properties have been proven in Agda and Coq without assuming that the model is decidable. □

Lemma 28.4. The following series of equivalences can be proven for decidable models.

1. `⊩↝ : ∀ {w A B} → M , w ⊩ A ↝ B ⇔ (M , w ⊩ A → M , w ⊩ B);`
2. `⊩▷ : ∀ {w A B} → M , w ⊩ A ▷ B ⇔ (∀ {u} → R w u → M , u ⊩ A → ∃[ v ] (S w u v × M , v ⊩ B));`
3. `⊩⇔¬⊮ : ∀ {w A} → M , w ⊩ A ⇔ (¬ M , w ⊮ A);`
4. `⊮⇔¬⊩ : ∀ {w A} → M , w ⊮ A ⇔ (¬ M , w ⊩ A).`

Proof. Note that we only need the decidability assumption for 1 (⇐), 2 (⇐), 3 (⇐) and 4 (⇐). □

From now on, we always restrict ourselves to decidable models as the usage of Lemma 28.4 is ubiquitous. If we were to assume that we are outside of Agda and that we accept the law of excluded middle as part of our metalogic, the mentioned assumption could be dropped.

29. Generalized Veltman semantics

In this chapter we explain how we have represented generalized Veltman semantics in Agda. As explained in Chapter 7 we consider eight different quasi-transitivity properties, thus, we need to define everything related to generalized Veltman semantics to be generic with respect to the quasi-transitivity condition used, which was certainly presented some challenges.

Analogously to ordinary semantics we start by defining a frame. We begin by defining a datatype that represents a frame without the quasi-transitivity condition. See that we define the type 𝕎 ≔ `Pred W ℓW`, which means that a term of type 𝕎 is a subset of `W` (see Section 25.7 for details on how to represent mathematical sets in Agda). See that the `S-ext` field adds extensionality restricted to the third component of the `S` relation.

```
record FrameNoTrans (W : Set ℓW) (R : Rel W ℓR) (S : REL₃ W W (Pred W ℓW) ℓS)
  : Set (lsuc lzero ⊔ ℓR ⊔ ℓS ⊔ lsuc ℓW) where
  constructor frame
  𝕎 : Set (lsuc ℓW)
  𝕎 = Pred W ℓW
  field
    witness : W
    Swu-sat : ∀ {w u Y} → S w u Y → Satisfiable Y
    R-trans : Transitive R
    R-noetherian : Noetherian R
    Sw⊆R[w] : ∀ {w u Y} → S w u Y → R w u
    SwuY⊆Rw : ∀ {w u Y} → S w u Y → ∀ {y} → y ∈ Y → R w y
    S-quasirefl : ∀ {w u} → R w u → S w u { u }
    R-Sw-trans : ∀ {w u v} → R w u → R u v → S w u { v }
    S-ext : ∀ {w x V V'} → S w x V → V ⊆ V' → V' ⊆ V → S w x V'
```

Then we define a new datatype `Frame` which represents a non-transitive Generalized Veltman frame plus some quasi-transitivity condition, which is left as a parameter `T`.

```
record Frame (W : Set ℓW) (R : Rel W ℓR) (S : REL₃ W W (Pred W ℓW) ℓS)
  (T : (W : Set ℓW) → REL₃ W W (Pred W ℓW) ℓS → Set (lsuc ℓW ⊔ ℓS))
  : Set (lsuc ℓW ⊔ ℓR ⊔ ℓS) where
  constructor frame
  field
    frame-0 : FrameNoTrans {ℓW} {ℓR} {ℓS} W R S
    quasitrans : T W S
```

We now define all the quasi-transitivity conditions. Here we only present Condition 4.

```
Trans-4 : (W : Set ℓW) → REL₃ W W (Pred W ℓW) ℓS → Set (lsuc ℓW ⊔ ℓS))
Trans-4 W S = ∀ {x u Y} → S x u Y → ∃[ y ] (y ∈ Y × (∀ {Y'} → S x y Y' → S x u Y'))
```

And finally we can define a datatype that represents a generalized Veltman frame for each of the quasi-transitivity conditions as a simple instantiation of the generic `Frame` defined before. Here we only present Condition 4.

```
Frame4 : (W : Set ℓW) (R : Rel W ℓR) (S : REL₃ W W (Pred W ℓW) ℓS) → Set _
Frame4 W R S = Frame W R S (Trans-4 W S)
```

In order to define the generalized Veltman semantics forcing relation, since we need to define it generically to work for any quasi-transitivity condition assume that we have some term `T` representing such condition of typed thus:

```
(T : ∀ {ℓW ℓS} (W : Set ℓW) → REL₃ W W (Pred W ℓW) ℓS → Set (lsuc ℓW ⊔ ℓS))
```

Then we define a generalized model in an analogous way to how we did it for ordinary semantics:

```
record Model
  {ℓW ℓR ℓS}
  (W : Set ℓW)
  (R : Rel W ℓR)
  (S : REL₃ _ _ _ ℓS)
  (V : REL W Var lzero)
  : Set (lsuc ℓW ⊔ ℓR ⊔ ℓS) where
  constructor model
  field
    F : Frame {ℓW} {ℓR} {ℓS} W R S T
```

And finally we define the forcing relation using mutually recursive datatypes as we did for ordinary semantics. The only difference is in the `rhd` constructor.

```
data _,_⊮_ {ℓW ℓR ℓS W R S V} (M : Model {ℓW} {ℓR} {ℓS} W R S V) (w : W)
  : Fm → Set (lsuc ℓW ⊔ ℓR ⊔ ℓS)

data _,_⊩_ {ℓW ℓR ℓS W R S V} (M : Model {ℓW} {ℓR} {ℓS} W R S V) (w : W)
  : Fm → Set (lsuc ℓW ⊔ ℓR ⊔ ℓS)

data _,_⊩_ {ℓW} {ℓR} {ℓS} {W} {R} {S} {V} M w where
  var : ∀ {a : Var} → a ∈ V w → M , w ⊩ var a
  impl : ∀ {A B} → M , w ⊮ A ⊎ M , w ⊩ B → M , w ⊩ A ⇝ B
  rhd : ∀ {A B} →
    (∀ {u} → R w u → M , u ⊮ A ⊎ (∃[ Y ] (S w u Y × (Y ⊆ M ,_⊩ B))))
    → M , w ⊩ A ▷ B

data _,_⊮_ {ℓW} {ℓR} {ℓS} {W} {R} {S} {V} M w where
  var : ∀ {a : Var} → a ∉ V w → M , w ⊮ var a
  impl : ∀ {A B} → M , w ⊩ A → M , w ⊮ B → M , w ⊮ A ⇝ B
  rhd : ∀ {A B} →
    ∃[ u ] (R w u × M , u ⊩ A
   × ∀ Y → Satisfiable Y → (¬ S w u Y) ⊎ (Satisfiable (Y ∩ (M ,_⊮ B))))
    → M , w ⊮ A ▷ B
  bot : M , w ⊮ ⊥'
```

Recall that in Chapter 28 in order to prove some properties we had to assume that the ordinary models were decidable in the sense of Definition 28.2. For generalized semantics we need to make a stronger assumption described in the next definition.

Definition 29.1. We say that `M` is multi-decidable model if for any set of worlds `Y` and formula `A` we can decide whether

1. for every element `y` in `Y` we have `M , y ⊮ A`; or
2. there is an element `y` in `Y` such that `M , y ⊩ A`.

In Agda terms:

```
MultiDecidableModel : ∀ {ℓW ℓR ℓS W R S V} → Model {ℓW} {ℓR} {ℓS} W R S V
  → Set (lsuc ℓW ⊔ ℓR ⊔ ℓS ⊔ lsuc ℓW)
MultiDecidableModel {ℓW = ℓW} {W = W} M =
  ∀ (Y : Pred W ℓW) A → Y ⊆ M ,_⊩ A ⊎ Satisfiable (Y ∩ (M ,_⊮ A))
```

Lemma 29.2. Every multi-decidable model is also decidable.

Lemma 29.3. Assuming that we restrict ourselves to multi-decidable models then properties in Lemmas 28.1, 28.3 and 28.4 also hold for generalized semantics.

29.1. A guided Agda proof

In this section we guide the user through a non-trivial Agda proof, which will hopefully give the reader a feel of how we can proof generalized Veltman semantic properties in Agda. We prove that for any generalized Veltman model M and world w we have that w forces Löb's axiom. In symbols:

$$\forall M \forall w \forall A.\ M, w \Vdash \Box(\Box A \to A) \to \Box A.$$

Note that since Löb's axiom is in IL we need to show this as part of the soundness proof.

We begin by outlining the proof without Agda. Assume that for some world w_0 we have $w_0 \Vdash \Box(\Box A \to A)$. Assume for a contradiction that $w \nVdash \Box A$, then there exists some w_1 such that $w_0 R w_1 \nVdash A$. Since $w_0 R w_1$ it follows that $w_1 \Vdash \Box A \to A$. Then since $w_1 \nVdash A$ we necessarily have that $w_1 \nVdash \Box A$. Then there exists w_2 such that $w_1 R w_2 \nVdash A$. Since R is transitive we have that $w_0 R w_2$ and thus $w_2 \Vdash \Box A \to A$. We can repeat the previous argument indefinitely to build an infinite chain $w_0 R w_1 R ...$, which is a contradiction since R is Noetherian. This concludes the pen and paper proof.

We proceed with the Agda proof. During the course of this example we use some lemmas listed below, which we have proved in Agda, however, we just display their type here and we omit their proof in order to save space. Also, assume that we have some model `M` in scope.

```
⊩4' : ∀ {w A} → M , w ⊩ □ A → M , w ⊩ □ □ A
⊮□ : ∀ {w A} → M , w ⊮ □ A ⇔ (∃[ u ] (R w u × M , u ⊮ A))
⊩□ : ∀ {w A} → M , w ⊩ □ A ⇔ (∀ {v} → R w v → M , v ⊩ A)
⊩⇔¬⊮ : ∀ {w A} → M , w ⊩ A ⇔ ¬ (M , w ⊮ A)
_⇒_ : ∀ {a b} {A : Set a} {B : Set b} → A ⇔ B → A → B
```

Naming convention. In this proof we use the popular convention to name variables after their type, which greatly improves the readability of proofs. For instance, if we bind some variable of type `R w u` we will name it `Rwu`; if we bind a variable of type `M , w ⊩ A` we will name it `w⊩A`; and so on.

We begin by showing a useful lemma: for any w, u, A, if wRu and $u \nVdash A$ and $w \Vdash \Box(\Box A \to A)$ then we can build an infinite R-chain starting at w. The following type expresses the aforementioned property.

```
R-chain : ∀ {w u A} → R w u → M , u ⊮ A → M , w ⊩ □ (□ A ↝ A) → InfiniteChain R w
```

Recall that infinite chains are defined as coinductive datatypes in Chapter 27. Hence we proceed by building the infinite chain using copatterns. The first two components are clear:

```
InfiniteChain.b (R-chain {w} {u} Rwu uA uF) = u
InfiniteChain.a<b (R-chain {w} {u} Rwu uA uF) = Rwu
```

Then we must show that there is an infinite chain starting at u. The argument `w⊩□⟨□A↝A⟩` has type `M , w ⊩ □ (□ A ↝ A)`, hence by applying the lemma `⊩□` in the right (⇒) direction and using the fact that `Rwu` has type `R w u` we get a term of type `M , u ⊩ □ A ↝ A` which we pattern match using the `widh` construct[1]. By the definition of the constructor `impl` for the `_,_⊩_` datatype it follows that we have two cases: either `M , u ⊩ A` or `M , u ⊮ □ A`.

If it is the case that `M , u ⊩ A` we can build a term of type $\bot$ by using the `⊩→¬⊮` lemma and then we can use the principle of explosion to return anything.

```
InfiniteChain.tail (R-chain Rwu u⊮A w⊩□⟨□A↝A⟩) with (⊩□ ⇒ w⊩□⟨□A↝A⟩) Rwu
... | impl (inj₂ u⊩A) = explosion (⊩→¬⊮ u⊩A u⊮A)
```

On the contrary, if we have that `M , u ⊮ □ A` then by the `⊮□` lemma we get that there exists some `v` such that `R u v` and `M , v ⊩ A`. Then we can use the `⊩4'` lemma to get a term of type `M , w ⊩ □ (□ (□ A ↝ A))`, then by lemma `⊩□` and the fact we have a proof of `R w u` we can build a term of type `M , u ⊩ □ (□ A ↝ A)`. Finally by a recursive call (induction hypothesis) to `R-chain` with `R u v` and `v⊩A` and the term described above we get a term of the desired type.

```
... | impl (inj₁ x⊮□A) with ⊮□ ⇒ x⊮□A
... | (v , Ruv , v⊩A) = R-chain Ruv v⊩A ((⊩□ ⇒ ⊩4' w⊩□⟨□A↝A⟩) Rwu)
```

This concludes the proof of the lemma. We now proceed to prove our theorem. The statement of the theorem is represented by the following type:

```
⊩L : ∀ {w A} → M , w ⊩ □ (□ A ↝ A) ↝ □ A
```

We use lemma `⊩↝` to get a proof of `M , w ⊩ □ (□ A ↝ A)`. Then we use lemma `⊩□` on the left direction, so we assume `R w u` and our goal is to show `M , u ⊩ A`. By using lemma `⊩⇔¬⊮` on the left direction. Our goal is to show `¬ (M , w ⊮ A)` which normalizes to `M , w ⊮ A → ⊥`, hence we assume `M , u ⊮ A` and we aim to build a proof of $\bot$. We can build an infinite chain with lemma `R-chain` proved above and the facts that `R w u`, `M , u ⊮ A` and `M , w ⊩ □ (□ A ↝ A)`. Before the final step it may be useful to recall the definition of a `Noetherian` relation (see Chapter 27):

```
Noetherian _<_ = ∀ {a} → ¬ (InfiniteChain _<_ a)
```

Finally we use the property that the `R` relation of the model is Noetherian to get a term of type $\bot$ as desired.

```
⊩L : ∀ {w A} → M , w ⊩ □ (□ A ↝ A) ↝ □ A
⊩L {w} {A} = ⊩↝ ⇐ λ w⊩□⟨□A→A⟩ → ⊩□ ⇐ λ {u} Rwu → ⊩⇔¬⊮ ⇐
  λ {u⊮A → R-noetherian (R-chain Rwu u⊮A w⊩□⟨□A→A⟩)}
```

Putting it all together we have:

```
R-chain : ∀ {w u A} → R w u → M , u ⊮ A → M , w ⊩ □ (□ A ↝ A) → InfiniteChain R w
InfiniteChain.b (R-chain {w} {u} Rwu uA uF) = u
InfiniteChain.a<b (R-chain {w} {u} Rwu uA uF) = Rwu
InfiniteChain.tail (R-chain {w} {u} Rwu u⊮A w⊩□⟨□A↝A⟩)
   with (⊩□ ⇒ w⊩□⟨□A↝A⟩) Rwu
... | impl (inj₂ u⊩A) = ⊥-elim (⊩→¬⊮ u⊩A u⊮A)
... | impl (inj₁ x⊮□A) with ⊮□ ⇒ x⊮□A
... | (v , Ruv , v⊩A) = R-chain Ruv v⊩A ((⊩□ ⇒ ⊩4' w⊩□⟨□A↝A⟩) Rwu)

⊩L : ∀ {w A} → M , w ⊩ □ (□ A ↝ A) ↝ □ A
⊩L {w} {A} = ⊩↝ ⇐ λ w⊩□⟨□A→A⟩ → ⊩□ ⇐ λ {u} Rwu → ⊩⇔¬⊮ ⇐
  λ {u⊮A → R-noetherian (R-chain Rwu u⊮A w⊩□⟨□A→A⟩)}
```

[1]The `width` construct allows us to pattern match on terms that can be build from the arguments of the function.

30. Logic IL and syntactic proofs

Here we present our efforts on formalizing syntactic IL proofs in Agda. We restrict ourselves to finite sets of assumptions.

We begin by defining the necessary type to represent a finite list:

```
data List {a : Level} (A : Set a) : Set a where
  []   : List A
  _::_ : A → List A → List A
```

Then we can define a proof of membership inductively in the following way:

```
data _∈_ {a : Level} {A : Set a} (a : A) : Pred (List A) a where
  here  : {x : A} {xs : List A} → a ≡ x → a ∈ (x :: xs)
  there : {x : A} {xs : List A} → a ∈ xs → a ∈ (x :: xs)
```

Now that we have all the necessary tools, we proceed to define the relation _⊢_, which represents the IL logic in Agda.

```
data _⊢_ (Π : List Fm) : Fm → Set where
  -- identity rule
  Ax : ∀ {A} → A ∈ Π → Π ⊢ A
  -- classical axioms
  C1 : ∀ {A B} → Π ⊢ A ↝ (B ↝ A)
  C2 : ∀ {A B C} → Π ⊢ (A ↝ (B ↝ C)) ↝ ((A ↝ B) ↝ (A ↝ C))
  C3 : ∀ {A B} → Π ⊢ (¬' A ↝ ¬' B) ↝ (B ↝ A)
  -- GL axioms
  K : ∀ {A B} → Π ⊢ (□ (A ↝ B)) ↝ (□ A ↝ □ B)
  L : ∀ {A} → Π ⊢ □ (□ A ↝ A) ↝ □ A
  -- IL axioms
  J1 : ∀ {A B} → Π ⊢ □ (A ↝ B) ↝ A ▷ B
  J2 : ∀ {A B C} → Π ⊢ A ▷ B ∧ B ▷ C ↝ A ▷ C
  J3 : ∀ {A B C} → Π ⊢ (A ▷ C ∧ B ▷ C) ↝ (A ∨ B) ▷ C
  J4 : ∀ {A B} → Π ⊢ A ▷ B ↝ ◇ A ↝ ◇ B
  J5 : ∀ {A} → Π ⊢ ◇ A ▷ A
  -- rules
  MP : ∀ {A B} → Π ⊢ A ↝ B → Π ⊢ A → Π ⊢ B
  nec : ∀ {A} → [] ⊢ A → Π ⊢ □ A
```

We include constructor `Ax` so we can use assumptions. We include constructors `C1`, `C2` and `C3` so that every classical tautology in the language of IL can be proved. Then we add the axioms of IL and finally we add `MP` for modus ponens and `nec` for necessitation. Note that the necessitation rule only accepts IL theorems (empty set of assumptions) and thus this definition is fitting for local semantics. We have formalized several results about IL, which are presented in Chapter 5.

Consider the pen and paper syntactic proof of $A \to A$.

0. $(A \to ((A \to A) \to A)) \to ((A \to (A \to A)) \to (A \to A))$	By $C2$
1. $A \to ((A \to A) \to A)$	By $C1$
2. $A \to (A \to A)$	By $C1$
3. $(A \to (A \to A)) \to A \to A$	By MP 0, 1
4. $A \to A$	By MP 3, 2

■

Now see how we could replicate the proof in Agda using our definition of _⊢_.

```
⊢A↝A : ∀ {A Π} → Π ⊢ A ↝ A
⊢A↝A {A} = MP (MP (C2 {Π} {A} {A ↝ A} {A}) (C1 {Π} {A} {A ↝ A})) (C1 {Π} {A} {A})
```

We see that it is extremely verbose and hard to read. We can substantially shorten it by relying on Agda's type inference to automatically infer the instantiation of almost all of the axiom schemas used. However, it is still far from being human friendly.

```
⊢A↝A : ∀ {A Π} → Π ⊢ A ↝ A
⊢A↝A {A} = MP (MP C2 C1) (C1 {B = A})
```

For now, we forget about the obscure syntax and we use the previous result to prove that $A \rhd A$ is an IL theorem. First, the pen and paper proof:

0. $A \to A$	By $\vdash A \to A$
1. $\Box(A \to A)$	By necessitation on 0
2. $\Box(A \to A) \to (A \rhd A)$	By J1
3. $A \rhd A$	By MP 2, 1

■

And now the same proof in Agda.

```
⊢A▷A : ∀ {A Π} → Π ⊢ A ▷ A
⊢A▷A {A} = MP J1 (nec ⊢A↝A)
```

Although the Agda proofs of $A \to A$ and $A \rhd A$ are short, it is very hard for a human to fully understand them with the Agda syntax used above. This problem motivates our next chapter, in which we present a way to express syntactic proofs in Agda using paper-like syntax.

31. An eDSL for syntactic proofs

In this chapter we present a verified language for writing Hilbert style proofs for logic IL. The language has been designed and implemented by the author of this thesis. It was first presented in [24] for logic K.

We begin by introducing the concept of eDSL. The acronym eDSL stands for *Embedded Domain Specific Language*. It refers to a small language (a set of functions and datatypes) embedded in another language (in this case Agda) that has been designed to solve a problem in a very specific domain, in this case, Hilbert style proofs.

We begin by showing how we could write the two syntactic proofs presented in the previous section in our eDSL. Then we will present the language in detail.

The first example shows how we can formalize the proof $\vdash_{\mathsf{IL}} A \to A$ in the new language.

```
⊢A↝A : ∀ {A} → [] ⊢ A ↝ A
⊢A↝A {A} =
  begin[ 0 ] (A ↝ ((A ↝ A) ↝ A)) ↝ ((A ↝ (A ↝ A)) ↝ (A ↝ A)) By C2
       [ 1 ] A ↝ ((A ↝ A) ↝ A)                               By C1
       [ 2 ] A ↝ (A ↝ A)                                     By C1
       [ 3 ] (A ↝ (A ↝ A)) ↝ A ↝ A                          ByMP 0 , 1
       [ 4 ] A ↝ A                                          ByMP 3 , 2
       ■
```

Second example:

```
⊢A▷A : ∀ {A} → [] ⊢ A ▷ A
⊢A▷A {A} =
  begin[ 0 ] A ↝ A                      By ⊢A↝A
       [ 1 ] □ (A ↝ A)                  ByNec 0
       [ 2 ] □ (A ↝ A) ↝ (A ▷ A)        By J1
       [ 3 ] A ▷ A                      ByMP 2 , 1
       ■
```

We see that our eDSL allows us to write syntactic proofs in a very similar human-friendly syntax which is almost identical to the pen and paper usual syntax with the standout benefit that the proof is computer checked. In particular, it is checked by Agda's type checking algorithm.

We want to emphasize, as it may be surprising to the reader, that the proofs shown above are actual Agda code. It is crucial to make clear that this eDSL is *not* an alternative definition of the logic IL presented in different syntax. The eDSL is layer above the definition which allows us to use nice syntax while still relying on the simple definition of IL we provided in the previous chapter. Of course, the main challenge is to prove that we can transform proofs in the nice syntax to proofs in the original syntax. We will comment on it later on this section.

We proceed by giving a short description of the language, which consists of four types of instructions:

1. `[_]_By_`. This instruction is used to include a theorem in the proof. The theorem can be any axiom scheme of IL or anything proved to be a theorem. More precisely, the theorem can be any `A` if we have `Π ⊢ A` for some `Π`. The first instruction must be of this kind and must be preceded with `begin`.

2. `[_]_ByNec_`. This instruction applies the necessitation rule to a formula in a previous line referenced by its number. This rule can only be applied if we have an empty set of assumptions.

3. `[_]_ByMP_`. This instruction applies the modus ponens rule to two formulas in previous lines referenced by their number.

4. ■. The proof must be closed using this instruction.

Every instruction must be numbered in increasing order starting at 0.

Thanks to the design of the language, if the user mistakenly numbers one of the instructions Agda will report an error indicating where the error is. If the user improperly instantiates an axiom scheme or theorem or references an incorrect line, they will also be prompted with an error. Summarizing, an error will appear if the proof has any deficiency. This holds true as the eDSL is implemented in Agda and thus it is verified.

We proceed by giving a rough approximation on how the language has been implemented. Details of the inner workings of the language are left out as they fall out of the scope of this paper, however, we encourage the reader to look for further information in [24] if they are interested.

We begin by defining the datatype that represents our language.

```
data HilbertProof : List Fm → Fm → Nat → Set where
  begin : ∀ {Σ A} → Σ ⊢ A → HilbertProof Σ A 0
  by : ∀ {Σ A B n} → Σ ⊢ B → HilbertProof Σ A n → HilbertProof Σ B (suc n)
  Ax : ∀ {Σ B n} → (A : Fm) → HilbertProof Σ B n → HilbertProof (A :: Σ) A (suc n)
  nec : ∀ {Σ n □A C} (H : HilbertProof [] C n) (i : HilbertRef H (□A) □_)
    → HilbertProof Σ (□A) (suc n)
  MP : ∀ {n Σ A B C} (H : HilbertProof Σ C n) → HilbertRef H (A ⇝ B) id
    → HilbertRef H A id → HilbertProof Σ B (suc n)
```

Each instruction (except ■) has its corresponding constructor. We see that the datatype is indexed by a list of formulas and a formula. Those are the set of assumptions and the formula that is shown to be a theorem. The third index is a natural number. This number keeps track of the length of the proof and it is needed to ensure that references to previous lines are not out of bounds. The type `HilbertRef` represents a reference to a previous line in the proof. We omit its definition here as is not crucial for understanding the overall idea.

Then we define the front end syntax for each of the constructors. For instance, the definition of the `[_]_By_` instruction is as follows (we omit the type for simplicity as it is the same as the type of the constructor `by`).

```
infixl 10 _[_]_By_
_[_]_By_ : ...
H [ n ] B By p = by p H
```

Notice that we also declare the instruction to have left associativity (`infixl`), which will allow us to write each subsequent instruction below the other without need of parentheses.

We skip how references work for simplicity, as they use advanced Agda features (type classes and instance arguments ([3])) in order to be automatically checked without an explicit proof.

Observe now how we can build a proof in this language.

```
⊢A▷A' : ∀ {A} → HilbertProof [] (A ▷ A) 3
⊢A▷A' {A} =
  begin[ 0 ] A ⇝ A                    By ⊢A⇝A
       [ 1 ] □ (A ⇝ A)                ByNec 0
       [ 2 ] □ (A ⇝ A) ⇝ (A ▷ A)      By J1
       [ 3 ] A ▷ A                    ByMP 2 , 1
```

It is essentially the same as we showcased before but it is lacking the closing ■ instruction. The type of such instruction is:

```
_■ : ∀ {n Σ A} → HilbertProof Σ A n → Σ ⊢ A
```

We see that ■ is defined as a postfix operator which translates a proof `HilbertProof Σ A n` into a proof `Σ ⊢ A`. This translation step is where most of the complexity lays. Needless to say, as is implemented and verified in Agda it is guaranteed to be correct. To reiterate, by correct we mean that every proof in the new syntax can be transformed (and definition of the term ■, which can be found in Appendix B.22, is actually the algorithm which performs the transformation) to a proof using only axioms and rules of IL as presented in the previous section. This ends the tour of the language.

We strongly believe in the practical usefulness of this language as it can be used by logicians that are not Agda experts due to its simple and familiar syntax. We are all aware that long syntactic proofs are error prone. This language completely removes such problem. Of course, there is some room for improvement, for instance, the language does not include the deduction theorem rule, which is frequently used in practice. Note that this limitation can be ameliorated as we can use the deduction theorem outside of the eDSL and then include the result by using a `[_]_By_` instruction.

Part VI.

Glossary and bibliography

Glossary

$\Vdash^{gen}_{M}$ Forcing relation for generalized semantics 26

$\Vdash^{ord}_{M}$ Forcing relation for ordinary semantics 23

IL Base logic for interpretability logics 19

A definition or theorem formalized in Agda 14, 18–20, 22–27, 29, 31, 36–38, 40–42, 46–51, 53, 54, 56, 58–61, 116, 117, 119, 120

A definition or theorem formalized in Coq 15, 18, 19, 22–24, 117

`Pred` A predicate or a subset 105

`REL` Heterogeneous relation 104

`Rel` Homogeneous relation 104

choice set Choice set 54

decidable model A model whose forcing relation is decidable 116

dependent pair A pair in which the type of the second component may depend on the first component 105

frame Generalized Veltman frame 25

frame Ordinary Veltman frame 22

modally equivalent models Two models which have modally equivalent worlds 34

modally equivalent worlds Two worlds that force the same formulas 34

model Generalized Veltman model 26

model Ordinary Veltman model 23

multi-decidable model A model whose forcing relation is decidable for sets 119

Noetherian Conversely well-founded relation 113

Bibliography

[1] Andreas Abel. “foetus – Termination checker for simple functional programs”. In: (1998).

[2] Andreas Abel, Brigitte Pientka, David Thibodeau, and Anton Setzer. “Programming Infinite Structures by Observations”. In: (2013).

[3] *Agda’s documentation*. `https://agda.readthedocs.io/en/latest/`. 2020.

[4] Mark van Atten. “The Development of Intuitionistic Logic”. In: *The Stanford Encyclopedia of Philosophy*. Ed. by Edward N. Zalta. Winter 2017. Metaphysics Research Lab, Stanford University, 2017.

[5] Alessandro Berarducci. “The interpretability logic of Peano arithmetic”. In: *Journal of Symbolic Logic* (1990), pp. 1059–1089.

[6] Marta Bílková, Dick de Jongh, and Joost J. Joosten. “Interpretability in PRA”. In: *Annals of Pure and Applied Logic* 161.2 (2009), pp. 128–138.

[7] Ana Bove, Peter Dybjer, and Ulf Norell. “A brief overview of Agda–a functional language with dependent types”. In: *International Conference on Theorem Proving in Higher Order Logics*. Springer. 2009, pp. 73–78.

[8] Jesper Cockx and Andreas Abel. “Elaborating dependent (co)pattern matching”. In: *Proceedings of the ACM on Programming Languages* 2.ICFP (2018), pp. 1–30.

[9] Thierry Coquand. “Type Theory”. In: *The Stanford Encyclopedia of Philosophy*. Ed. by Edward N. Zalta. Fall 2018. Metaphysics Research Lab, Stanford University, 2018.

[10] Nils Anders Danielsson, Ulf Norell, SC Mu, S Bronson, D Doel, P Jansson, and LT et al Chen. “The Agda standard library”. In: (2020). URL: `https://github.com/agda/agda-stdlib`.

[11] Nicolaas Govert De Bruijn. “Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem”. In: *Indagationes Mathematicae (Proceedings)*. Vol. 75. 5. North-Holland. 1972, pp. 381–392.

[12] Peter Dybjer and Erik Palmgren. “Intuitionistic Type Theory”. In: *The Stanford Encyclopedia of Philosophy*. Ed. by Edward N. Zalta. Summer 2020. Metaphysics Research Lab, Stanford University, 2020.

[13] Kurt Gödel. “Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I”. In: *Monatshefte für mathematik und physik* 38.1 (1931), pp. 173–198.

[14] Evan Goris and Joost J. Joosten. “A new principle in the interpretability logic of all reasonable arithmetical theories”. In: *Logic Journal of the IGPL* 19.1 (2011), pp. 14–17.

[15] Evan Goris and Joost J. Joosten. “Modal Matters for Interpretability Logics”. In: *Logic Journal of the IGPL* 16.4 (Aug. 2008), pp. 371–412. ISSN: 1367-0751. DOI: `10.1093/jigpal/jzn013`. eprint: `https://academic.oup.com/jigpal/article-pdf/16/4/371/2055430/jzn013.pdf`. URL: `https://doi.org/10.1093/jigpal/jzn013`.

[16] Evan Goris and Joost J. Joosten. “Two New Series of Principles in the Interpretability Logic of All Reasonable Arithmetical Theories”. In: *Journal of Symbolic Logic* 85.1 (2020), pp. 1–25. DOI: `10.1017/jsl.2019.90`.

[17] Dick de Jongh and Frank Veltman. “Provability logics for relative interpretability”. In: *Mathematical Logic, Proceedings of the Heyting 1988 summer school in Varna, Bulgaria.* Ed. by P.P. Petkov. New York: Plenum Press, Boston, 1990, pp. 31–42. ISBN: 978-1-4612-7890-0. DOI: 10.1007/978-1-4613-0609-2_3.

[18] Joost J. Joosten. “Towards the interpretability logic of all reasonable arithmetical theories”. MA thesis. University of Amsterdam, 1998.

[19] Joost J. Joosten, Jan Mas Rovira, Luka Mikec, and Mladen Vuković. *An overview of Generalised Veltman Semantics.* submitted 2020. arXiv: 2007.04722 [math.LO].

[20] Joost J. Joosten and Albert Visser. “The interpretability logic of all reasonable arithmetical theories. The new conjecture”. In: *Erkenntnis* 53.1-2 (2000), pp. 3–26. ISSN: 0165-0106.

[21] Martin Hugo Löb. “Solution of a problem of Leon Henkin”. In: *The Journal of Symbolic Logic* 20.2 (1955), pp. 115–118.

[22] Per Martin-Löf. “Unpublished manuscript (An intuitionistic theory of types)”. Amsterdam, 1971.

[23] Per Martin-Löf and Giovanni Sambin. *Intuitionistic type theory.* Vol. 9. Bibliopolis Naples, 1984.

[24] Jan Mas Rovira. *Personal blog.* 2020. URL: https://janmasrovira.gitlab.io/ascetic-slug/ (visited on 06/22/2020).

[25] Jan Mas Rovira, Luka Mikec, and Joost J. Joosten. “Generalised Veltman semantics in Agda”. In: *Short Papers, Advances in Modal Logic, AiML 2020.* Ed. by R. Verbrugge and N. Olivetti. 2020, pp. 86–90. URL: https://www.helsinki.fi/sites/default/files/atoms/files/finalshortpapermain.pdf.

[26] Luka Mikec and Mladen Vuković. “Interpretability Logics and Generalised Veltman Semantics”. In: *The Journal of Symbolic Logic* (June 2020), pp. 1–21. DOI: 10.1017/jsl.2020.7.

[27] Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf’s type theory.* Vol. 200.

[28] Ulf Norell. “Dependently typed programming in Agda”. In: *International school on advanced functional programming.* Springer. 2008, pp. 230–266.

[29] Ulf Norell. “Towards a practical programming language based on dependent type theory”. PhD thesis. SE-412 96 Göteborg, Sweden: Department of Computer Science and Engineering, Chalmers University of Technology, Sept. 2007.

[30] Panu Raatikainen. “Gödel’s Incompleteness Theorems”. In: *The Stanford Encyclopedia of Philosophy.* Ed. by Edward N. Zalta. Winter 2020. Metaphysics Research Lab, Stanford University, 2020.

[31] Vladimir Yurievich Shavrukov. *The logic of relative interpretability over Peano arithmetic.* Preprint. In Russian. Moscow: Steklov Mathematical Institute, 1988.

[32] Robert Martin Solovay. “Provability interpretations of modal logic”. In: *Israel journal of mathematics* 25.3-4 (1976), pp. 287–304.

[33] The Coq Development Team. *Coq.* Version 8.12. Nov. 5, 2020. URL: https://coq.inria.fr.

[34] Alan Turing. “Computability and λ-definability”. In: *The Journal of Symbolic Logic* 2.4 (1937), pp. 153–163.

[35] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics.* Institute for Advanced Study: https://homotopytypetheory.org/book, 2013.

[36] L.C. Verbrugge. “Verzamelingen-Veltman frames en modellen (Set Veltman frames and models)”. Unpublished manuscript. 1992.

[37] Rineke Verbrugge. “Provability Logic”. In: *The Stanford Encyclopedia of Philosophy*. Ed. by Edward N. Zalta. Fall 2017. Metaphysics Research Lab, Stanford University, 2017.

[38] Albert Visser. “An overview of Interpretability Logic”. In: *Advances in modal logic ’96*. Ed. by M. Kracht, M. de Rijke, and H. Wansing. CSLI Publications, Stanford, CA, 1997, pp. 307–359.

[39] Albert Visser. “Interpretability Logic”. In: *Mathematical Logic*. Ed. by Petio Petrov Petkov. Boston, MA: Springer US, 1990, pp. 175–209. ISBN: 978-1-4613-0609-2. DOI: 10.1007/978-1-4613-0609-2_13. URL: https://doi.org/10.1007/978-1-4613-0609-2_13.

[40] Albert Visser. “The formalization of interpretability”. In: *Studia Logica* 50.1 (1991), pp. 81–106.

[41] Mladen Vuković. “Bisimulations between generalized Veltman models and Veltman models”. In: *Mathematical Logic Quarterly* 54.4 (2008), pp. 368–373.

[42] Philip Wadler and Wen Kokke. *Programming Language Foundations in Agda*. Available at http://plfa.inf.ed.ac.uk/. 2019.

Part VII.

Appendix

A. Official Agda reference

This appendix contains a thorough description of some of the Agda related topics that were discussed in the thesis. We believe that the intuition given in the introduction should be enough for the reader to be able to read Part V, however, we encourage the reader to resort to the ensuing reference if they wish for more precise information on the language. The contents of this section have been merely copied from the online Agda documentation ([3]).

A.1. Function definitions and pattern matching

A function is defined by first declaring its type followed by a number of equations called clauses. Each clause consists of the function being defined applied to a number of patterns, followed by = and a term called the right-hand side. For example:

```
not : Bool → Bool
not true  = false
not false = true
```

Functions are allowed to call themselves recursively, for example:

```
twice : Nat → Nat
twice zero    = zero
twice (suc n) = suc (suc (twice n))
```

The general form for defining a function is

```
f : (x₁ : A₁) → … → (xₙ : Aₙ) → B
f p₁ … pₙ = d
…
f q₁ … qₙ = e
```

where f is a new identifier, p_i and q_i are patterns of type A_i, and d and e are expressions.

The declaration above gives the identifier f the type `(x₁ : A₁) → … → (xₙ : Aₙ) → B` and f is defined by the defining equations. Patterns are matched from top to bottom, i.e., the first pattern that matches the actual parameters is the one that is used.

By default, Agda checks the following properties of a function definition:

1. The patterns in the left-hand side of each clause should consist only of constructors and variables.
2. No variable should occur more than once on the left-hand side of a single clause.
3. The patterns of all clauses should together cover all possible inputs of the function.
4. The function should be terminating on all possible inputs.

A.2. Absurd patterns

Absurd patterns can be used when none of the constructors for a particular argument would be valid. The syntax for an absurd pattern is `()`.

As an example, if we have a datatype Even defined as follows:

```
data Even : Nat → Set where
 even-zero  : Even zero
 even-plus2 : {n : Nat} → Even n → Even (suc (suc n))
#+end_src text

Then we can define a function =one-not-even : Even 1 → ⊥= by using an absurd
pattern:
#+begin_src text
one-not-even : Even 1 → ⊥
one-not-even ()
```

Note that if the left-hand side of a clause contains an absurd pattern, its right-hand side must be omitted.

In general, when matching on an argument of type $D\ i_1\ \ldots\ i_n$ with an absurd pattern, Agda will attempt for each constructor $c\ :\ (x_1\ :\ A_1) \to \ldots \to (x_m\ :\ A_m) \to D\ j_1\ \ldots\ j_n$ of the datatype `D` to unify $i_1\ \ldots\ i_n$ with $j_1\ \ldots\ j_n$. The absurd pattern will only be accepted if all of these unifications end in a conflict.

A.3. Implicit arguments and automatic inference

It is possible to omit terms that the type checker can figure out for itself, replacing them by _. If the type checker cannot infer the value of an _ it will report an error. For instance, for the polymorphic identity function

```
id : (A : Set) → A → A
```

the first argument can be inferred from the type of the second argument, so we might write `id _ zero` for the application of the identity function to `zero`.

We can even write this function application without the first argument. In that case we declare an implicit function space:

```
id : {A : Set} → A → A
```

and then we can use the notation `id zero`.

Another example:

```
_==_  : {A : Set} → A → A → Set
subst : {A : Set} (C : A → Set) {x y : A} → x == y → C x → C y
```

Note how the first argument to _==_ is left implicit. Similarly, we may leave out the implicit arguments `A`, `x`, and `y` in an application of `subst`. To give an implicit argument explicitly, enclose it in curly braces. The following two expressions are equivalent:

```
x1 = subst C eq cx
x2 = subst {_} C {_} {_} eq cx
```

It is worth noting that implicit arguments are also inserted at the end of an application, if it is required by the type. For example, in the following, `y1` and `y2` are equivalent.

```
y1 : a == b → C a → C b
y1 = subst C

y2 : a == b → C a → C b
y2 = subst C {_} {_}
```

Implicit arguments are inserted eagerly in left-hand sides so `y3` and `y4` are equivalent. An exception is when no type signature is given, in which case no implicit argument insertion takes place. Thus in the definition of `y5` the only implicit is the `A` argument of `subst`.

```
y3 : {x y : A} → x == y → C x → C y
y3 = subst C

y4 : {x y : A} → x == y → C x → C y
y4 {x} {y} = subst C {_} {_}

y5 = subst C
```

It is also possible to write lambda abstractions with implicit arguments. For example, given `id : (A : Set) → A → A`, we can define the identity function with implicit type argument as

```
id' = λ {A} → id A
```

Implicit arguments can also be referred to by name, so if we want to give the expression `e` explicitly for `y` without giving a value for `x` we can write

```
subst C {y = e} eq cx
```

In rare circumstances it can be useful to separate the name used to give an argument by name from the name of the bound variable, for instance if the desired name shadows an existing name. To do this you write

```
id₂ : {A = X : Set} → X → X  -- name of bound variable is X
id₂ x = x

use-id₂ : (Y : Set) → Y → Y
use-id₂ Y = id₂ {A = Y}      -- but the label is A
```

Labeled bindings must appear by themselves when typed, so the type `Set` needs to be repeated in this example:

```
const : {A = X : Set} {B = Y : Set} → A → B → A
const x y = x
```

When constructing implicit function spaces the implicit argument can be omitted, so both expressions below are valid expressions of type `{A : Set} → A → A`:

```
z1 = λ {A} x → x
z2 = λ x → x
```

The ∀ (or `forall`) syntax for function types also has implicit variants:

```
① : (∀ {x : A} → B)    is-the-same-as  ({x : A} → B)
② : (∀ {x} → B)        is-the-same-as  ({x : _} → B)
③ : (∀ {x y} → B)      is-the-same-as  (∀ {x} → ∀ {y} → B)
```

In very special situations it makes sense to declare unnamed hidden arguments `{A} → B`. In the following `example`, the hidden argument to `scons` of type `zero ≤ zero` can be solved by η-expansion, since this type reduces to `⊤`.

```
data ⊥ : Set where

_≤_ : Nat → Nat → Set
zero ≤ _      = ⊤
suc m ≤ zero  = ⊥
suc m ≤ suc n = m ≤ n

data SList (bound : Nat) : Set where
[]    : SList bound
scons : (head : Nat) → {head ≤ bound} → (tail : SList head) → SList bound

example : SList zero
example = scons zero []
```

There are no restrictions on when a function space can be implicit. Internally, explicit and implicit function spaces are treated in the same way. This means that there are no guarantees that implicit arguments will be solved. When there are unsolved implicit arguments the type checker will give an error message indicating which application contains the unsolved arguments. The reason for this liberal approach to implicit arguments is that limiting the use of implicit argument to the cases where we guarantee that they are solved rules out many useful cases in practice.

A.4. datatype definitions and constructors

The general form of the definition of a simple datatype `D` is the following

```
data D (x_1 : P_1) ... (x_k : P_k) : (y_1 : Q_1) → ... → (y_l : Q_l) → Set ℓ where
  c_1 : A_1
  ...
  c_n : A_n
```

The name `D` of the data type and the names c_1, ..., c_n of the constructors must be new w.r.t. the current signature and context, and the types A_1, ..., A_n must be function types ending in `D`, i.e. they must be of the form

```
(y_1 : B_1) → ... → (y_m : B_m) → D
```

Datatypes can have parameters. They are declared after the name of the datatype but before the colon, for example:

```
data List (A : Set) : Set where
  []  : List A
  _::_ : A → List A → List A
```

In addition to parameters, datatypes can also have indices. In contrast to parameters which are required to be the same for all constructors, indices can vary from constructor to constructor. They are declared after the colon as function arguments to `Set`. For example, fixed-length vectors can be defined by indexing them over their length of type `Nat`:

```
data Vector (A : Set) : Nat → Set where
  []  : Vector A zero
  _::_ : {n : Nat} → A → Vector A n → Vector A (suc n)
```

Notice that the parameter `A` is bound once for all constructors, while the index `{n : Nat}` must be bound locally in the constructor _::_.

Indexed datatypes can also be used to describe predicates, for example the predicate `Even : Nat → Set` can be defined as follows:

```
data Even : Nat → Set where
  even-zero  : Even zero
  even-plus2 : {n : Nat} → Even n → Even (suc (suc n))
```

The general form of the definition of a (parametrized, indexed) datatype `D` is the following

```
data D (x₁ : P₁) ... (xₖ : Pₖ) : (y₁ : Q₁) → ... → (yₗ : Qₗ) → Set ℓ where
  c₁ : A₁
  ...
  cₙ : Aₙ
```

where the types A_1, ..., A_n are function types of the form

```
(z₁ : B₁) → ... → (zₘ : Bₘ) → D x₁ ... xₖ t₁ ... tₗ
```

A.5. Function types

Function types are written `(x : A) → B`, or in the case of non-dependent functions simply `A → B`. For instance, the type of the addition function for natural numbers is:

```
Nat → Nat → Nat
```

and the type of the addition function for vectors is:

```
(A : Set) → (n : Nat) → (u : Vec A n) → (v : Vec A n) → Vec A n
```

where `Set` is the type of sets and `Vec A n` is the type of vectors with `n` elements of type `A`. Arrows between consecutive hypotheses of the form `(x : A)` may also be omitted, and `(x : A) (y : A)` may be shortened to `(x y : A)`:

```
(A : Set) (n : Nat) (u v : Vec A n) → Vec A n
```

Functions are constructed by lambda abstractions, which can be either typed or untyped. For instance, both expressions below have type `(A : Set) → A → A` (the second expression checks against other types as well):

```
example₁ = λ (A : Set) (x : A) → x
example₂ = λ A x → x
```

The application of a function `f : (x : A) → B` to an argument `a : A` is written `f a` and the type of this is `B[x := a]`.

Some notation conventions follow.

- Function types:

```
prop₁ : ((x : A) (y : B) → C) is-the-same-as   ((x : A) → (y : B) → C)
prop₂ : ((x y : A) → C)       is-the-same-as   ((x : A)(y : A) → C)
prop₃ : (∀ (x : A) → C)  is-the-same-as   ((x : A) → C)
prop₄ : (∀ x → C)        is-the-same-as   ((x : _) → C)
prop₅ : (∀ x y → C)      is-the-same-as   (∀ x → ∀ y → C)
```

- Functional abstraction:

```
(λ x y → e)                    is-the-same-as   (λ x → (λ y → e))
```

- Functional application:

```
(f a b)                        is-the-same-as   ((f a) b)
```

A.6. Record types

The general form of a record declaration is as follows:

```
record <recordname> <parameters> : Set <level> where
  <directives>
  constructor <constructorname>
  field
    <fieldname1> : <type1>
    <fieldname2> : <type2>
    -- ...
  <declarations>
```

All the components are optional, and can be given in any order. In particular, fields can be given in more than one block, interspersed with other declarations. Each field is a component of the record. Types of later fields can depend on earlier fields.

The directives available are eta-equality, no-eta-equality, inductive and co-inductive. For more information visit [3].

A.7. Universes

Russell's paradox implies that the collection of all sets is not itself a set. Namely, if there were such a set U, then one could form the subset A ⊆ U of all sets that do not contain themselves. Then we would have A ∈ A if and only if A ∉ A, a contradiction.

For similar reasons, not every Agda type is a Set. For example, we have

```
Bool : Set
Nat : Set
```

but not Set : Set. However, it is often convenient for Set to have a type of its own, and so in Agda, it is given the type $\texttt{Set}_1$:

```
Set : Set₁
```

In many ways, expressions of type $\texttt{Set}_1$ behave just like expressions of type Set; for example, they can be used as types of other things. However, the elements of $\texttt{Set}_1$ are potentially larger; when A : $\texttt{Set}_1$, then A is sometimes called a large set. In turn, we have:

```
Set₁ : Set₂
Set₂ : Set₃
```

and so on. A type whose elements are types is called a universe; Agda provides an infinite number of universes Set, $\texttt{Set}_1$, $\texttt{Set}_2$, $\texttt{Set}_3$, ..., each of which is an element of the next one. In fact, Set itself is just an abbreviation for $\texttt{Set}_0$. The subscript n is called the level of the universe $\texttt{Set}_n$.

A note on syntax: you can also write Set1, Set2, etc., instead of $\texttt{Set}_1$, $\texttt{Set}_2$. To enter a subscript in the Emacs mode, type _1.

A.7.1. Universe example

So why are universes useful? Because sometimes it is necessary to define, and prove theorems about, functions that operate not just on sets but on large sets. In fact, most Agda users sooner or later experience an error message where Agda complains that $\texttt{Set}_1$!= Set. These errors usually mean that a small set was used where a large one was expected, or vice versa.

For example, suppose you have defined the usual datatypes for lists and Cartesian products:

```
data List (A : Set) : Set where
  [] : List A
  _::_ : A → List A → List A

data _×_ (A B : Set) : Set where
 _,_ : A → B → A × B

infixr 5 _::_
infixr 4 _,_
infixr 2 _×_
```

Now suppose you would like to define an operator `Prod` that inputs a list of `n` sets and takes their Cartesian product, like this:

```
Prod (A :: B :: C :: []) = A × B × C
```

There is only one small problem with this definition. The type of `Prod` should be:

```
Prod : List Set → Set
```

However, the definition of `List A` specified that `A` had to be a `Set`. Therefore, `List Set` is not a valid type. The solution is to define a special version of the `List` operator that works for large sets:

```
data List₁ (A : Set₁) : Set₁ where
  []   : List₁ A
  _::_ : A → List₁ A → List₁ A

With this, we can indeed define:

Prod : List₁ Set → Set
Prod []       = ⊤
Prod (A :: As) = A × Prod As
```

A.7.2. Universe polymorphism

Although we were able to give a type to the `Prod` operator by defining a special notion of large list, this quickly gets tiresome. Sooner or later, we find that we require yet another list type `List₂`, and it doesn't stop there. Also every function on lists (such as append) must be re-defined, and every theorem about such functions must be re-proved, for every possible level.

The solution to this problem is universe polymorphism. Agda provides a special primitive type `Level`, whose elements are possible levels of universes. In fact, the notation for the `n` th universe, $\texttt{Set}_n$, is just an abbreviation for `Set n`, where `n : Level` is a level. We can use this to write a polymorphic `List` operator that works at any level. The library `Agda.Primitive` must be imported to access the `Level` type. The definition then looks like this:

```
open import Agda.Primitive

data List {n : Level} (A : Set n) : Set n where
[]   : List A
_::_ : A → List A → List A
```

This new operator works at all levels; for example, we have

```
List Nat : Set
List Set : Set₁
List Set₁ : Set₂
```

A.7.3. Level arithmetic

Even though we don't have the number of levels specified, we know that there is a lowest level `lzero`, and for each level `n`, there exists some higher level `lsuc n`; therefore, the set of levels is infinite. In addition, we can also take the least upper bound `n ⊔ m` of two levels. In summary, the following (and only the following) operations on levels are provided:

```
lzero : Level
lsuc  : (n : Level) → Level
_⊔_   : (n m : Level) → Level
```

This is sufficient for most purposes; for example, we can define the Cartesian product of two types of arbitrary (and not necessarily equal) levels like this:

```
data _×_ {n m : Level} (A : Set n) (B : Set m) : Set (n ⊔ m) where
   _,_ : A → B → A × B
```

With this definition, we have, for example:

```
Nat × Nat : Set
Nat x Set : Set₁
Set × Set : Set₁
```

B. Agda library code

B.1. All

This is a helper module which imports everything in the library. It is used to check all the library at once.

```
module _ where

-- This modules imports everything and it is meant to help checking
the whole
-- library at once.

open import Base
open import Classical
open import Formula
open import Principles

open import IL
open import IL.Properties
open import IL.Edsl

open import OrdinaryVeltmanSemantics
open import OrdinaryVeltmanSemantics.Finite
open import OrdinaryVeltmanSemantics.Properties
open import OrdinaryVeltmanSemantics.Properties.M
open import OrdinaryVeltmanSemantics.Properties.P₀
open import OrdinaryVeltmanSemantics.Properties.R
open import OrdinaryVeltmanSemantics.Properties.M₀

open import GeneralizedVeltmanSemantics
open import GeneralizedVeltmanSemantics.Properties
open import GeneralizedVeltmanSemantics.Properties.M
open import GeneralizedVeltmanSemantics.Properties.M₀
open import GeneralizedVeltmanSemantics.Properties.P₀
open import GeneralizedVeltmanSemantics.Properties.Rⁿ
open import GeneralizedVeltmanSemantics.Properties.R
open import GeneralizedVeltmanSemantics.Properties.R¹
open import GeneralizedVeltmanSemantics.Properties.R²
open import GeneralizedVeltmanSemantics.Properties.R₁
open import GeneralizedVeltmanSemantics.Properties.GenericFrameCond
open import GeneralizedVeltmanSemantics.Properties.Verbrugge
open import GeneralizedVeltmanSemantics.Properties.Luka

open import GeneralizedFrame
open import GeneralizedFrame.Properties
open import OrdinaryFrame
```

B.2. Base

This module contains some definitions and helper functions which are used throughout the thesis. It includes, for instance, the definition of a Noetherian relation.

```
module Base where

open import Function.Equivalence using (_⇔_; equivalence; module Equivalence)

open import Agda.Builtin.Nat using (Nat; zero; suc)
open import Agda.Builtin.Bool
open import Data.Empty using (⊥; ⊥-elim)
open import Data.Product
open import Relation.Binary.PropositionalEquality using (_≡_; refl;
subst; trans; sym; cong)
open import Data.Sum using (_⊎_; inj₁; inj₂)
open import Function using (_∘_; const; case_of_; id)
open import Function.Equality using (_⟨$⟩_)
open import Level using (Level; _⊔_) renaming (suc to lsuc; zero to
lzero)
open import Relation.Binary using (REL; Rel; Transitive; Irreflexive)
open import Relation.Nullary using (Dec; yes; no)
open import Relation.Nullary using (¬_)
open import Relation.Unary using (Pred; Decidable; _⊆_) renaming (_⇒_
to _P⇒_)

private
  variable
    ℓa ℓb ℓc ℓ ℓ₁ ℓ₂ ℓ₃ : Level
    A : Set ℓa
    B : Set ℓb
    C : Set ℓc

infix 5 _⇒_
_⇒_ : ∀ {a b} {A : Set a} {B : Set b} → A ⇔ B → A → B
f ⇒ x  = Equivalence.to f ⟨$⟩ x

infix 5 _⇐_
_⇐_ : ∀ {f t} {A : Set f} {B : Set t} → A ⇔ B → B → A
f ⇐ x  = Equivalence.from f ⟨$⟩ x

REL₃ : Set ℓa → Set ℓb → Set ℓc → (ℓ : Level) → Set (ℓa ⊔ ℓb ⊔ ℓc ⊔
lsuc ℓ)
REL₃ A B C ℓ = A → B → C → Set ℓ

Rel₃ : Set ℓa → (ℓ : Level) → Set (ℓa ⊔ lsuc ℓ)
Rel₃ A ℓ = REL₃ A A A ℓ

Decidable₃ : REL₃ A B C ℓ → Set _
```

```
Decidable₃ R = ∀ x y z → Dec (R x y z)

subst₃ : ∀ {ℓ ℓ'} {A B C : Set ℓ} (R : REL₃ A B C ℓ') {w z x y u v} →
w ≡ z → x ≡ y → u ≡ v → R x u w → R y v z
subst₃ _ refl refl refl p = p

record InfiniteChain {ℓW ℓR} {W : Set ℓW} (_<_ : Rel W ℓR) (a : W)
  : Set (ℓR ⊔ ℓW) where
  coinductive
  field
    b : W
    a<b : a < b
    tail : InfiniteChain _<_ b

infiniteRefl : ∀ {R : Rel A ℓ} {x} → R x x → InfiniteChain R x
InfiniteChain.b (infiniteRefl {x = x} Rxx) = x
InfiniteChain.a<b (infiniteRefl {x = x} Rxx) = Rxx
InfiniteChain.tail (infiniteRefl {x = x} Rxx) = infiniteRefl Rxx

Noetherian : ∀ {ℓR ℓW} {W : Set ℓW} → Rel W ℓR → Set (ℓR ⊔ ℓW)
Noetherian _<_ = ∀ {a} → ¬ (InfiniteChain _<_ a)

Noetherian⇒Irreflexive : ∀ {ℓR ℓW} {W : Set ℓW} {R : Rel W ℓR} →
Noetherian R → Irreflexive _≡_ R
Noetherian⇒Irreflexive noeth refl Rxx = noeth (infiniteRefl Rxx)

Noetherian⇒Irreflexive' : ∀ {ℓR ℓW} {W : Set ℓW} {R : Rel W ℓR} →
Noetherian R → ∀ {x} → ¬ R x x
Noetherian⇒Irreflexive' noeth Rxx = noeth (infiniteRefl Rxx)

data _≤_ : Rel Nat lzero where
  z≤n : (a : Nat) → zero ≤ a
  s≤s : {a b : Nat} → a ≤ b → suc a ≤ suc b

≤-trans : Transitive _≤_
≤-trans {a} {b} {c} (z≤n b) b≤c = z≤n c
≤-trans {suc a} {suc b} {suc c} (s≤s a≤b) (s≤s b≤c) = s≤s (≤-trans a≤b
b≤c)

reflex : ∀ {ℓ} {A : Set ℓ} {a : A} → a ≡ a
reflex = refl

symm : ∀ {ℓ} {A : Set ℓ} {a b : A} → a ≡ b → b ≡ a
symm refl = refl

transitivity : ∀ {ℓ} {A : Set ℓ} {a b c : A} → a ≡ b → b ≡ c → a ≡ c
transitivity refl refl = refl
```

B.3. Classical

```
module Classical where
```

```
open import Function.Equivalence using (_⇔_; equivalence; map; module
Equivalence)

open import Agda.Builtin.Nat using (Nat)
open import Agda.Builtin.Unit using (⊤; tt)
open import Data.List using (List; []; _∷_)
open import Data.Sum using (_⊎_; inj₁; inj₂)
open import Data.List.Membership.Propositional using (_∈_)
open import Data.List.Relation.Unary.Any using (Any; here; there)
open import Data.Product using (Σ; proj₁; proj₂; _×_) renaming (_,_ to
_,_)
open import Relation.Binary.PropositionalEquality using (_≡_; refl)
open import Base using (_⇒_; _⇐_)

Var : Set
Var = Nat

infixr 20 _⤳_
data Fm : Set where
  var : Var → Fm
  ⊥' : Fm
  _⤳_ : Fm → Fm → Fm

infix 60 ¬'_
¬'_ : Fm → Fm
¬' a = a ⤳ ⊥'

⊤' : Fm
⊤' = ¬' ⊥'

infix 30 _∨_
_∨_ : Fm → Fm → Fm
ϕ ∨ ψ = ¬' ϕ ⤳ ψ

infix 40 _∧_
_∧_ : Fm → Fm → Fm
a ∧ b = ¬' (a ⤳ ¬' b)

infix 5 _⊢_
data _⊢_ (Π : List Fm) : Fm → Set where
  -- classical axioms
  C1 : ∀ {A B} → Π ⊢ A ⤳ B ⤳ A
  C2 : ∀ {A B C} → Π ⊢ (A ⤳ B ⤳ C) ⤳ ((A ⤳ B) ⤳ A ⤳ C)
  C3 : ∀ {A B} → Π ⊢ (¬' A ⤳ ¬' B) ⤳ B ⤳ A
  MP : ∀ {A B} → Π ⊢ A ⤳ B → Π ⊢ A → Π ⊢ B
  Ax : ∀ {A} → A ∈ Π → Π ⊢ A


record ClassicalLanguage (fm : Set) : Set where
  field
```

```
    implication : fm → fm → fm
    varia : Var → fm
    bottom : fm


  negation : fm → fm
  negation x = implication x bottom

  -- infix 30 _∨'_
  -- _∨'_ : fm → fm → fm
  -- ϕ ∨' ψ = ¬'' ϕ c⇝ ψ

  -- infix 40 _∧_
  -- _∧_ : Fm → Fm → Fm
  -- a ∧ b = ¬' (a ⇝ ¬' b)

instance
  FmClassical : ClassicalLanguage Fm
  FmClassical = record { implication = _⇝_ ; varia = var ; bottom = ⊥'
}


record ExtendsClassical {fm : Set} {{l : ClassicalLanguage fm}} (_⊢_ :
List fm → fm → Set) : Set where
    infixr 20 _⇝'_
    _⇝'_ = ClassicalLanguage.implication l

    infix 60 ¬''_
    ¬''_ = ClassicalLanguage.negation l

    field C1' : ∀ {A B : fm} {Π : List fm} → Π ⊢ (A ⇝' B ⇝' A)
    field C2' : ∀ {A B C Π} → Π ⊢ ((A ⇝' B ⇝' C) ⇝' ((A ⇝' B) ⇝' A ⇝'
C))
    field C3' : ∀ {A B Π} → Π ⊢ ((¬'' A ⇝' ¬'' B) ⇝' B ⇝' A)
    field MP' : ∀ {A B Π} → Π ⊢ (A ⇝' B) → Π ⊢ A → Π ⊢ B
    field Ax' : ∀ {A Π} → A ∈ Π → Π ⊢ A

instance
  ClassicalExt : ExtendsClassical _⊢_
  ClassicalExt = record { C1' = C1 ; C2' = C2 ; C3' = C3 ; MP' = MP ;
Ax' = Ax }

-- fromClassical : ∀ {fm : Set} {{_ : ClassicalLanguage fm}} {_⊢'_ :
List fm → fm → Set} {{_ : ExtendsClassical _⊢'_}} {Π A} → Π ⊢ A → Π ⊢'
A
-- fromClassical = {!!}

weak : ∀ {Π A B} → Π ⊢ A → (B :: Π) ⊢ A
weak C1 = C1
weak C2 = C2
weak C3 = C3
weak (MP x x₁) = MP (weak x) (weak x₁)
weak (Ax x) = Ax (there x)

cut : ∀ {Π A B} → Π ⊢ B → (B :: Π) ⊢ A → Π ⊢ A
cut y C1 = C1
cut y C2 = C2
cut y C3 = C3
cut y (MP x x₁) = MP (cut y x) (cut y x₁)
cut y (Ax (here refl)) = y
cut y (Ax (there x)) = Ax x

deduction : ∀ {Π A B} → Π ⊢ A ↝ B ⇔ (A :: Π) ⊢ B
deduction {Π} {A} {B} = equivalence ⇨ ⇦
  where
  ⇨ : ∀ {Π A B} → Π ⊢ A ↝ B → (A :: Π) ⊢ B
  ⇨ x = MP (weak x) (Ax (here refl))
  ⇦ : ∀ {Π A B} → (A :: Π) ⊢ B → Π ⊢ A ↝ B
  ⇦ C1 = MP C1 C1
  ⇦ C2 = MP C1 C2
  ⇦ C3 = MP C1 C3
  ⇦ {Π} {A} {B} (MP {C} x y) = MP (MP C2 (⇦ x)) (⇦ y)
  ⇦ (Ax (here refl)) = MP (MP C2 C1) (C1 {_} {_} {⊥'})
  ⇦ (Ax (there x)) = MP C1 (Ax x)

⊢A↝A : ∀ {Π A} → Π ⊢ A ↝ A
⊢A↝A {_} {A} = deduction ⇐ Ax (here refl)

⊢A∨¬A : ∀ {Π A} → Π ⊢ A ∨ (¬' A)
⊢A∨¬A {A} = ⊢A↝A

trans : ∀ {A B C Π} → Π ⊢ (A ↝ B) ↝ (B ↝ C) ↝ A ↝ C
trans {A} {B} {C} = deduction ⇐ (deduction ⇐ (deduction ⇐ MP (Ax (there
(here refl)))
  (MP (Ax (there (there (here refl)))) (Ax (here refl)))))

⊢⟦A↝B⟧↝⟦B↝C⟧↝A↝C : ∀ {A B C Π} → Π ⊢ (A ↝ B) ↝ (B ↝ C) ↝ A ↝ C
⊢⟦A↝B⟧↝⟦B↝C⟧↝A↝C = trans

⊢A↝¬¬A : ∀ {Π A} → Π ⊢ A ↝ ¬' ¬' A
⊢A↝¬¬A = deduction ⇐ (deduction ⇐ MP (Ax (here refl)) (Ax (there (here
refl))))

⊢¬¬A↝A : ∀ {A Π} → Π ⊢ (¬' ¬' A) ↝ A
⊢¬¬A↝A {A} = MP C3 ⊢A↝¬¬A

⟦A↝B⟧↝¬B↝¬A : ∀ {A B Π} → Π ⊢ (A ↝ B) ↝ ¬' B ↝ ¬' A
⟦A↝B⟧↝¬B↝¬A = deduction ⇐ (deduction ⇐ (deduction ⇐ MP (Ax (there (here
refl)))
  (MP (Ax (there (there (here refl)))) (Ax (here refl)))))

⟦A↝B↝C⟧↝B↝A↝C : ∀ {A B C Π} → Π ⊢ (A ↝ B ↝ C) ↝ B ↝ A ↝ C
```

```
⟦A⇾B⇾C⟧⇾B⇾A⇾C = deduction ⇐ (deduction ⇐ (deduction ⇐ cut
   (MP (Ax (there (there (here refl)))) (Ax (here refl))) (MP (Ax (here
refl))
   (Ax (there (there (here refl)))))))

⊢A⇾⊤ : ∀ {Π A} → Π ⊢ A ⇾ ⊤'
⊢A⇾⊤ = deduction ⇐ (deduction ⇐ Ax (here refl))

⊢⊥⇾A : ∀ {Π A} → Π ⊢ ⊥' ⇾ A
⊢⊥⇾A = MP C3 ⊢A⇾⊤

⊢¬A⇾A⇾B : ∀ {A B Π} → Π ⊢ ¬' A ⇾ A ⇾ B
⊢¬A⇾A⇾B = MP (MP C2 (MP C1 C3)) C1

⊢⇾ : ∀ {A B Π} → (Π ⊢ ¬' A ⊎ Π ⊢ B) → Π ⊢ A ⇾ B
⊢⇾ (inj₁ x) = MP C3 (MP C1 x)
⊢⇾ (inj₂ y) = MP C1 y

⊢∨ : ∀ {Π A B} → (Π ⊢ A ⊎ Π ⊢ B) → Π ⊢ A ∨ B
⊢∨ (inj₁ x) = deduction ⇐ cut (weak x) (MP ⊢⊥⇾A (MP (Ax (there (here
refl)))
  (Ax (here refl))))
⊢∨ (inj₂ y) = deduction ⇐ weak y

⊢A∧B⇾A : ∀ {Π A B} → Π ⊢ A ∧ B ⇾ A
⊢A∧B⇾A = MP (MP trans (MP ⟦A⇾B⟧⇾¬B⇾¬A ⊢¬A⇾A⇾B)) ⊢¬¬A⇾A
⊢A∧B⇾B : ∀ {Π A B} → Π ⊢ A ∧ B ⇾ B
⊢A∧B⇾B = MP (MP trans (MP ⟦A⇾B⟧⇾¬B⇾¬A C1)) ⊢¬¬A⇾A

⊢A⇾B⇾A∧B : ∀ {Π A B} → Π ⊢ A ⇾ B ⇾ A ∧ B
⊢A⇾B⇾A∧B = deduction ⇐ (deduction ⇐ (deduction ⇐ cut (MP (Ax (here
refl))
   (Ax (there (there (here refl))))) (MP (Ax (here refl))
   (Ax (there (there (here refl)))))))

⊢∧ : ∀ {Π A B} → (Π ⊢ A × Π ⊢ B) ⇔ Π ⊢ A ∧ B
⊢∧ {Π} {A} {B} = equivalence ⇒ ⇐
   where
   ⇒ : (Π ⊢ A × Π ⊢ B) → Π ⊢ A ∧ B
   ⇒ (fst , snd) = MP (MP ⊢A⇾B⇾A∧B fst) snd
   ⇐ : Π ⊢ A ∧ B → (Π ⊢ A × Π ⊢ B)
   ⇐ x = MP ⊢A∧B⇾A x , MP ⊢A∧B⇾B x
```

B.4. Formula

The definition of formulas in the language of interpretability logics.

```
module Formula where

open import Agda.Builtin.Nat using (Nat; _-_)
open import Agda.Builtin.Char using (Char; primCharToNat)
open import Data.Empty using (⊥; ⊥-elim)
open import Function using (_∘_)
open import Relation.Binary using (REL; Rel)

Var : Set
Var = Nat

infixr 20 _↝_
infix 50 _▷_


data Fm : Set where
  var : Var → Fm
  ⊥' : Fm
  _↝_ : Fm → Fm → Fm
  _▷_ : Fm → Fm → Fm

car : Char → Fm
car c = var (primCharToNat c - primCharToNat 'a')

infix 60 ¬'_
¬'_ : Fm → Fm
¬' a = a ↝ ⊥'

~ : Fm → Fm
~ (var x) = ¬' (var x)
~ ⊥' = ¬' ⊥'
~ a@(x ↝ var _) = ¬' a
~ (a ↝ ⊥') = a
~ a@(_ ↝ _ ↝ _) = ¬' a
~ a@(_ ↝ _ ▷ _) = ¬' a
~ a@(_ ▷ _) = ¬' a

⊤' : Fm
⊤' = ¬' ⊥'

infix 30 _∨_
_∨_ : Fm → Fm → Fm
a ∨ b = ¬' a ↝ b

infixr 40 _∧_
_∧_ : Fm → Fm → Fm
a ∧ b = ¬' (a ↝ ¬' b)

infix 70 □_
□_ : Fm → Fm
□_ a = ¬' a ▷ ⊥'

infix 70 ◇_
◇_ : Fm → Fm
◇_ a = ¬' □ (¬' a)
```

```
infixr 15 _↔_
_↔_ : Fm → Fm → Fm
a ↔ b = (a ⤳ b) ∧ (b ⤳ a)

infix 50 _◁_
_◁_ : Fm → Fm → Fm
a ◁ b = ¬' (a ▷ ¬' b)
```

B.5. GeneralizedFrame/Properties

Properties of generalized Veltman frames.

```
module GeneralizedFrame.Properties where

open import Agda.Builtin.Nat using (Nat; suc; _+_)
open import Agda.Primitive using (Level; lzero; lsuc; _⊔_)
open import Data.Empty using (⊥; ⊥-elim)
open import Data.List using (List)
open import Data.List.Relation.Unary.All using (All)
open import Data.Product
open import Data.Sum using (_⊎_; inj₁; inj₂)
open import Relation.Nullary using (yes; no; ¬_)
open import Function using (_∘_; case_of_; _$_)
open import Relation.Binary using (REL; Rel; Transitive; Reflexive)
renaming (Decidable to Decidable₂)
open import Relation.Binary.PropositionalEquality using (_≡_; _≢_; refl;
subst; cong)
open import Relation.Nullary using (¬_)
open import Relation.Unary using (Pred; _∈_; _∉_; Decidable; {_}; _∩_;
_⊆_; Satisfiable)

open import Formula using (Fm; Var; _⤳_; ⊥'; _▷_; var; ¬'_)
open import Base using (Noetherian; REL₃)


import GeneralizedVeltmanSemantics as G
open import GeneralizedFrame

private
  variable
    ℓW ℓR ℓS : Level

module FrameProperties
    {W R S}
    (T : ∀ {ℓW ℓS} (W : Set ℓW) → REL₃ W W (Pred W ℓW) ℓS → Set (lsuc
ℓW ⊔ ℓS))
    (F : Frame {ℓW} {ℓR} {ℓS} W R S T)
  where
  open Frame F
  S⊆{v} : ∀ {w u v V} → V ⊆ { v } → S w u V → S w u { v }
  S⊆{v} V⊆v SwuV = S-ext SwuV V⊆v (λ { refl → case Swu-sat SwuV of
             λ {  (_ , snd) → case V⊆v snd of λ {refl → snd}}})

  S⊆{v}' : ∀ {w u k} → ∃[ V ] (V ⊆ { k } × S w u V) → S w u { k }
  S⊆{v}' (V , V⊆k , SwuV) = S⊆{v} V⊆k SwuV


module FrameLProperties
  {ℓW ℓR ℓS}
  {W R S}
  (F : FrameL {ℓW} {ℓR} {ℓS} W R S)
   where
   open FrameL F

   transUni : ∀ {b w x V} → S b w { x } → S b x V → S b w V
   transUni {b} {w} {x} Sbwx SbxV = case quasitrans Sbwx (λ {refl → _
, SbxV}) of
            λ {SbwU → S-ext SbwU (λ { (_ , refl , snd) → snd}) (λ {k →
_ , refl , k})}

   S-trans : ∀ {x w u} {V : 𝕎} → R x w → R w u → S x u V → S x w V
   S-trans {x} {w} {u} {V} Rxw Rwu = transUni (R-Sw-trans Rxw Rwu)


module Mono-closure
  (W : Set ℓW)
  (R : Rel W ℓR)
  (S : REL₃ W W (Pred W ℓW) ℓS) where
  private
    𝕎 = Pred W ℓW

  data S' : REL₃ W W 𝕎 (ℓR ⊔ lsuc ℓW ⊔ ℓS) where
    s' : ∀ {w u Y Y'} → S w u Y → Y ⊆ Y' → Y' ⊆ R w → S' w u Y'

  S'→S : ∀ {w u Y'} → S' w u Y' → Σ 𝕎 λ Y → S w u Y × Y ⊆ Y' × Y' ⊆ R w
  S'→S (s' {Y = Y} x x₁ x₂) = Y , x , x₁ , x₂

  S'→S[Y] : ∀ {w u} {Y' : 𝕎} → (Σ 𝕎 λ Y → S w u Y × Y ⊆ Y' × Y' ⊆ R
w) → 𝕎
  S'→S[Y] = proj₁

  S'→S[S] : ∀ {w u} {Y' : 𝕎} → (P : Σ 𝕎 λ Y → S w u Y × Y ⊆ Y' × Y' ⊆
R w) → let Y = proj₁ P in S w u Y
  S'→S[S] = proj₁ ∘ proj₂

  S'→S[⊆] : ∀ {w u} {Y' : 𝕎} → (P : Σ 𝕎 λ Y → S w u Y × Y ⊆ Y' × Y' ⊆
R w) → let Y = proj₁ P in Y ⊆ Y'
  S'→S[⊆] = proj₁ ∘ proj₂ ∘ proj₂

  S'→S[R] : ∀ {w u} {Y' : 𝕎} → (P : Σ 𝕎 λ Y → S w u Y × Y ⊆ Y' × Y' ⊆
R w) → Y' ⊆ R w
  S'→S[R] = proj₂ ∘ proj₂ ∘ proj₂

module Transitivity'
  {W : Set ℓW}
  {R : Rel W ℓR}
  {S : REL₃ W W (Pred W ℓW) ℓS}
  where
  𝕎 : Set _
  𝕎 = Pred W ℓW

  Frame-1 Frame-2 Frame-3 Frame-4 Frame-5 Frame-6 Frame-7 Frame-8
    : Set _
  Frame-1 = Frame W R S (Trans-conditions.Trans-1)
  Frame-2 = Frame W R S (Trans-conditions.Trans-2)
  Frame-3 = Frame W R S (Trans-conditions.Trans-3)
  Frame-4 = Frame W R S (Trans-conditions.Trans-4)
  Frame-5 = Frame W R S (Trans-conditions.Trans-5)
  Frame-6 = Frame W R S (Trans-conditions.Trans-6)
  Frame-7 = Frame W R S (Trans-conditions.Trans-7)
  Frame-8 = Frame W R S (Trans-conditions.Trans-8)

  open Mono-closure W R S

 Frame-1' Frame-2' Frame-3' Frame-4' Frame-5' Frame-6' Frame-7' Frame-
8'
    : Set _
  Frame-1' = Frame W R S' (Trans-conditions.Trans-1)
  Frame-2' = Frame W R S' (Trans-conditions.Trans-2)
  Frame-3' = Frame W R S' (Trans-conditions.Trans-3)
  Frame-4' = Frame W R S' (Trans-conditions.Trans-4)
  Frame-5' = Frame W R S' (Trans-conditions.Trans-5)
  Frame-6' = Frame W R S' (Trans-conditions.Trans-6)
  Frame-7' = Frame W R S' (Trans-conditions.Trans-7)
  Frame-8' = Frame W R S' (Trans-conditions.Trans-8)

  lemma : FrameNoTrans W R S → FrameNoTrans W R S'
  lemma (frame witness Swu-sat R-trans R-noetherian Sw⊆R[w]
    SwuY⊆Rw S-quasirefl R-Sw-trans S-ext) =
    frame witness
      (λ { (s' SwuY Y⊆Y' Y'⊆Rw) → case Swu-sat SwuY of λ { (fst , snd)
→ fst , (Y⊆Y' snd)}})
      R-trans
      R-noetherian
      (λ { (s' x x₁ x₂) → Sw⊆R[w] x})
      (λ { (s' x x₂ x₃) x₁ → x₃ x₁})
      (λ x → S⊆S' (S-quasirefl x))
      (λ {x x₁ → S⊆S' (R-Sw-trans x x₁)})
      λ { (Mono-closure.s' SwxY Y⊆V V⊆R) V⊆V' V'⊆V →
        Mono-closure.s' SwxY (V⊆V' ∘ Y⊆V) (V⊆R ∘ V'⊆V)}
      where
      S⊆S' : ∀ {w u Y} → S w u Y → S' w u Y
      S⊆S' {w} {u} {Y} x = s' x (λ z → z) (SwuY⊆Rw x)

  S'-⊆-Irrel : Set (lsuc ℓW ⊔ ℓR ⊔ ℓS)
  S'-⊆-Irrel = ∀ {x u Y'} (SxuY' : S' x u Y') →
    let Y = S'→S[Y] (S'→S SxuY')
    in Σ (Y ⊆ Y') λ Y⊆Y' → ∀ {y} {y∈Y y∈'Y : y ∈ Y} → Y⊆Y' y∈Y ≡ Y⊆Y'
y∈'Y

  1⇒2' : S'-⊆-Irrel → Frame-1 → Frame-2'
  1⇒2' ⊆-irrel (frame frame-0 trans) = frame (lemma frame-0)
    λ { a@(s' {x} {u} {Y} {Y'} SxuY Y⊆ Y⊆Rx) y'→Y' irrel SxyY'y →
    case ⊆-irrel a of λ { (Y⊆Y' , irrel') →
     case trans SxuY
      (λ y∈Y → S'→S[Y] (S'→S (SxyY'y (Y⊆Y' y∈Y))))
      (cong (S'→S[Y] ∘ S'→S ∘ SxyY'y) irrel')
      (λ y∈Y → S'→S[S] (S'→S (SxyY'y (Y⊆Y' y∈Y)))) of
     λ { (Z , Z⊆ , SxuZ) → s' SxuZ
        (λ { z  → case Z⊆ z of
        λ { ((v , v∈Y) , snd) →
           (v , (Y⊆Y' v∈Y)) , (S'→S[⊆] (S'→S (SxyY'y (Y⊆Y' v∈Y)))) snd
}})
        λ { {v} ((y , y∈Y') , v∈Y_y) → (S'→S[R] (S'→S (SxyY'y y∈Y')))
v∈Y_y}}}}

  2⇒2' : S'-⊆-Irrel → Frame-2 → Frame-2'
  2⇒2' ⊆-irrel (frame frame-0 trans) = frame (lemma frame-0)
    λ { a@(s' {x} {u} {Y} {Y'} SxuY Y⊆ _) y'→Y' irrel SxyY'y →
    case ⊆-irrel a of λ { (Y⊆Y' , irrel') →
     case trans SxuY
      (λ y∈Y → S'→S[Y] ∘ S'→S ∘ SxyY'y ∘ Y⊆Y' $ y∈Y)
      (cong (S'→S[Y] ∘ S'→S ∘ SxyY'y) irrel')
      (λ y∈Y → S'→S[S] (S'→S (SxyY'y (Y⊆Y' y∈Y)))) of
     λ {SxuU → s' SxuU
        (λ { ((v , v∈Y) , snd) →
          (v , (Y⊆Y' v∈Y)) , (S'→S[⊆] ∘ S'→S ∘ SxyY'y ∘ Y⊆Y' $ v∈Y) snd
})
        λ { {v} ((y , y∈Y') , v∈Yy) → (S'→S[R] ∘ S'→S ∘ SxyY'y $ y∈Y')
v∈Yy}}}}

  3⇒2' : Frame-3 → Frame-2'
  3⇒2' (frame frame-0 trans) = frame (lemma frame-0)
   λ { (s' {x} {u} {Y} SxuY Y⊆Y' _) y'→Y irrel y'→S' → case trans SxuY
of
    λ { (y₀ , y₀∈Y , snd) → case snd (S'→S[S] ∘ S'→S ∘ y'→S' ∘ Y⊆Y' $
y₀∈Y) of
    λ { (Z , Z⊆ , SxuZ) → s' SxuZ
      (λ { {v} v∈Yy → (y₀ , Y⊆Y' y₀∈Y) , (S'→S[⊆] ∘ S'→S ∘ y'→S' ∘ Y⊆Y'
$ y₀∈Y) (Z⊆ v∈Yy)})
   λ { {v} ((y , y∈Y') , snd) → (S'→S[R] ∘ S'→S ∘ y'→S' $ y∈Y') snd}}}}
  4⇒2' : Frame-4 → Frame-2'
  4⇒2' (frame frame-0 trans) = frame (lemma frame-0)
    λ { (s' {x} {u} {Y} SxuY Y⊆Y' _) y'→Y irrel y'→S' → case trans SxuY
of
     λ { (y₀ , y₀∈Y , snd) → s' (snd (S'→S[S] ∘ S'→S ∘ y'→S' ∘ Y⊆Y' $
y₀∈Y))
     (λ { {v} v∈Yy → (y₀ , Y⊆Y' y₀∈Y) , (S'→S[⊆] ∘ S'→S ∘ y'→S' ∘ Y⊆Y'
$ y₀∈Y) v∈Yy})
    λ { {v} ((y , y∈Y') , snd) → (S'→S[R] ∘ S'→S ∘ y'→S' $ y∈Y') snd}}}

  5⇒2' : Frame-5 → Frame-2'
  5⇒2' (frame frame-0 trans) = frame (lemma frame-0)
    λ { (s' {x} {u} {Y} SxuY Y⊆Y' _) y'→Y irrel y'→S' →
    case Swu-sat SxuY of λ { (y₀ , y₀∈Y) →
    case trans SxuY y₀∈Y (S'→S[S] ∘ S'→S ∘ y'→S' ∘ Y⊆Y' $ y₀∈Y) of
    λ { (Z , Z⊆SxuYy₀ , SxuYy₀) → s' SxuYy₀
     (λ {∈Z → case Z⊆SxuYy₀ ∈Z of
    λ {v∈Yy → (y₀ , Y⊆Y' y₀∈Y) , (S'→S[⊆] ∘ S'→S ∘ y'→S' ∘ Y⊆Y' $ y₀∈Y)
v∈Yy}})
    λ { {v} ((y , y∈Y') , snd) → (S'→S[R] ∘ S'→S ∘ y'→S' $ y∈Y') snd}}}}
    where open FrameNoTrans frame-0

  6⇒2' : Frame-6 → Frame-2'
  6⇒2' (frame frame-0 trans) = frame (lemma frame-0)
    λ { (s' {x} {u} {Y} SxuY Y⊆Y' _) y'→Y irrel y'→S' →
    case Swu-sat SxuY of λ { (y₀ , y₀∈Y) →
    case trans SxuY y₀∈Y (S'→S[S] ∘ S'→S ∘ y'→S' ∘ Y⊆Y' $ y₀∈Y) of
    λ { SxuYy₀ → s' SxuYy₀
    ((λ { {v} v∈Yy → (y₀ , Y⊆Y' y₀∈Y) , (S'→S[⊆] ∘ S'→S ∘ y'→S' ∘ Y⊆Y'
$ y₀∈Y) v∈Yy}))
    λ { {v} ((y , y∈Y') , snd) → (S'→S[R] ∘ S'→S ∘ y'→S' $ y∈Y') snd}}}}
    where open FrameNoTrans frame-0

  dec-⊆ : Set _
  dec-⊆ = (X Y : 𝕎) → X ⊆ Y ⊎ (Σ W λ x → x ∈ X × x ∉ Y)

  7⇒2' : dec-⊆ → Frame-7 → Frame-2'
  7⇒2' ⊆? (frame frame-0 trans) = frame (lemma frame-0)
    λ { a@(s' {x} {u} {Y} {Y'} SxuY Y⊆Y' _) y'→Y irrel y'→S' →
     case ⊆? Y (⋃[ Σ W (_∈ Y') ] (y'→Y ∘ proj₂)) of
     λ { (inj₁ Y⊆U) → s' SxuY Y⊆U
     λ { {v} ((y , y∈Y') , snd) → (S'→S[R] ∘ S'→S ∘ y'→S' $ y∈Y') snd}
     ; (inj₂ (y₀ , y₀∈Y , ∉U)) →
        let r = S'→S ∘ y'→S' ∘ Y⊆Y' $ y₀∈Y in
        case trans SxuY y₀∈Y (S'→S[S] r)
         (λ { y₀∈Yy₀- → ∉U ((y₀ , Y⊆Y' y₀∈Y) , S'→S[⊆] r y₀∈Yy₀-)}) of
         λ { (Z , Z⊆ , SxuYy₀) → s' SxuYy₀ (λ ∈Y'- → (y₀ , Y⊆Y' y₀∈Y)
, S'→S[⊆] r (Z⊆ ∈Y'-))
         (λ { {v} ((y , y∈Y') , snd) → (S'→S[R] ∘ S'→S ∘ y'→S' $ y∈Y')
snd})} }}
```

```
  8⇒2' : dec-⊆ → Frame-8 → Frame-2'
  8⇒2' ⊆? (frame frame-0 trans) = frame (lemma frame-0)
    λ { a@(s' {x} {u} {Y} {Y'} SxuY Y⊆Y' _) y'→Y irrel y'→S' →
     case ⊆? Y (⋃[ Σ W (_∈ Y') ] (y'→Y ∘ proj₂)) of
     λ { (inj₁ Y⊆U) → s' SxuY Y⊆U
     λ { {v} ((y , y∈Y') , snd) → (S'→S[R] ∘ S'→S ∘ y'→S' $ y∈Y') snd}
     ; (inj₂ (y₀ , y₀∈Y , ∉U)) →
        let r = S'→S ∘ y'→S' ∘ Y⊆Y' $ y₀∈Y in
        case trans SxuY y₀∈Y (S'→S[S] r)
          (λ { y₀∈Yy₀- → ∉U ((y₀ , Y⊆Y' y₀∈Y) , S'→S[⊆] r y₀∈Yy₀-)}) of
           λ { SxuYy₀ → s' SxuYy₀ (λ ∈Y'- → (y₀ , Y⊆Y' y₀∈Y) , S'→S[⊆]
r ∈Y'-)
          (λ { {v} ((y , y∈Y') , snd) → (S'→S[R] ∘ S'→S ∘ y'→S' $ y∈Y')
snd})}}}

module Transitivity {W} {R} {S}
  (F : FrameNoTrans {ℓW} {ℓR} {ℓS} W R S)
  where
  open FrameNoTrans F

  Monotone : Set _
  Monotone = ∀ {w u} {V Z : 𝕎} → S w u V → V ⊆ Z → Z ⊆ R w → S w u Z

  R-Decidable : Set _
  R-Decidable = Decidable₂ R

  S₃-Decidable : Set _
  S₃-Decidable = ∀ {w u Y} → S w u Y → Decidable Y

  open Trans-conditions W S

  ---------------

  1⇒2 : Monotone → Trans-1 → Trans-2
  --
  2⇒1 : Trans-2 → Trans-1
  --
  3⇒4 : Monotone → Trans-3 → Trans-4
  --
  4⇒3 : Trans-4 → Trans-3
  --
  5⇒1 : Trans-5 → Trans-1
  5⇒2 : Monotone → Trans-5 → Trans-2
  5⇒3 : Trans-5 → Trans-3
  5⇒4 : Monotone → Trans-5 → Trans-4
  5⇒6 : Monotone → Trans-5 → Trans-6
  5⇒7 : Trans-5 → Trans-7
  5⇒8 : Monotone → Trans-5 → Trans-8
  --
  6⇒1 : Trans-6 → Trans-1
  6⇒2 : Monotone → Trans-6 → Trans-2
  6⇒3 : Trans-6 → Trans-3
  6⇒4 : Trans-6 → Trans-4
  6⇒5 : Trans-6 → Trans-5
  6⇒7 : Trans-6 → Trans-7
  6⇒8 : Trans-6 → Trans-8
  --
  7⇒8 : Monotone → Trans-7 → Trans-8
  --
  8⇒7 : Trans-8 → Trans-7

  ---------------

  1⇒2 mono t SxuY y→Y irrel p = case t SxuY y→Y irrel p of
    λ { (Z , Z⊆ , SxuZ) → mono SxuZ Z⊆
    λ { {v} ((y , y∈Y) , snd) → SwuY⊆Rw (p y∈Y) snd}}

  2⇒1 t {Y = Y} SxuY y→Y irrel p = ⋃[ Σ W (_∈ Y) ] (y→Y ∘ proj₂) , (λ
z → z) , t SxuY y→Y irrel p

  3⇒4 mono t SxuY = case t SxuY of
    λ { (y , y∈Y , snd) → y , y∈Y , λ {SxyY' → case snd SxyY' of
    λ { (Y'' , Y''⊆Y' , SxuY'') → mono SxuY'' Y''⊆Y' (λ { {v} v∈
    → SwuY⊆Rw SxyY' v∈})}}}

  4⇒3 t SxuY = case t SxuY of λ { (y , fst₁ , snd) → y , fst₁ ,
    λ { {Y'} SxyY' → Y' , (λ x → x) , snd SxyY'}}

  -- needs irrel
  -- 2⇒2L : Trans-2 → Trans-L
  -- 2⇒2L T2 SxuY f = S-ext (T2 SxuY (λ { {y} y∈Y → proj₁ (f y∈Y)})
  --    (λ { {y} {a} {b} → {!!} }) λ {y∈Y → proj₂ (f y∈Y)})
  --    (λ { ((x , x∈V) , snd) → _ , x∈V , snd})
  --    λ { (y , y∈Y , snd) → (y , y∈Y) , snd}

  2L⇒2 : Trans-L → Trans-2
  2L⇒2 TL SxuY y→Y irrel y→SYy = S-ext
    (TL SxuY λ {v∈Y → _ , y→SYy v∈Y})
    (λ { (y , y∈Y , snd) → (y , y∈Y) , snd})
    λ { ((y , y∈Y) , snd) → _ , y∈Y , snd}

  5⇒1 t SxuY y→Y irrel p = case Swu-sat SxuY of λ { (y , y∈Y)
    → case t SxuY y∈Y (p y∈Y) of λ { (Y' , Y'⊆Yy , SxuY') → Y' ,
    (λ {x → (_ , y∈Y) , Y'⊆Yy x}) , SxuY'}}

  5⇒2 mono t SxuY y→Y irrel p = case Swu-sat SxuY of λ { (y , y∈Y)
    → case t SxuY y∈Y (p y∈Y) of λ { (Y' , Y'⊆Yy , SxuY') →
    mono SxuY' (λ x₁ → (y , y∈Y) , Y'⊆Yy x₁)
    (λ { {v} ((y' , y'∈Y) , snd) → SwuY⊆Rw (p y'∈Y) snd})}}

  5⇒3 t SxuY = case Swu-sat SxuY of λ { (y , y∈Y) → y , y∈Y ,
    λ {SxyY' → t SxuY y∈Y SxyY'}}
  5⇒4 mono t SxuY = case Swu-sat SxuY of λ { (y , y∈Y) → y , y∈Y ,
    λ {SxyY' → case t SxuY y∈Y SxyY' of λ { (Y'' , Y''⊆Y' , SxuY'') →
    mono SxuY'' Y''⊆Y' (λ x₁ → SwuY⊆Rw SxyY' x₁)}}}

  5⇒6 mono t SxuY y∈Y SxyY' = case t SxuY y∈Y SxyY' of
     λ { (Y'' , Y''⊆Y' , SxuY'') → mono SxuY'' Y''⊆Y' λ { {v} v∈ →
SwuY⊆Rw SxyY' v∈}}

  5⇒7 t SxuY y∈Y SxyY' y∉Y' = t SxuY y∈Y SxyY'

  5⇒8 mono t = 6⇒8 (5⇒6 mono t)

  6⇒1 t = 5⇒1 (6⇒5 t)

  6⇒2 mono t = 1⇒2 mono (6⇒1 t)

  6⇒3 t = 4⇒3 (6⇒4 t)

  6⇒4 t SxuY = case Swu-sat SxuY of
    λ { (y , y∈Y) → y , y∈Y , λ {SxyY' → t SxuY y∈Y SxyY'}}

  6⇒5 t {Y' = Y'} SxuY y∈Y SxyY' = Y' , (λ z → z) , t SxuY y∈Y SxyY'

  6⇒8 t SxuY y∈Y SxyY' y∉Y' = t SxuY y∈Y SxyY'

  6⇒7 x = 8⇒7 (6⇒8 x)

  7⇒8 mono t7 SxuY y∈Y SxyY' y∉Y' = case t7 SxuY y∈Y SxyY' y∉Y'
    of λ { (Y'' , Y''⊆Y' , snd) → mono snd Y''⊆Y' (λ {v} v∈Y' → SwuY⊆Rw
SxyY' v∈Y')}

  8⇒7 t8 {Y' = Y'} SxuY y∈Y SxyY' y∉Y' = Y' , ((λ x₆ → x₆) , t8 SxuY
y∈Y SxyY' y∉Y')
```

B.6. GeneralizedFrame

```
module GeneralizedFrame where

open import Agda.Builtin.Nat using (Nat; suc; _+_)
open import Agda.Primitive using (Level; lzero; lsuc; _⊔_)
open import Data.Empty using (⊥; ⊥-elim)
open import Data.List using (List)
open import Data.List.Relation.Unary.All using (All)
open import Data.Product
open import Data.Sum using (_⊎_; inj₁; inj₂)
open import Relation.Nullary using (yes; no; ¬_)
open import Function using (_∘_; case_of_; _$_)
open import Relation.Binary using (REL; Rel; Transitive; Reflexive)
renaming (Decidable to Decidable₂)
open import Relation.Binary.PropositionalEquality using (_≡_; _≢_; refl;
subst; cong)
open import Relation.Nullary using (¬_)
open import Relation.Unary using (Pred; _∈_; _∉_; Decidable; {_}; _∩_;
_⊆_; Satisfiable)

open import Formula using (Fm; Var; _⥽_; ⊥'; _▷_; var; ¬'_)
open import Base

private
  variable
    ℓW ℓR ℓS : Level

⋃[_] : ∀ {ℓ ℓ' ℓ''} {B : Set ℓ'} → (A : Set ℓ) → (A → Pred B ℓ'') →
Pred B _
⋃[ y∈Y ] y→Y z = ∃[ y∈Y ] (z ∈ (y→Y y∈Y))

module Trans-conditions (W : Set ℓW) (S : REL₃ W W (Pred W ℓW) ℓS)
where
  private
    𝕎 = Pred W ℓW

   Trans-1 Trans-2 Trans-3 Trans-4 Trans-5 Trans-6 Trans-7 Trans-8
Trans-L
    : Set (lsuc ℓW ⊔ ℓS)

  Trans-1 = ∀ {u x Y} → S x u Y → (y→Y : ∀ {y} → y ∈ Y → 𝕎)
    → (irrel : ∀ {y} {y∈Y y∈'Y : y ∈ Y} → y→Y y∈Y ≡ y→Y y∈'Y)
    → (∀ {y} (y∈Y : y ∈ Y) → S x y (y→Y y∈Y))
    → ∃[ Z ] (Z ⊆ ⋃[ Σ W (_∈ Y) ] (y→Y ∘ proj₂) × S x u Z)
  Trans-2 = ∀ {u x Y} → S x u Y → (y→Y : ∀ {y} → y ∈ Y → 𝕎)
    → (irrel : ∀ {y} {y∈Y y∈'Y : y ∈ Y} → y→Y y∈Y ≡ y→Y y∈'Y)
    → (∀ {y} (y∈Y : y ∈ Y) → S x y (y→Y y∈Y))
    → S x u (⋃[ Σ W (_∈ Y) ] (y→Y ∘ proj₂))
  Trans-3 = ∀ {x u Y} → S x u Y → ∃[ y ] (Σ[ y∈Y ∈ (y ∈ Y) ] (∀ {Y'} →
S x y Y' → ∃[ Y'' ]
    (Y'' ⊆ Y' × S x u Y'')))
  Trans-4 = ∀ {x u Y} → S x u Y → ∃[ y ] (y ∈ Y × (∀ {Y'} → S x y Y' →
S x u Y'))
  Trans-5 = ∀ {u x y Y Y'} → S x u Y → y ∈ Y → S x y Y' → ∃[ Y'' ] (Y''
⊆ Y' × S x u Y'')
  Trans-6 = ∀ {u x y Y Y'} → S x u Y → y ∈ Y → S x y Y' → S x u Y'
  Trans-7 = ∀ {u x y Y Y'} → S x u Y → y ∈ Y → S x y Y' → y ∉ Y'
    → ∃[ Y'' ] (Y'' ⊆ Y' × S x u Y'')
  Trans-8 = ∀ {u x y Y Y'} → S x u Y → y ∈ Y → S x y Y' → y ∉ Y' → S x
u Y'
  Trans-L1 = ∀ {w u V} → S w u V → (f : ∀ {v} → v ∈ V → ∃[ Z ] (S w v
Z))
      → ∃[ Z ] (Z ⊆ (λ { x → ∃[ v ] (Σ[ v∈V ∈ (v ∈ V) ] (x ∈ proj₁ (f
v∈V)))}) × S w u Z)
  Trans-L = ∀ {w u V} → S w u V → (f : ∀ {v} → v ∈ V → ∃[ Z ] (S w v Z))
      → S w u λ {x → ∃[ v ] (Σ[ v∈V ∈ (v ∈ V) ] (x ∈ proj₁ (f v∈V)))}
```

```
record FrameNoTrans (W : Set ℓW) (R : Rel W ℓR) (S : REL₃ W W (Pred W
ℓW) ℓS)
  : Set (lsuc lzero ⊔ ℓR ⊔ ℓS ⊔ lsuc ℓW) where
  constructor frame
  𝕎 : Set _
  𝕎 = Pred W ℓW
  field
    witness : W
    Swu-sat : ∀ {w u Y} → S w u Y → Satisfiable Y
    R-trans : Transitive R
    R-noetherian : Noetherian R
    Sw⊆Rw : ∀ {w u Y} → S w u Y → R w u
    SwuY⊆Rw : ∀ {w u Y} → S w u Y → ∀ {y} → y ∈ Y → R w y
    S-quasirefl : ∀ {w u} → R w u → S w u ｛ u ｝
    R-Sw-trans : ∀ {w u v} → R w u → R u v → S w u ｛ v ｝
    S-ext : ∀ {w x V V'} → S w x V → V ⊆ V' → V' ⊆ V → S w x V'

  IsChoiceSet : 𝕎 → (w x : W) → Set (lsuc ℓW ⊔ ℓR ⊔ ℓS)
  IsChoiceSet Γ w x = R w x × ∀ {Y} → S w x Y → Satisfiable (Y ∩ Γ)

  ChoiceSet : (w x : W) → Set (lsuc ℓW ⊔ ℓR ⊔ ℓS)
  ChoiceSet w x = Σ 𝕎 λ Γ → IsChoiceSet Γ w x

record Frame (W : Set ℓW) (R : Rel W ℓR) (S : REL₃ W W (Pred W ℓW) ℓS)
  (T : (W : Set ℓW) → REL₃ W W (Pred W ℓW) ℓS → Set (lsuc ℓW ⊔ ℓS))
  : Set (lsuc ℓW ⊔ ℓR ⊔ ℓS) where
  constructor frame
  field
    frame-0 : FrameNoTrans {ℓW} {ℓR} {ℓS} W R S
    quasitrans : T W S
  open FrameNoTrans frame-0 public

FrameL : (W : Set ℓW) (R : Rel W ℓR) (S : REL₃ W W (Pred W ℓW) ℓS) →
Set _
FrameL W R S = Frame W R S (Trans-conditions.Trans-L)

Frame1 : (W : Set ℓW) (R : Rel W ℓR) (S : REL₃ W W (Pred W ℓW) ℓS) →
Set _
Frame1 W R S = Frame W R S (Trans-conditions.Trans-1)

Frame2 : (W : Set ℓW) (R : Rel W ℓR) (S : REL₃ W W (Pred W ℓW) ℓS) →
Set _
Frame2 W R S = Frame W R S (Trans-conditions.Trans-2)

Frame3 : (W : Set ℓW) (R : Rel W ℓR) (S : REL₃ W W (Pred W ℓW) ℓS) →
Set _
Frame3 W R S = Frame W R S (Trans-conditions.Trans-3)

Frame4 : (W : Set ℓW) (R : Rel W ℓR) (S : REL₃ W W (Pred W ℓW) ℓS) →
Set _
```

```
Frame4 W R S = Frame W R S (Trans-conditions.Trans-4)

Frame5 : (W : Set ℓW) (R : Rel W ℓR) (S : REL₃ W W (Pred W ℓW) ℓS) →
Set _
Frame5 W R S = Frame W R S (Trans-conditions.Trans-5)

Frame6 : (W : Set ℓW) (R : Rel W ℓR) (S : REL₃ W W (Pred W ℓW) ℓS) →
Set _
Frame6 W R S = Frame W R S (Trans-conditions.Trans-6)

Frame7 : (W : Set ℓW) (R : Rel W ℓR) (S : REL₃ W W (Pred W ℓW) ℓS) →
Set _
Frame7 W R S = Frame W R S (Trans-conditions.Trans-7)

Frame8 : (W : Set ℓW) (R : Rel W ℓR) (S : REL₃ W W (Pred W ℓW) ℓS) →
Set _
Frame8 W R S = Frame W R S (Trans-conditions.Trans-8)

module FrameL
  {W R S}
  (F : FrameL {ℓW} {ℓR} {ℓS} W R S) where
  open Frame F public

module Frame1
  {W R S}
  (F : Frame1 {ℓW} {ℓR} {ℓS} W R S) where
  open Frame F public

module Frame2
  {W R S}
  (F : Frame2 {ℓW} {ℓR} {ℓS} W R S) where
  open Frame F public

module Frame3
  {W R S}
  (F : Frame3 {ℓW} {ℓR} {ℓS} W R S) where
  open Frame F public

module Frame4
  {W R S}
  (F : Frame4 {ℓW} {ℓR} {ℓS} W R S) where
  open Frame F public

module Frame5
  {W R S}
  (F : Frame5 {ℓW} {ℓR} {ℓS} W R S) where
  open Frame F public

module Frame6
  {W R S}
  (F : Frame6 {ℓW} {ℓR} {ℓS} W R S) where
  open Frame F public

module Frame7
  {W R S}
  (F : Frame7 {ℓW} {ℓR} {ℓS} W R S) where
  open Frame F public

module Frame8
  {W R S}
  (F : Frame8 {ℓW} {ℓR} {ℓS} W R S) where
  open Frame F public

module Predicates
  {W R S}
  (F : FrameL {lzero} {lzero} {lzero} W R S)
  where
  open FrameL F

  R[_] : 𝕎 → 𝕎
  R[ V ] x = ∃[ v ] (v ∈ V × R v x)

  R⁻¹[_] : 𝕎 → 𝕎
  R⁻¹[ E ] x = ∃[ e ] (e ∈ E × R x e)

  R⁻¹_[_] : W → 𝕎 → 𝕎
  R⁻¹ x [ V ] = R⁻¹[ V ] ∩ R x
```

B.7. GeneralizedVeltmanSemantics/Properties/GenericFrameCond

It contains the proof that we can find a frame condition for any principle.

```
module GeneralizedVeltmanSemantics.Properties.GenericFrameCond where

open import Function.Equivalence using (_⇔_; equivalence; module Equivalence)

open import Agda.Builtin.Nat using (Nat; suc; zero)
open import Agda.Builtin.Unit using (⊤; tt)
open import Agda.Primitive using (Level; lzero; lsuc; _⊔_)
open import Data.Empty using (⊥; ⊥-elim)
open import Data.Product using (Σ; proj₁; proj₂; _×_) renaming (_,_ to
_,_)
open import Data.Sum using (_⊎_; inj₁; inj₂; [_,_])
open import Function using (_∘_; const; case_of_; id)
open import Function.Equality using (_⟨$⟩_)
open import Relation.Binary using (REL; Rel; Transitive)
open import Relation.Nullary using (yes; no; ¬_)
open import Relation.Unary using (Pred; _∈_; _∉_; Decidable; Satisfiable;
_⊆_; _∩_; {_}; ∅)
open import Relation.Binary using (Irreflexive) renaming (Decidable to
Decidable₂)
open import Relation.Binary.PropositionalEquality using (_≡_; refl;
subst; trans; sym)
```

```
open import Formula using (Fm; Var; _↝_; ⊥'; _▷_; _◁_; var; ⊤'; ¬'_;
□_; ◊_; _∧_; _∨_; car)
open import GeneralizedVeltmanSemantics
open import Base using (_⇒_; _⇐_; Decidable₃; Rel₃; REL₃)
open import GeneralizedVeltmanSemantics.Properties
  using (module Extended; ⊩¬; ⊮¬; ⊩□; ⊩◊; ⊩MP; ⊩∧; ⊩→¬⊮)
import Principles as P

private
  variable
    ℓW ℓR ℓS : Level
    W : Set ℓW
    R : Rel W ℓR
    S : REL₃ W W (Pred W ℓW) ℓS
    V : REL W Var lzero

module Condition
  (F : FrameL {lzero} {lzero} {lzero} W R S)
  where
  open FrameL F

  lift : Set → Set₁
  lift Sa = Sa × (⊥ → Set)

  Tagger : Set₁
  Tagger = Var → 𝕎

  fm2pred : Tagger → Fm → W → Set₁
  fm2pred ⟦_⟧ (var v) u = lift (u ∈ ⟦ v ⟧)
  fm2pred _ ⊥' w = lift ⊥
  fm2pred ⟦_⟧ (A ↝ B) w = w ∈ fm2pred ⟦_⟧ A → w ∈ fm2pred ⟦_⟧ B
  fm2pred ⟦_⟧ (A ▷ B) w = ∀ {u} → R w u → u ∈ fm2pred ⟦_⟧ A
    → Σ 𝕎 λ Y → S w u Y × Y ⊆ fm2pred ⟦_⟧ B

  Frame-cond : Fm → Set₁
  Frame-cond A = ∀ g w → fm2pred g A w

module Truth-lemma
  {M : Model {lzero} {lzero} {lzero} W R S V}
  (M,_*⊩?_ : MultiDecidableModel M)
  (∈S? : Decidable₃ S)
  (∈SV? : ∀ {w u Y} → S w u Y → Decidable Y) where

  open Model M
  open FrameL F
  open Extended M,_*⊩?_ ∈S? ∈SV?
  open Condition F

  lemma : ∀ {⟦_⟧ w A} → (q : ∀ {w v} → M , w ⊩ var v ⇔ w ∈ ⟦ v ⟧) →
    M , w ⊩ A ⇔ w ∈ fm2pred ⟦_⟧ A
  lemma {g} {w} {A} q = equivalence ⇨ ⇦
    where
    ⇨ : ∀ {w A} → M , w ⊩ A → w ∈ fm2pred g A
    ⇦ : ∀ {w A} → w ∈ fm2pred g A → M , w ⊩ A
    ⇨ v@(var x) = q ⇒ v , λ ()
    ⇨ i@(impl _) u = ⇨ ((⊩↝ ⇒ i) (⇦ u))
    ⇨ {w} r@(rhd _) Rwu u∈[A] = case (⊩▷ ⇒ r) Rwu (⇦ u∈[A]) of
      λ { (Z , SZ , snd) → Z , SZ , λ {v → ⇨ (snd v)}}
    ⇦ {w} {var x} (fst , _) = q ⇐ fst
    ⇦ {w} {A ↝ B} p = ⊩↝ ⇐ λ {x → ⇦ (p (⇨ x))}
    ⇦ {w} {A ▷ B} p = ⊩▷ ⇐ λ {Rwu uA → case p Rwu (⇨ uA) of
      λ { (Z , ZS , snd) → Z , ZS , λ {x → ⇦ (snd x)}}}


module soundness
  {M : Model {lzero} {lzero} {lzero} W R S V}
  (M,_*⊩?_ : MultiDecidableModel M)
  (∈S? : Decidable₃ S)
  (∈SV? : ∀ {w u Y} → S w u Y → Decidable Y) where

  open Model M
  open FrameL F
  open Extended M,_*⊩?_ ∈S? ∈SV?
  open Condition F
  open Truth-lemma M,_*⊩?_ ∈S? ∈SV?

  sound : ∀ w P → Frame-cond P → M , w ⊩ P
  sound w A cond = lemma g-aux ⇐ cond ⟦_⟧ w
    where
    ⟦_⟧ : Tagger
    ⟦ x ⟧ = [⊩ var x ]
    g-aux : ∀ {w v} → M , w ⊩ var v ⇔ w ∈ [⊩ var v ]
    g-aux {w} {v} = equivalence ⇨ ⇦
      where
      ⇨ : M , w ⊩ var v → w ∈ [⊩ var v ]
      ⇨ x = ∈[⊩ var v ] ⇐ x
      ⇦ : w ∈ [⊩ var v ] → M , w ⊩ var v
      ⇦ x = ∈[⊩ var v ] ⇒ x


module P⇒ℱ
  {F : FrameL {lzero} {lzero} {lzero} W R S}
  (∈S? : Decidable₃ S)
  (dec : ∀ V → MultiDecidableModel (model {V = V} F))
  (∈SV? : ∀ {w u Y} → S w u Y → Decidable Y)
  where
  open FrameL F
  open Condition F

  thm : ∀ P → F *⊩ P → Frame-cond P
  thm P x g w = lemma g-aux ⇒ w⊩P
```

```
    where
    Val : Valuation F
    Val u a = u ∈ g a
    M = model {V = Val} F
    open Extended (dec Val) ∈S? ∈SV?
    open Truth-lemma (dec Val) ∈S? ∈SV?
    w⊩P : M , w ⊩ P
    w⊩P = x (λ a b → g b a) w
    g-aux : ∀ {w v} → M , w ⊩ var v ⇔ w ∈ g v
    g-aux {w} {v} = equivalence (λ { (var x) → x}) var
```

B.8. GeneralizedVeltmanSemantics/Properties/Luka

It contains the proof that we build an ordinary frame from a generalized model.

```
module GeneralizedVeltmanSemantics.Properties.Luka where

open import Function.Equivalence using (_⇔_; equivalence; module Equivalence)

open import Agda.Builtin.Nat using (Nat; suc; zero)
open import Agda.Builtin.Unit using (⊤; tt)
open import Agda.Primitive using (Level; lzero; lsuc; _⊔_)
open import Data.Empty using (⊥; ⊥-elim)
open import Function using (_$_)
open import Data.Product renaming (_,_ to _,_)
open import Data.Sum using (_⊎_; inj₁; inj₂; [_,_])
open import Function using (_∘_; const; case_of_; id)
open import Function.Equality using (_⟨$⟩_)
open import Relation.Binary using (REL; Rel; Transitive)
open import Relation.Nullary using (yes; no; ¬_)
open import Relation.Unary using (Pred; _∈_; _∉_; Decidable; Satisfiable;
_⊆_; _∩_; {_}; ∅)
open import Relation.Binary using (Irreflexive) renaming (Decidable to
Decidable₂)
open import Relation.Binary.PropositionalEquality

open import Formula using (Fm; Var; _↣_; ⊥'; _▷_; _◁_; var; ⊤'; ¬'_;
□_; ◇_; _∧_; _∨_; car)
open import Base
open import GeneralizedVeltmanSemantics.Properties
  using (module SemanticsProperties-4;
    module SemanticsProperties-3;
    module SemanticsProperties-L; module PGeneric)
open import GeneralizedFrame
open Trans-conditions
open import GeneralizedFrame.Properties
import OrdinaryFrame as O
import OrdinaryVeltmanSemantics as O
import OrdinaryVeltmanSemantics.Properties as O

private
  variable
```

```
    ℓW ℓR ℓS : Level

-- I think an easier transformation would be (W, R, S) |-> (W, R, {
(w, u, v) : for some V, S w u V, and v in V })

module OrdModel
  {ℓW ℓR ℓS}
  (T : ∀ {ℓW ℓS} → (W : Set ℓW) → REL₃ W W (Pred W ℓW) ℓS → Set (lsuc
ℓW ⊔ ℓS))
  {W R S}
  (F : Frame {ℓW} {ℓR} {ℓS} W R S T)
  (T3 : Trans-3 W S)
  (V : W → Pred Var lzero)
  where
  open Frame F
  open PGeneric T
  open FrameProperties T F

  W' : Set _
  W' = W

  R' : Rel W' _
  R' = R

  S' : Rel₃ W' _
  S' w x v = ∃[ V ] (Σ[ SwxV ∈ S w x V ] (v ≡ proj₁ (T3 SwxV)))

  V' : W' → Pred Var lzero
  V' = V

  f-chain : ∀ {a} → InfiniteChain R' a → InfiniteChain R _
  InfiniteChain.b (f-chain x) = InfiniteChain.b x
  InfiniteChain.a<b (f-chain x) = InfiniteChain.a<b x
  InfiniteChain.tail (f-chain x) = f-chain (InfiniteChain.tail x)

  R'-Noetherian : Noetherian R'
  R'-Noetherian i = R-noetherian (f-chain i)

  F' : O.Frame W' R' S'
  F' = O.frame
    witness
    R-trans
    R'-Noetherian
     (λ { {w} {u} {v} (V , SwuV , refl) → Sw⊆Rw SwuV , SwuY⊆Rw SwuV
(proj₁ ∘ proj₂ $ T3 SwuV)})
    (λ { Rwu → _ , S-quasirefl Rwu , (proj₁ ∘ proj₂ $ T3 (S-quasirefl
Rwu))})
    (λ { {w} {i} (V , SwiV , refl) (V' , SwjV' , refl) →
      let
        j = proj₁ (T3 SwiV)
        k = proj₁ (T3 SwjV')
        Rwk : R w k
        Rwk = SwuY⊆Rw SwjV' (proj₁ ∘ proj₂ $ T3 SwjV')
        Swkk : S w k { k }
        Swkk = S-quasirefl Rwk
        Swik : S w i { k }
         Swik = S⊆{v}' $ (proj₂ ∘ proj₂ $ T3 SwiV) (S⊆{v}' $ (proj₂ ∘
proj₂ $ T3 SwjV') Swkk)
      in _ , Swik , (proj₁ ∘ proj₂ $ T3 Swik)
    })
   λ { Rwu Ruv → _ , R-Sw-trans Rwu Ruv , (proj₁ ∘ proj₂ $ T3 (R-Sw-trans
Rwu Ruv))}

  M' : O.Model W' R' S' V'
  M' = O.model F'

module PrefaceTheoremAll
  {ℓW ℓR ℓS}
  (T : ∀ {ℓW ℓS} → (W : Set ℓW) → REL₃ W W (Pred W ℓW) ℓS → Set (lsuc
ℓW ⊔ ℓS))
  {W R S}
  (F : Frame {ℓW} {ℓR} {ℓS} W R S T)
  (T3 : Trans-3 W S)
  (V : W → Pred Var lzero)
  where
  open Frame F
  open OrdModel T F T3 V
  open PGeneric T
  open FrameProperties T F

  M : Model W R S V
  M = model {V = V} F

  module Theorem
    (dec : MultiDecidableModel M)
    (dec' : O.DecidableModel M')
    (∈S? : Decidable₃ S)
    (∈S'? : Decidable₃ S')
    (∈SV? : ∀ {w u Y} → S w u Y → Decidable Y)
    where

    private
      open Extended dec ∈S? ∈SV?
      module O' = O.Extended dec' ∈S'?

      thm⇒ : ∀ {w A} → M , w ⊩ A → M' O., w ⊩ A
      thm⇐ : ∀ {w A} → M' O., w ⊩ A → M , w ⊩ A
      thm'⇒ : ∀ {w A} → M , w ⊮ A → M' O., w ⊮ A

      thm⇐ x = ⊩⇔¬⊮ ⇐ λ {y → O.⊮→¬⊩ (thm'⇒ y) x}

      thm⇒ (var x) = O.var x
      thm⇨ A@(impl x) = O'.⊩↝ ⇐ λ {wA → thm⇨ ((⊩↝ ⇒ A) (thm⇦ wA))}
      thm⇨ {w} F@(rhd {D} {E} x) = O'.⊩▷ ⇐
        λ { {x} Rwx x⊩D → case (⊩▷ ⇒ F) Rwx (thm⇦ x⊩D) of
        λ { (V , SwxV , V⊩E) → proj₁ (T3 SwxV) , (V , SwxV , refl)
        , thm⇨ ( V⊩E (proj₁ ∘ proj₂ $ T3 SwxV) )}}

      thm'⇨ (var x) = O.var x
      thm'⇨ {w} (impl {A} {B} a b) = O.impl (thm⇨ a) (thm'⇨ b)
      thm'⇨ bot = O.bot
      thm'⇨ {w} F@(rhd {D} {E} _) = case ⊮▷ ⇒ F of
        λ {(x , Rwx , x⊩D , px) → O'.⊮▷ ⇐ (x , Rwx , thm⇨ x⊩D ,
          λ { (V , SwxV , refl) →
          let k = proj₁ (T3 SwxV)
              Rwk : R w k
              Rwk = SwuY⊆Rw SwxV (proj₁ ∘ proj₂ $ T3 SwxV)
              Swxk : S w x { k }
               Swxk = S⊆{v}' $ (proj₂ ∘ proj₂ $ T3 SwxV) (S-quasirefl
Rwk)
          in case px _ (_ , refl) Swxk of
            λ { (_ , refl , snd) → thm'⇨ snd}})}
```

B.9. GeneralizedVeltmanSemantics/Properties/M

The frame condition for the M principle.

```
module GeneralizedVeltmanSemantics.Properties.M where

open import Function.Equivalence using (_⇔_; equivalence; module Equivalence)

open import Agda.Builtin.Nat using (Nat; suc; zero)
open import Agda.Builtin.Unit using (⊤; tt)
open import Agda.Primitive using (Level; lzero; lsuc)
open import Data.Empty using (⊥; ⊥-elim)
open import Data.Product using (Σ; proj₁; proj₂; _×_) renaming (_,_ to
_,_)
open import Data.Sum using (_⊎_; inj₁; inj₂; [_,_])
open import Function using (_∘_; const; case_of_; id)
open import Function.Equality using (_⟨$⟩_)
open import Relation.Binary using (REL; Rel; Transitive)
open import Relation.Nullary using (yes; no; ¬_)
open import Relation.Unary using (Pred; _∈_; _∉_; Decidable; Satisfiable;
_⊆_; _∩_; {_})
open import Relation.Binary using (Irreflexive) renaming (Decidable to
Decidable₂)
open import Relation.Binary.PropositionalEquality using (_≡_; refl;
subst; trans; sym)

open import Formula
open import GeneralizedVeltmanSemantics
open import Base
open import GeneralizedVeltmanSemantics.Properties
  using (module Extended; ⊩¬; ⊮¬; ⊩□; ⊩◊; ⊩MP; ⊩∧; ⊩→¬⊮; ⊮→¬⊩)
```

```
import Principles as P

private
  variable
    ℓW ℓR ℓS : Level

M-condition : ∀ {W R S} → FrameL {ℓW} {ℓR} {ℓS} W R S → Set _
M-condition {W = W} {R = R} {S = S} F = ∀ {w u V} → S w u V → Σ 𝕎 λ V'
→ V' ⊆ V × S w u V'
  × ∀ {z v'} → v' ∈ V' → R v' z → R u z
  where open FrameL F

module M-soundness
  {W R S V}
  {M : Model {lzero} {lzero} {lzero} W R S V}
  (M,_*⊩?_ : MultiDecidableModel M)
  (∈S? : Decidable₃ S)
  (S? : S-decidable (Model.F M))
  (∈SV? : ∀ {w u Y} → S w u Y → Decidable Y) where

  open Model M
  open FrameL F
  open Extended M,_*⊩?_ ∈S? ∈SV?

  ⊩M : ∀ {w A B C} → M-condition F → M , w ⊩ A ▷ B ↝ (A ∧ □ C) ▷ (B ∧
□ C)
  ⊩M {w} {A} {B} {C} cM = ⊩↝ ⇐ λ A▷B → ⊩▷ ⇐ λ { {x} Rwx x⊩A∧□C
    → case ⊩∧ ⇒ x⊩A∧□C of λ { (x⊩A , x⊩□C) → case (⊩▷ ⇒ A▷B) Rwx x⊩A of
    λ { (Z , SwxZ , Z⊩B) → case cM SwxZ of
    λ { (Z' , Z'⊆Z , SwxZ' , snd) → Z' , SwxZ' , λ {p → ⊩∧ ⇐ (Z⊩B (Z'⊆Z
p) ,
     ⊩□ ⇐ λ {y → (⊩□ ⇒ x⊩□C) (snd p y)})}}}}}

module M-completeness
  {W R S}
  {F : FrameL {lzero} {lzero} {lzero} W R S}
  (∈S? : Decidable₃ S)
  (dec : ∀ V → MultiDecidableModel (model {V = V} F))
  (∈SV? : ∀ {w u Y} → S w u Y → Decidable Y)
  where
  open FrameL F

  *⊩M : Set₁
  *⊩M = P.M (F *⊩_)

  pattern a = 0
  pattern b = 1
  pattern c = 2

  ⊩M⇒M-condition : *⊩M → M-condition F
  ⊩M⇒M-condition ⊩M {w} {u} {V} SwuV
```

```
    = case (⊩▷ ⇒ (⊩MP (⊩M Val w) w⊩a▷b)) Rwu u⊩a∧□c of
       λ { (Z , SwuZ , Z⊩b∧□c) → Z , (λ {x → [b] ⇒ proj₁ (⊩∧ ⇒ Z⊩b∧□c
x)}) , SwuZ
      , λ {x x₁ → [c] ⇒ (⊩□ ⇒ (proj₂ (⊩∧ ⇒ Z⊩b∧□c x))) x₁}}
    where
    Rwu : R w u
    Rwu = Sw⊆Rw SwuV
    Val : Valuation F
    Val w a = w ≡ u
    Val w b = w ∈ V
    Val w c = R u w
    Val w (suc (suc (suc _))) = ⊥
    M = model {V = Val} F
    open Extended (dec Val) ∈S? ∈SV?
    [a] : ∀ {w} → M , w ⊩ var a ⇔ w ≡ u
    [a] = equivalence (λ { (var x) → x}) λ {z → var z}
    [b] : ∀ {w} → M , w ⊩ var b ⇔ w ∈ V
    [b] = equivalence (λ { (var x) → x}) λ {z → var z}
    [c] : ∀ {w} → M , w ⊩ var c ⇔ R u w
    [c] = equivalence (λ { (var x) → x}) λ {z → var z}
    w⊩a▷b : M , w ⊩ var a ▷ var b
    w⊩a▷b = ⊩▷ ⇐ λ { {i} Rwi ia → case [a] ⇒ ia of λ {refl → V , (SwuV
, λ {x → [b] ⇐ x})}}
    u⊩a∧□c : M , u ⊩ var a ∧ □ var c
    u⊩a∧□c = ⊩∧ ⇐ ([a] ⇐ refl , ⊩□ ⇐ λ x → [c] ⇐ x)
```

B.10. GeneralizedVeltmanSemantics/Properties/M_0

The frame condition for the M_0 principle.

```
module GeneralizedVeltmanSemantics.Properties.M₀ where

open import Function.Equivalence using (_⇔_; equivalence; module Equivalence)

open import Agda.Builtin.Nat using (Nat; suc; zero)
open import Agda.Builtin.Unit using (⊤; tt)
open import Agda.Primitive using (Level; lzero; lsuc)
open import Data.Empty using (⊥; ⊥-elim)
open import Data.Product using (Σ; proj₁; proj₂; _×_) renaming (_,_ to
_,_)
open import Data.Sum using (_⊎_; inj₁; inj₂; [_,_])
open import Function using (_∘_; const; case_of_; id)
open import Function.Equality using (_⟨$⟩_)
open import Relation.Binary using (REL; Rel; Transitive)
open import Relation.Nullary using (yes; no; ¬_)
open import Relation.Unary using (Pred; _∈_; _∉_; Decidable; Satisfiable;
_⊆_; _∩_; {_})
open import Relation.Binary using (Irreflexive) renaming (Decidable to
Decidable₂)
open import Relation.Binary.PropositionalEquality using (_≡_; refl;
subst; trans; sym)
```

```
open import Formula
open import GeneralizedVeltmanSemantics
open import Base
open import GeneralizedVeltmanSemantics.Properties
  using (module Extended; ⊩¬; ⊮¬; ⊩□; ⊩◇; ⊩MP; ⊩∧; ⊩→¬⊮; ⊮→¬⊩)
import Principles as P

private
  variable
    ℓW ℓR ℓS : Level

M₀-condition : ∀ {W R S} → FrameL {ℓW} {ℓR} {ℓS} W R S → Set _
M₀-condition {W = W} {R = R} {S = S} F = ∀ {w x y Y} → R w x → R x y →
S w y Y → Σ 𝕎 λ Y'
  → Y' ⊆ Y × S w x Y' × ∀ {y'} z → y' ∈ Y' → R y' z → R x z
  where open FrameL F


module M₀-soundness
  {W R S V}
  {M : Model {lzero} {lzero} {lzero} W R S V}
  (M,_*⊩?_ : MultiDecidableModel M)
  (∈S? : Decidable₃ S)
  (S? : S-decidable (Model.F M))
  (∈SV? : ∀ {w u Y} → S w u Y → Decidable Y) where

  open Model M
  open FrameL F
  open Extended M,_*⊩?_ ∈S? ∈SV?

  ⊩M₀ : ∀ {w A B C} → M₀-condition F → M , w ⊩ A ▷ B ⇝ (◇ A ∧ □ C) ▷
(B ∧ □ C)
  ⊩M₀ {w} {A} {B} {C} c = ⊩⇝ ⇐ λ A▷B → ⊩▷ ⇐ λ {u} Rwu a → case ⊩∧ ⇒ a of
    λ { (u◇A , u□C) → case ⊩◇ ⇒ u◇A of λ { (v , Ruv , vA)
    → case (⊩▷ ⇒ A▷B) (R-trans Rwu Ruv) vA of λ { (Y , SwvY , YB)
    → case c Rwu Ruv SwvY of λ { (Y' , Y'⊆Y , SwuY' , snd)
    → Y' , SwuY' , λ { {y'} y'∈Y' → ⊩∧ ⇐ (YB (Y'⊆Y y'∈Y') , (⊩□ ⇐ (λ
{z} Ry'z
    → (⊩□ ⇒ u□C) (snd z y'∈Y' Ry'z))))}}}}}

module M₀-completeness
  {W R S}
  {F : FrameL {lzero} {lzero} {lzero} W R S}
  (∈S? : Decidable₃ S)
  (dec : ∀ V → MultiDecidableModel (model {V = V} F))
  (∈SV? : ∀ {w u Y} → S w u Y → Decidable Y)
  where
  open FrameL F

  *⊩M₀ : Set₁
  *⊩M₀ = P.M₀ (F *⊩_)

  pattern a = 0
  pattern b = 1
  pattern c = 2

  ⊩M₀⇒M₀-condition : *⊩M₀ → M₀-condition F
  ⊩M₀⇒M₀-condition ⊩M₀ {w} {x} {y} {Y} Rwx Rxy SwyY
    = case (⊩▷ ⇒ ⊩MP (⊩M₀ Val w) w⊩a▷b) Rwx x⊩◊a∧□c of
      λ { (Y' , SwxY' , snd) → Y' , (λ {y → [b] ⇒ proj₁ (⊩∧ ⇒ snd y)})
      , SwxY' , (λ { {y'} z y'∈Y' Ry'z → [c] ⇒ (⊩□ ⇒ (proj₂ (⊩∧ ⇒ snd
y'∈Y'))) Ry'z})}
    where
    Val : Valuation F
    Val w a = w ≡ y
    Val w b = w ∈ Y
    Val w c = R x w
    Val w (suc (suc (suc _))) = ⊥
    M = model {V = Val} F
    open Extended (dec Val) ∈S? ∈SV?
    [a] : ∀ {w} → M , w ⊩ var a ⇔ w ≡ y
    [a] = equivalence (λ { (var x) → x}) λ {z → var z}
    [b] : ∀ {w} → M , w ⊩ var b ⇔ w ∈ Y
    [b] = equivalence (λ { (var x) → x}) λ {z → var z}
    [c] : ∀ {w} → M , w ⊩ var c ⇔ R x w
    [c] = equivalence (λ { (var x) → x}) λ {z → var z}
    w⊩a▷b : M , w ⊩ var a ▷ var b
    w⊩a▷b = ⊩▷ ⇐ λ {Rwu ua → case [a] ⇒ ua of λ {refl → Y , (SwyY , λ
{x₁ → var x₁})}}
    x⊩◊a∧□c : M , x ⊩ ◊ var a ∧ □ var c
    x⊩◊a∧□c = ⊩∧ ⇐ (⊩◊ ⇐ (y , Rxy , [a] ⇐ refl) , ⊩□ ⇐ λ {p → [c] ⇐ p})
```

B.11. GeneralizedVeltmanSemantics/Properties/P$_0$

The frame condition for the P$_0$ principle.

```
module GeneralizedVeltmanSemantics.Properties.P₀ where

open import Function.Equivalence using (_⇔_; equivalence; module Equivalence)

open import Agda.Builtin.Nat using (Nat; suc; zero)
open import Agda.Builtin.Unit using (⊤; tt)
open import Agda.Primitive using (Level; lzero; lsuc)
open import Data.Empty using (⊥; ⊥-elim)
open import Data.Product using (Σ; proj₁; proj₂; _×_) renaming (_,_ to
_,_)
open import Data.Sum using (_⊎_; inj₁; inj₂; [_,_])
open import Function using (_∘_; const; case_of_; id)
open import Function.Equality using (_⟨$⟩_)
open import Relation.Binary using (REL; Rel; Transitive)
open import Relation.Nullary using (yes; no; ¬_)
open import Relation.Unary using (Pred; _∈_; _∉_; Decidable; Satisfiable;
_⊆_; _∩_; {_})
open import Relation.Binary using (Irreflexive) renaming (Decidable to
Decidable₂)
open import Relation.Binary.PropositionalEquality using (_≡_; refl;
subst; trans; sym)

open import Formula
open import GeneralizedVeltmanSemantics
open import Base
open import GeneralizedVeltmanSemantics.Properties
  using (module Extended; ⊩¬; ⊮¬; ⊩□; ⊩◇; ⊩MP; ⊩∧; ⊩→¬⊮; ⊮→¬⊩)
import Principles as P

private
  variable
    ℓW ℓR ℓS : Level

P₀-condition : ∀ {ℓZ} {W R S} → FrameL {ℓW} {ℓR} {ℓS} W R S → Set _
P₀-condition {ℓZ = ℓZ} {W = W} {R = R} {S = S} F = ∀ {w x y Y} → R w x
→ R x y → S w y Y →
  (Z : Pred W ℓZ)
  → (∀ y' → y' ∈ Y → Satisfiable (Z ∩ R y'))
  → Σ 𝕎 λ Z' → (Z' ⊆ Z) × S x y Z'
  where open FrameL F

module P₀-soundness
  {W R S V}
  {M : Model {lzero} {lzero} {lzero} W R S V}
  (M,_*⊩?_ : MultiDecidableModel M)
  (∈S? : Decidable₃ S)
  (S? : S-decidable (Model.F M))
  (∈SV? : ∀ {w u Y} → S w u Y → Decidable Y) where

  open Model M
  open FrameL F
  open Extended M,_*⊩?_ ∈S? ∈SV?

  ⊩P₀ : ∀ {w A B} → P₀-condition F → M , w ⊩ A ▷ (◇ B) ⇝ □ (A ▷ B)
  ⊩P₀ {w} {A} {B} c-P₀ = ⊩⇝ ⇐ λ A▷◇B → ⊩□ ⇐ λ {x} Rwx → ⊩▷ ⇐ λ {y} Rxy
yA
    → case (⊩▷ ⇒ A▷◇B) (R-trans Rwx Rxy) yA of
    λ { (Y , SwyY , snd) → case c-P₀ Rwx Rxy SwyY (M ,_⊩ B)
     (λ y' y'∈Y → case ⊩◇ ⇒ snd y'∈Y of λ { (z , Ry'z , zB) → z , zB ,
Ry'z}) of
    λ { (Z' , ZB , SxyZ') → Z' , SxyZ' , λ { w' → ZB w'}}
    }

module P₀-completeness
  {W R S}
  {F : FrameL {lzero} {lzero} {lzero} W R S}
  (∈S? : Decidable₃ S)
  (dec : ∀ V → MultiDecidableModel (model {V = V} F))
  (∈SV? : ∀ {w u Y} → S w u Y → Decidable Y)
  where
  open FrameL F

  *⊩P₀ : Set₁
  *⊩P₀ = P.P₀ (F *⊩_)

  pattern a = 0
  pattern b = 1

  ⊩P₀⇒P₀-condition : *⊩P₀ → P₀-condition F
  ⊩P₀⇒P₀-condition ⊩P₀ {w} {x} {y} {Y} Rwx Rxy SwyY Z p =
    case (⊩▷ ⇒ x⊩a▷b) Rxy ([a] ⇐ refl) of
    λ { (Z' , SxyZ' , Z'⊩b) → Z' , (λ {z → [b] ⇒ (Z'⊩b z)}) , SxyZ'}
    where
    Val : Valuation F
    Val w a = w ≡ y
    Val w b = w ∈ Z
    Val w (suc (suc _)) = ⊥
    M = model {V = Val} F
    open Extended (dec Val) ∈S? ∈SV?
    [a] : ∀ {w} → M , w ⊩ var a ⇔ w ≡ y
    [a] = equivalence (λ { (var x) → x}) λ {z → var z}
    [b] : ∀ {w} → M , w ⊩ var b ⇔ w ∈ Z
    [b] = equivalence (λ { (var x) → x}) λ {z → var z}
    w⊩a▷◊b : M , w ⊩ var a ▷ ◊ var b
    w⊩a▷◊b = ⊩▷ ⇐ λ { {u} Rwu u⊩a → case [a] ⇒ u⊩a of
      λ { refl → Y , SwyY , λ { {v} v∈Y → case p v v∈Y of
      λ { (fst , fst₁ , snd) → ⊩◊ ⇐ (fst , (snd , ([b] ⇐ fst₁)))}}}}
    w⊩□a▷b : M , w ⊩ □ (var a ▷ var b)
    w⊩□a▷b = ⊩MP (⊩P₀ Val w) w⊩a▷◊b
    x⊩a▷b : M , x ⊩ var a ▷ var b
    x⊩a▷b = (⊩□ ⇒ w⊩□a▷b) Rwx
```

B.12. GeneralizedVeltmanSemantics/Properties/R

The frame condition for the R principle.

```
module GeneralizedVeltmanSemantics.Properties.R where

open import Function.Equivalence using (_⇔_; equivalence; module Equivalence)

open import Agda.Builtin.Nat using (Nat; suc; zero)
open import Agda.Builtin.Unit using (⊤; tt)
open import Agda.Primitive using (Level; lzero; lsuc)
open import Data.Empty using (⊥; ⊥-elim)
open import Data.Product using (Σ; proj₁; proj₂; _×_) renaming (_,_ to
_,_)
open import Data.Sum using (_⊎_; inj₁; inj₂; [_,_])
open import Function using (_∘_; const; case_of_; id)
open import Function.Equality using (_⟨$⟩_)
open import Relation.Binary using (REL; Rel; Transitive)
open import Relation.Nullary using (yes; no; ¬_)
open import Relation.Unary using (Pred; _∈_; _∉_; Decidable; Satisfiable;
_⊆_; _∩_; {_})
open import Relation.Binary using (Irreflexive) renaming (Decidable to
Decidable₂)
open import Relation.Binary.PropositionalEquality using (_≡_; refl;
subst; trans; sym)

open import Formula
open import GeneralizedVeltmanSemantics
open import Base using (_⇒_; _⇐_; Decidable₃)
open import GeneralizedVeltmanSemantics.Properties
  using (module Extended; ⊩¬; ⊮¬; ⊩□; ⊩◊; ⊩MP; ⊩∧; ⊩→¬⊮; ⊮→¬⊩)
import Principles as P

private
  variable
    ℓW ℓR ℓS : Level

R-condition : ∀ {W R S} → FrameL {ℓW} {ℓR} {ℓS} W R S → Set _
R-condition {W = W} {R = R} {S = S} F = ∀ {w x y Y ℂ} → R w x → R x y
→ S w y Y → IsChoiceSet ℂ x y → Decidable ℂ
  → Σ 𝕎 λ Y' → Y' ⊆ Y × S w x Y' × ∀ {y'} → y' ∈ Y' → ∀ {z} → R y' z →
z ∈ ℂ
  where
  open FrameL F


module R-soundness
  {W R S V}
  {M : Model {lzero} {lzero} {lzero} W R S V}
  (M,_*⊩?_ : MultiDecidableModel M)
  (∈S? : Decidable₃ S)
  (S? : S-decidable (Model.F M))
  (∈SV? : ∀ {w u Y} → S w u Y → Decidable Y) where

  open Model M
  open FrameL F
  open Extended M,_*⊩?_ ∈S? ∈SV?

  ⊩R : ∀ {w A B C} → R-condition F →
    M , w ⊩ A ▷ B ↝ (¬' (A ▷ C) ▷ (B ∧ (□ (¬' C))))
  ⊩R {w} {A} {B} {C} c = ⊩↝ ⇐ λ A▷B → ⊩▷ ⇐ λ {x} Rwx tmp → case ⊩¬ ⇒
tmp of
    λ { (rhd (y , Rxy , vA , snd)) → case (⊩▷ ⇒ A▷B) (R-trans Rwx Rxy)
vA of
    λ { (Y , SwxY , YB) → case c Rwx Rxy SwxY (Rxy
     , λ {Z} SxyZ → case snd Z (Swu-sat SxyZ) of
     λ { (inj₁ p) → ⊥-elim (p SxyZ)
     ; (inj₂ (z , zZ , z¬C)) → z , zZ , aux x y ⇐ (⊩¬ ⇐ z¬C , Z , zZ ,
SxyZ)}) (dec x y)
      of λ { (Y' , Y'⊆Y , SwxY' , snd) → Y' , SwxY' , λ { {y'} y'∈Y'
      → ⊩∧ ⇐ (YB (Y'⊆Y y'∈Y') , ⊩□ ⇐ λ { {v} Ry'v → proj₁ (aux x y ⇒
snd y'∈Y' Ry'v)})}}}}
      where
      ℂ : W → W → 𝕎
      ℂ x y g with M, g ⊩? ¬' C
      ... | inj₂ _ = ⊥
      ... | inj₁ _ with S? x y g
      ... | inj₁ _  = ⊤
      ... | inj₂ _ = ⊥
      dec : (x y : W) → Decidable (ℂ x y)
      dec x y g with M, g ⊩? ¬' C
      ... | inj₂ _ = no λ z → z
      ... | inj₁ _ with S? x y g
      ... | inj₁ _  = yes tt
      ... | inj₂ _ = no λ z → z
      aux : ∀ {g} x y → g ∈ ℂ x y ⇔ (M , g ⊩ ¬' C × Σ 𝕎 λ U → g ∈ U ×
S x y U)
      aux {g} x y = equivalence ⇨ ⇦
       where ⇨ : g ∈ ℂ x y → (M , g ⊩ ¬' C × Σ 𝕎 λ U → g ∈ U × S x y U)
             ⇨ p with M, g ⊩? ¬' C
             ... | inj₂ _ = ⊥-elim p
             ... | inj₁ k with S? x y g
             ... | inj₂ _ = ⊥-elim p
             ... | inj₁ k' = k , k'
            ⇦ : (M , g ⊩ ¬' C × Σ 𝕎 λ U → g ∈ U × S x y U) → g ∈ ℂ x y
             ⇦ (p1 , p2) with M, g ⊩? ¬' C
             ... | inj₂ m1 = ⊮→¬⊩ m1 p1
             ... | inj₁ m1 with S? x y g
             ... | inj₁ m2 = tt
              ... | inj₂ m2 = case p2 of λ { (Z , gZ , SxyZ) → m2 Z
SxyZ gZ}

module R-condition
  {W R S}
  {F : FrameL {lzero} {lzero} {lzero} W R S}
  (S? : Decidable₃ S)
  (dec : ∀ V → MultiDecidableModel (model {V = V} F))
  (∈SV? : ∀ {w u Y} → S w u Y → Decidable Y)
  where
  open FrameL F
  *⊩R : Set₁
  *⊩R = P.R (F *⊩_)

  pattern p = 0
  pattern q = 1
  pattern s = 2

  ⊩R⇒R-condition : *⊩R → R-condition F
```

```
 ⊩R⇒R-condition ⊩Ra {w} {x} {y} {Y} {ℂ} Rwx Rxy SwyY (_ , C) ∈ℂ?
  = case (⊩▷ ⇒ w⊩G) Rwx (⊩¬ ⇐ x⊮p▷s) of
   λ { (Y' , SwxY' , snd) → Y' , (λ {v} v∈ → Y⊩q ⇒ proj₁ (⊩∧ ⇒ (snd
v∈))) , SwxY' ,
   λ {y'} y'∈ {z} Ry'z → ℂ⊩¬s ⇒ ((⊩□ ⇒ proj₂ (⊩∧ ⇒ (snd y'∈))) Ry'z)}
  where
  V : Valuation F
  V u p = u ≡ y
  V u q = u ∈ Y
  V u s = u ∈ ℂ
  V w (suc (suc (suc x))) = ⊥
  M = model {V = V} F
  open Extended (dec V) S? ∈SV?
  Y⊩q : ∀ {w'} → M , w' ⊩ var q ⇔ w' ∈ Y
  Y⊩q {w'} = equivalence ⇒ ⇐
   where
   ⇒ : M , w' ⊩ var q → Y w'
   ⇒ (var x) = x
   ⇐ : Y w' → M , w' ⊩ var q
   ⇐ x = var x
  ℂ⊩¬s : ∀ {w'} → M , w' ⊩ var s ⇔ w' ∈ ℂ
  ℂ⊩¬s {w'} = equivalence ⇒ ⇐
   where
   ⇐ : ℂ w' → M , w' ⊩ var s
   ⇐ x = var x
   ⇒ : M , w' ⊩ var s → ℂ w'
   ⇒ (var vr) with ∈ℂ? w'
   ... | yes ans = ans
   ... | no z = vr
  y⊩p : ∀ {w'} → M , w' ⊩ var p ⇔ w' ≡ y
  y⊩p {w'} = equivalence ⇒ ⇐
   where ⇒ : M , w' ⊩ var 0 → w' ≡ y
         ⇒ (var refl) = refl
         ⇐ : w' ≡ y → M , w' ⊩ var 0
         ⇐ refl = var refl
  w⊩p▷q : M , w ⊩ var p ▷ var q
  w⊩p▷q = ⊩▷ ⇐ λ {w'} Rww' w'⊩p → case y⊩p ⇒ w'⊩p of
   λ {refl → Y , SwyY , λ {v'} x₁ → var x₁}
  w⊩G : M , w ⊩ (¬' (var 0 ▷ ¬' (var 2))) ▷ (var 1 ∧ (□ (var 2)))
  w⊩G = ⊩MP (⊩Ra V w) w⊩p▷q
  x⊮p▷s : M , x ⊮ var p ▷ (¬' (var s))
  x⊮p▷s = rhd (y , Rxy , y⊩p ⇐ refl , λ {Y' sY' → case S? x y Y' of
   λ { (yes z) → inj₂ (case C z of (λ { (y' , fst , snd) → y' ,
     (fst , impl (var snd) bot)}));
   (no z) → inj₁ z}})
```

B.13. GeneralizedVeltmanSemantics/Properties/R^1

The frame condition for the R^1 principle.

```
module GeneralizedVeltmanSemantics.Properties.R¹ where
```

```
open import Function.Equivalence using (_⇔_; equivalence; module Equivalence)

open import Agda.Builtin.Nat using (Nat; suc; zero)
open import Agda.Builtin.Unit using (⊤; tt)
open import Agda.Primitive using (Level; lzero; lsuc)
open import Data.Empty using (⊥; ⊥-elim)
open import Data.Product using (Σ; proj₁; proj₂; _×_) renaming (_,_ to
_,_)
open import Data.Sum using (_⊎_; inj₁; inj₂; [_,_])
open import Function using (_∘_; const; case_of_; id)
open import Function.Equality using (_⟨$⟩_)
open import Relation.Binary using (REL; Rel; Transitive)
open import Relation.Nullary using (yes; no; ¬_)
open import Relation.Unary using (Pred; _∈_; _∉_; Decidable; Satisfiable;
_⊆_; _∩_; {_})
open import Relation.Binary using () renaming (Decidable to Decidable₂)
open import Relation.Binary.PropositionalEquality using (_≡_; refl;
subst; trans; sym)

open import Formula
open import GeneralizedVeltmanSemantics
open import Base using (_⇒_; _⇐_; Decidable₃)
open import GeneralizedVeltmanSemantics.Properties
  using (module Extended; ⊩¬; ⊮¬; ⊩□; ⊩◇; ⊩MP; ⊩∧; ⊩→¬⊮; ⊮→¬⊩)
open import GeneralizedFrame using (module Predicates)
import Principles as P

private
  variable
    ℓW ℓR ℓS : Level

R¹-condition : ∀ {W R S} → FrameL {lzero} {lzero} {lzero} W R S → Set _
R¹-condition {W = W} {R = R} {S = S} F = ∀ {w x y z} {𝔸 𝔹 ℂ 𝔻 : 𝕎}
  → R w x → R x y → R y z
  → (∀ {u} → R w u → u ∈ 𝔸 → Σ 𝕎 λ V → S w u V × V ⊆ 𝔹)
  → (∀ {u} → R x u → u ∈ 𝔻 → Σ 𝕎 λ V → S x u V × V ⊆ 𝔸)
  → (∀ {V} → S y z V → Σ W λ v → v ∈ V × v ∈ ℂ)
  → z ∈ 𝔻
  → Σ 𝕎 λ V → V ⊆ 𝔹 × S w x V × R[ V ] ⊆ ℂ
  where open FrameL F
        open Predicates F

module R₁-soundness
  {W R S V}
  {M : Model {lzero} {lzero} {lzero} W R S V}
  (M,_*⊩?_ : MultiDecidableModel M)
  (∈S? : Decidable₃ S)
  (S? : S-decidable (Model.F M))
  (∈SV? : ∀ {w u Y} → S w u Y → Decidable Y) where

  open Model M
  open FrameL F
  open Extended M,_*⊩?_ ∈S? ∈SV?

  ⊩R¹ : ∀ {w A B C D} → R¹-condition F
    → M , w ⊩ A ▷ B ↝ (◇ (D ◁ C) ∧ (D ▷ A)) ▷ (B ∧ □ C)
  ⊩R¹ {w} {A} {B} {C} {D} c = ⊩↝ ⇐ λ w⊩A▷B → ⊩▷ ⇐ λ { {x} Rwx x⊩∧ →
    case ⊩∧ ⇒ x⊩∧ of λ { (x⊩◇D◁C , x⊩D▷A) → case ⊩◇ ⇒ x⊩◇D◁C of
    λ { (y , Rxy , y⊩D◁C) → case ⊩◁ ⇒ y⊩D◁C of
    λ { (z , Ryz , z⊩D , pz) → case c {𝔸 = 𝔸} {𝔹 = 𝔹} {𝔻 = 𝔻} Rwx Rxy
Ryz
    (λ {u} Rwu u∈𝔸 → case (⊩▷ ⇒ w⊩A▷B) Rwu (∈[⊩ A ] ⇒ u∈𝔸) of λ { (V ,
SwuV , V⊩B)
      → V , SwuV , λ {z → ∈[⊩ B ] ⇐ V⊩B z}}) (λ {u} Rxu uD → case (⊩▷ ⇒
x⊩D▷A) Rxu (∈[⊩ D ] ⇒ uD) of
      λ { (V , SxuV , V⊩A) → V , SxuV , λ {v∈V → ∈[⊩ A ] ⇐ V⊩A v∈V}})
(λ { {V} SyzV → case pz SyzV of
      λ { (v , v∈V , vC) → v , v∈V , (∈K y z ⇐ (V , SyzV , v∈V , vC))}})
(∈[⊩ D ] ⇐ z⊩D) of
    λ { (V , V⊆𝔹 , SwxV , R[V]⊆K) → V , SwxV , λ { {v} v' → ⊩∧ ⇐ (∈[⊩ B
] ⇒ V⊆𝔹 v' , ⊩□ ⇐
    λ { {u} Rvu → case ∈K y z ⇒ R[V]⊆K (v , v' , Rvu) of
    λ { (_ , _ , _ , snd) → snd}})}}}}}}
    where
    K : (y z : W) → 𝕎
    K y z u with S? y z u
    ... | inj₂ _ = ⊥
    ... | inj₁ _ with M, u ⊩? C
    ... | inj₁ _ = ⊤
    ... | inj₂ _ = ⊥
    ∈K : ∀ y z {u} → u ∈ (K y z) ⇔ Σ 𝕎 λ V → S y z V × u ∈ V × M , u ⊩ C
    ∈K y z {u} = equivalence ⇒ ⇐
      where
      ⇒ : u ∈ (K y z) → Σ 𝕎 λ V → S y z V × u ∈ V × M , u ⊩ C
      ⇒ x with S? y z u
      ... | inj₁ (V , uV , SyzV) with M, u ⊩? C
      ... | inj₁ p = V , SyzV , uV , p
      ... | inj₂ _ = ⊥-elim x
      ⇐ : (Σ 𝕎 λ V → S y z V × u ∈ V × M , u ⊩ C) → u ∈ (K y z)
      ⇐ (V , SyzV , uV) with S? y z u
      ... | inj₂ p = p V SyzV (proj₁ uV)
      ... | inj₁ p with M, u ⊩? C
      ... | inj₁ p' = tt
      ⇐ (V , SyzV , fst , snd) | inj₁ p | inj₂ p' = ⊮→¬⊩ p' snd
    K∈𝒞 : {y z : W} → R y z → (∀ {V} → S y z V → Σ W (λ b → b ∈ V × M ,
b ⊩ C)) → IsChoiceSet (K y z) y z
    K∈𝒞 {y} {z} Ryz p = Ryz , λ { {Y} SyzY → case p SyzY of λ { (b ,
b∈V , bC) → b , b∈V , ∈K y z ⇐ (Y , SyzY , b∈V , bC)}}
    𝔸 𝔹 𝔻 : 𝕎
    𝔸 = [⊩ A ]
    𝔹 = [⊩ B ]
    𝔻 = [⊩ D ]
```

```
module R₁-completeness
  {W R S}
  {F : FrameL {lzero} {lzero} {lzero} W R S}
  (∈S? : Decidable₃ S)
  (dec : ∀ V → MultiDecidableModel (model {V = V} F))
  (∈SV? : ∀ {w u Y} → S w u Y → Decidable Y)
  where
  open FrameL F
  *⊩R¹ : Set₁
  *⊩R¹ = P.R¹ (F *⊩_)

  pattern a = 0
  pattern b = 1
  pattern c = 2
  pattern d = 3


  ⊩R¹⇒R¹-condition : *⊩R¹ → R¹-condition F
  ⊩R¹⇒R¹-condition ⊩R¹ {w} {x} {y} {z} {𝔸} {𝔹} {ℂ} {𝔻}
    Rwx Rxy Ryz ∈𝔸 ∈𝔻 [d◁c] z∈𝔻 = case (⊩▷ ⇒ ⊩MP w⊩R¹ w⊩a▷b) Rwx (⊩∧ ⇐
(x⊩◇d◁c , x⊩d▷a)) of
    λ { (V , SwxV , V⊩b∧□c) → V , (λ {p → [b] ⇒ proj₁ (⊩∧ ⇒ V⊩b∧□c p)})
, SwxV ,
    λ { {h} (v , v∈V , Rv) → [c] ⇒ (⊩□ ⇒ proj₂ (⊩∧ ⇒ V⊩b∧□c v∈V)) Rv}}
    where
    Val : Valuation F
    Val y a = y ∈ 𝔸
    Val y b = y ∈ 𝔹
    Val y c = y ∈ ℂ
    Val y d = y ∈ 𝔻
    Val y (suc (suc (suc (suc x)))) = ⊥
    M = model {V = Val} F
    [a] : ∀ {u} → M , u ⊩ var a ⇔ u ∈ 𝔸
    [a] {u} = equivalence (λ { (var x) → x}) (λ x₁ → var x₁)
    [b] : ∀ {u} → M , u ⊩ var b ⇔ u ∈ 𝔹
    [b] {u} = equivalence (λ { (var x) → x}) (λ x₁ → var x₁)
    [c] : ∀ {u} → M , u ⊩ var c ⇔ u ∈ ℂ
    [c] {u} = equivalence (λ { (var x) → x}) (λ x₁ → var x₁)
    [d] : ∀ {u} → M , u ⊩ var d ⇔ u ∈ 𝔻
    [d] {u} = equivalence (λ { (var x) → x}) (λ x₁ → var x₁)
    open Extended (dec Val) ∈S? ∈SV?
    w⊩R¹ : M , w ⊩ var a ▷ var b ⇝ (◇ (var d ◁ var c) ∧ (var d ▷ var
a)) ▷ (var b ∧ □ var c)
    w⊩R¹ = ⊩R¹ Val w
    w⊩a▷b : M , w ⊩ var a ▷ var b
    w⊩a▷b = ⊩▷ ⇐ λ { {u} Rwu u⊩a →
      case ∈𝔸 Rwu ([a] ⇒ u⊩a) of λ { (V , SwuV , V⊆𝔹)
      → V , SwuV , λ {z → [b] ⇐ V⊆𝔹 z}}}
    y⊩d◁c : M , y ⊩ var d ◁ var c
    y⊩d◁c = ⊩◁ ⇐ (z , Ryz , [d] ⇐ z∈𝔻 , λ { {V} SyzV → case [d◁c] SyzV
of λ { (v , v∈V , v∈ℂ) → v , v∈V , [c] ⇐ v∈ℂ}})
    x⊩d▷a : M , x ⊩ var d ▷ var a
    x⊩d▷a = ⊩▷ ⇐ λ { {u} Rxu ud → case ∈𝔻 Rxu ([d] ⇒ ud) of λ { (V ,
SxuV , snd) → V , SxuV , λ {i → [a] ⇐ snd i}}}
    x⊩◊d◁c : M , x ⊩ ◊ (var d ◁ var c)
    x⊩◊d◁c = ⊩◊ ⇐ (y , Rxy , y⊩d◁c)
```

B.14. GeneralizedVeltmanSemantics/Properties/R^2

The frame condition for the R^2 principle.

```
module GeneralizedVeltmanSemantics.Properties.R² where

open import Function.Equivalence using (_⇔_; equivalence; module Equivalence)

open import Agda.Builtin.Nat using (Nat; suc; zero)
open import Agda.Builtin.Unit using (⊤; tt)
open import Agda.Primitive using (Level; lzero; lsuc)
open import Data.Empty using (⊥; ⊥-elim)
open import Data.Product using (Σ; proj₁; proj₂; _×_) renaming (_,_ to
_,_)
open import Data.Sum using (_⊎_; inj₁; inj₂; [_,_])
open import Function using (_∘_; const; case_of_; id)
open import Function.Equality using (_⟨$⟩_)
open import Relation.Binary using (REL; Rel; Transitive)
open import Relation.Nullary using (yes; no; ¬_)
open import Relation.Unary using (Pred; _∈_; _∉_; Decidable; Satisfiable;
_⊆_; _∩_; {_})
open import Relation.Binary using (Irreflexive) renaming (Decidable to
Decidable₂)
open import Relation.Binary.PropositionalEquality using (_≡_; refl;
subst; trans; sym)

open import Formula
open import GeneralizedVeltmanSemantics
open import Base using (_⇒_; _⇐_; Decidable₃)
open import GeneralizedVeltmanSemantics.Properties
  using (module Extended; ⊩¬; ⊮¬; ⊩□; ⊩◊; ⊩MP; ⊩∧; ⊩→¬⊮; ⊮→¬⊩)
open import GeneralizedFrame using (module Predicates)
import Principles as P

private
  variable
    ℓW ℓR ℓS : Level

R²-condition : ∀ {W R S} → FrameL {lzero} {lzero} {lzero} W R S → Set _
R²-condition {W = W} {R = R} {S = S} F =
  ∀ {w x y z s} {𝔸 𝔹 ℂ 𝔻 𝔼 : 𝕎}
  → R w x → R x y → R y z → R z s
  → (∀ {u} → R w u → u ∈ 𝔸 → Σ 𝕎 λ V → S w u V × V ⊆ 𝔹)
  → (∀ {u} → R x u → u ∈ 𝔼 → Σ 𝕎 λ V → S x u V × V ⊆ 𝔸)
  → (∀ {u} → R y u → u ∈ 𝔻 → Σ 𝕎 λ V → S y u V × V ⊆ 𝔼)
  → (∀ {V} → S z s V → Σ W λ v → v ∈ V × v ∈ ℂ)
  → s ∈ 𝔻
  → Σ 𝕎 λ V → S w x V × V ⊆ 𝔹 × R[ V ] ⊆ ℂ
  where open FrameL F
        open Predicates F

module R²-soundness
  {W R S V}
  {M : Model {lzero} {lzero} {lzero} W R S V}
  (M,_*⊩?_ : MultiDecidableModel M)
  (∈S? : Decidable₃ S)
  (S? : S-decidable (Model.F M))
  (∈SV? : ∀ {w u Y} → S w u Y → Decidable Y) where

  open Model M
  open FrameL F
  open Extended M,_*⊩?_ ∈S? ∈SV?

  ⊩R²-aux : ∀ {A B C D E w x y z s}
    → R²-condition F
    → R w x → R x y → R y z → R z s
    → M , w ⊩ A ▷ B
    → M , x ⊩ E ▷ A
    → M , y ⊩ D ▷ E
    → M , s ⊩ D
    → (∀ {V} → S z s V → Satisfiable (V ∩ (M ,_⊩ C)))
    → Σ 𝕎 λ Y → S w x Y × Y ⊆ M ,_⊩ B ∧ □ C
  ⊩R²-aux {A} {B} {C} {D} {E} cond Rwx Rxy Ryz Rzs w⊩A▷B x⊩E▷A y⊩D▷E
s⊩D p
    = case cond Rwx Rxy Ryz Rzs
    (aux w⊩A▷B) (aux x⊩E▷A) (aux y⊩D▷E)
    (aux-choice p) (∈[⊩ D ] ⇐ s⊩D) of
    λ { (Y , SwxY , Y⊆𝔹 , snd) → Y , SwxY , λ { {e} i → ⊩∧ ⇐ (∈[⊩ B ] ⇒
Y⊆𝔹 i , ⊩□ ⇐ λ { {t} j → ∈[⊩ C ] ⇒ snd (e , i , j)})} }
    where
    -- the following function is general enough to be declared outside
this module.
    aux : ∀ {A B w u} → M , w ⊩ A ▷ B → R w u → u ∈ [⊩ A ] → Σ 𝕎 λ V →
S w u V × V ⊆ [⊩ B ]
     aux {A} {B} w⊩A▷B Rwu u∈[A] = case (⊩▷ ⇒ w⊩A▷B) Rwu (∈[⊩ A ] ⇒
u∈[A]) of
       λ { (V , SwuV , snd) → V , SwuV , λ {x₁ → ∈[⊩ B ] ⇐ snd x₁}}
    aux-choice : ∀ {A w x} → (∀ {V} → S w x V → Satisfiable (V ∩ (M ,_⊩
A))) → ∀ {V} → S w x V → Satisfiable (V ∩ [⊩ A ])
     aux-choice {A} p SwxV = case p SwxV of λ { (k , fst , snd) → k ,
fst , ∈[⊩ A ] ⇐ snd }

  ⊩R² : ∀ {w A B C D E} → R²-condition F
    → M , w ⊩ A ▷ B ⇝ (◊ ((D ▷ E) ∧ ◊ (¬' (D ▷ ¬' C))) ∧ (E ▷ A)) ▷ (B
∧ □ C)
```

```
  ⊩R² {w} {A} {B} {C} {D} {E} c = ⊩↝ ⇐ λ {w⊩A▷B → ⊩▷ ⇐
    λ { {x} Rwx x⊩ → case ⊩∧ ⇒ x⊩ of λ { (x⊩◊ , x⊩E▷A) → case ⊩◊ ⇒ x⊩◊
of
    λ { (y , Rxy , y⊩) → case ⊩∧ ⇒ y⊩ of λ { (y⊩D▷E , y⊩◊) → case ⊩◊ ⇒
y⊩◊ of
    λ { (z , Ryz , z⊩) → case ⊩◁ ⇒ z⊩ of λ { (s , Rzs , s⊩D , snd)
    → ⊩R²-aux c Rwx Rxy Ryz Rzs w⊩A▷B x⊩E▷A y⊩D▷E s⊩D snd}}}}}}}

module R²-completeness
  {W R S}
  {F : FrameL {lzero} {lzero} {lzero} W R S}
  (∈S? : Decidable₃ S)
  (dec : ∀ V → MultiDecidableModel (model {V = V} F))
  (∈SV? : ∀ {w u Y} → S w u Y → Decidable Y)
  where
  open FrameL F

  *⊩R² : Set₁
  *⊩R² = P.R² (F *⊩_)

  pattern a = 0
  pattern b = 1
  pattern c = 2
  pattern d = 3
  pattern e = 4


  ⊩R²⇒R²-condition : *⊩R² → R²-condition F
  ⊩R²⇒R²-condition ⊩R² {w} {x} {y} {z} {s} {𝔸} {𝔹} {ℂ} {𝔻} {𝔼}
    Rwx Rxy Ryz Rzs 𝔸▷𝔹 𝔼▷𝔸 𝔻▷𝔼 pℂ s∈𝔻
      = case (⊩▷ ⇒ ⊩MP w⊩R² w⊩a▷b) Rwx (⊩∧ ⇐ (⊩◊ ⇐ (y , Rxy , ⊩∧ ⇐
      (y⊩d▷e , ⊩◊ ⇐ (z , Ryz , z⊩¬d▷¬c))) , x⊩e▷a)) of
      λ { (V , SwxV , snd) → V , SwxV , (λ {i → [b] ⇒ proj₁ (⊩∧ ⇒ snd
i)}) ,
      λ { (u , uV , Rui) → [c] ⇒ (⊩□ ⇒ (proj₂ (⊩∧ ⇒ snd uV))) Rui}}
    where
    Val : Valuation F
    Val y a = y ∈ 𝔸
    Val y b = y ∈ 𝔹
    Val y c = y ∈ ℂ
    Val y d = y ∈ 𝔻
    Val y e = y ∈ 𝔼
    Val y (suc (suc (suc (suc (suc x))))) = ⊥
    M = model {V = Val} F
    [a] : ∀ {u} → M , u ⊩ var a ⇔ u ∈ 𝔸
    [a] {u} = equivalence (λ { (var x) → x}) (λ x₁ → var x₁)
    [b] : ∀ {u} → M , u ⊩ var b ⇔ u ∈ 𝔹
    [b] {u} = equivalence (λ { (var x) → x}) (λ x₁ → var x₁)
    [c] : ∀ {u} → M , u ⊩ var c ⇔ u ∈ ℂ
    [c] {u} = equivalence (λ { (var x) → x}) (λ x₁ → var x₁)
    [d] : ∀ {u} → M , u ⊩ var d ⇔ u ∈ 𝔻
    [d] {u} = equivalence (λ { (var x) → x}) (λ x₁ → var x₁)
    [e] : ∀ {u} → M , u ⊩ var e ⇔ u ∈ 𝔼
    [e] {u} = equivalence (λ { (var x) → x}) (λ x₁ → var x₁)
    open Extended (dec Val) ∈S? ∈SV?
    w⊩R² : M , w ⊩ var a ▷ var b ↝ (◊ ((var d ▷ var e) ∧ ◊ (¬' (var d
▷ ¬' var c))) ∧
      (var e ▷ var a)) ▷ (var b ∧ □ var c)
    w⊩R² = ⊩R² Val w
    w⊩a▷b : M , w ⊩ var a ▷ var b
    w⊩a▷b = [⊩▷] [a] [b] 𝔸▷𝔹
    x⊩e▷a : M , x ⊩ var e ▷ var a
    x⊩e▷a = [⊩▷] [e] [a] 𝔼▷𝔸
    y⊩d▷e : M , y ⊩ var d ▷ var e
    y⊩d▷e = [⊩▷] [d] [e] 𝔻▷𝔼
    z⊩¬d▷¬c : M , z ⊩ ¬' (var d ▷ ¬' var c)
    z⊩¬d▷¬c = ⊩◁ ⇐ (s , Rzs , var s∈𝔻 , λ {SzsV → case pℂ SzsV of
      λ { (s' , s'∈V , s'∈ℂ) → s' , s'∈V , [c] ⇐ s'∈ℂ}})
```

B.15. GeneralizedVeltmanSemantics/Properties/Rⁿ

The frame condition for the Rⁿ principle.

```
module GeneralizedVeltmanSemantics.Properties.Rⁿ where

open import Function.Equivalence using (_⇔_; equivalence; module Equivalence)

open import Agda.Builtin.Nat using (Nat; suc; zero; _-_)
open import Agda.Builtin.Unit using (⊤; tt)
open import Agda.Primitive using (Level; lzero; lsuc)
open import Data.Empty using (⊥; ⊥-elim)
open import Data.Nat.Base using (_≤_; _<_; s≤s; z≤n)
open import Data.Fin using (Fin; zero; suc; fromℕ; inject₁; _ℕ-_; toℕ;
_ℕ-ℕ_; fromℕ<)
open import Data.Fin.Properties using (toℕ‿ℕ-)
open import Data.Nat.Properties using (≤-step; ≤-pred; ≤-reflexive;
≤-irrelevant; m∸n≤m;
  <⇒≤; <-cmp; _<?_; 1+n≰n; n≤1+n; ≰⇒>; _≟_; ≤-antisym; ≤-trans; _≤?_)
open import Data.Product renaming (_,_ to _,_)
open import Data.Sum using (_⊎_; inj₁; inj₂; [_,_])
open import Function using (_∘_; const; case_of_; id)
open import Function.Equality using (_⟨$⟩_)
open import Relation.Binary using (Irreflexive) renaming (Decidable to
Decidable₂)
open import Relation.Binary using (REL; Rel; Transitive; tri<; tri>;
tri≈)
open import Relation.Binary.PropositionalEquality using (_≡_; refl;
subst; trans; sym; cong;
  subst₂)
open import Relation.Nullary using (yes; no; ¬_)
open import Relation.Unary using (Pred; _∈_; _∉_; Decidable; Satisfiable;
_⊆_; _∩_; {_})
```

```
open import Formula
open import GeneralizedVeltmanSemantics
open import Base using (_⇒_; _⇐_; Decidable₃; subst₃)
open import GeneralizedVeltmanSemantics.Properties
  using (module Extended; ⊩¬; ⊮¬; ⊩□; ⊩◇; ⊩MP; ⊩∧; ⊩→¬⊮; ⊮→¬⊩)
open import GeneralizedVeltmanSemantics.Properties.R using (R-condition)
import Principles as P
open import Principles using (RⁿU)
open import GeneralizedFrame using (module Predicates)

private
  variable
    ℓW ℓR ℓS : Level

module _
  {W R S}
  (F : FrameL {lzero} {lzero} {lzero} W R S)
  where
  open FrameL F
  open Predicates F

  Rⁿ-condition : Nat → Set _
  Rⁿ-condition zero = R-condition F
  Rⁿ-condition (suc n) =
    ∀ {w y z : W} {𝔸 𝔹 ℂ : 𝕎} (x : (i : Nat) → {i < suc n} → W) (𝔻 :
(i : Nat) → i ≤ n → 𝕎)
    → R w (x n {s≤s (≤-reflexive refl)})
    → R (x 0 {s≤s z≤n}) y
    → R y z
    → z ∈ 𝔻 zero z≤n
    → ((i : Nat) → (i<n : i < n) → R (x (suc i) {s≤s i<n}) (x i {s≤s
(<⇒≤ i<n)}))
    → (∀ {u} → R w u → u ∈ 𝔸 → ∃[ V ] (S w u V × V ⊆ 𝔹))
   → (∀ {u} → R (x n {s≤s (≤-reflexive refl)}) u → u ∈ 𝔻 n (≤-reflexive
refl)
        → Σ 𝕎 λ V → S (x n {s≤s (≤-reflexive refl)}) u V × V ⊆ 𝔸)
    → ((i : Nat) → (i<n : i < n) → {u : W} → R (x i {s≤s (<⇒≤ i<n)}) u
      → u ∈ 𝔻 i (<⇒≤ i<n)
      → ∃[ V ] (S (x i {s≤s (<⇒≤ i<n)}) u V × V ⊆ 𝔻 (suc i) i<n))
    → (∀ {V} → S y z V → Satisfiable (V ∩ ℂ))
      → ∃[ V ] (V ⊆ 𝔹 × S w (x n {s≤s (≤-reflexive refl)}) V × R[ V ]
⊆ ℂ)


extend : ∀ {ℓ} {A : Set ℓ} {n} → A → (f : (i : Nat) → {i < n} → A) →
(i : Nat) → {i < suc n} → A
extend {l} {_} {n} x f i {p} with i <? n
... | yes z = f i {z}
... | no z = x

extend-last : ∀ {ℓ} {A : Set ℓ} {n} → (x : A) → (f : (i : Nat) → {i <
n} → A) → extend x f n {≤-reflexive refl} ≡ x
extend-last {_} {_} {n} x f with n <? n
... | yes z = ⊥-elim (1+n≰n z)
... | no z = refl

module Rⁿ-soundness
  {W R S V}
  {M : Model {lzero} {lzero} {lzero} W R S V}
  (M,_*⊩?_ : MultiDecidableModel M)
  (∈S? : Decidable₃ S)
  (S? : S-decidable (Model.F M))
  (∈SV? : ∀ {w u Y} → S w u Y → Decidable Y) where

  open Model M
  open FrameL F
  open Extended M,_*⊩?_ ∈S? ∈SV?

  ⊩Rⁿ : ∀ {n w} → Rⁿ-condition F (suc n) → P.Rⁿ (suc n) (M , w ⊩_)
  ⊩Rⁿ {n} {w} cond {A} {B} {C} D = ⊩⇝ ⇐ λ { w⊩A▷B → ⊩▷ ⇐
    λ { {x} Rwx x⊩∧ → case ⊩∧ ⇒ x⊩∧ of
    λ { (x⊩Un , x⊩Dn▷A) → case ⊩U n (≤-reflexive refl) x⊩Un of
      λ { (y , z , xs , Ryz , Rx₀y , Rxs , xsn≡x , xs⊩U , xs⊩D , ¬D▷¬C ,
zD)
       → case cond xs (λ {i p → [⊩ D i {s≤s p} ]}) (subst (R w) (sym
xsn≡x) Rwx)
        Rx₀y Ryz (∈[⊩ D 0 {s≤s z≤n} ] ⇐ subst (λ p → M , z ⊩ D 0 {p})
          (≤-irrelevant _ _) zD) Rxs
        (λ {Rwu uA → case (⊩▷ ⇒ w⊩A▷B) Rwu (∈[⊩ A ] ⇒ uA) of
        λ { (V , SwuV , V⊩B) → V , SwuV , λ {p → ∈[⊩ B ] ⇐ V⊩B p}}})
         (λ { {u} Rxu u∈𝔻 → case (⊩▷ ⇒ x⊩Dn▷A) (subst (λ x → R x u)
xsn≡x Rxu)
        (∈[⊩ D n {s≤s (≤-reflexive refl)} ] ⇒ u∈𝔻) of
         λ { (V , SwxV , V⊩A) → V , subst (λ x → S x u V) (sym xsn≡x)
SwxV ,
        λ {∈V → ∈[⊩ A ] ⇐ V⊩A ∈V}}})
        (λ {j j<n {u} Rxsju u⊩Di → case (⊩▷ ⇒ xs⊩D j j<n) Rxsju
        (∈[⊩ D j {_} ] ⇒ subst (λ {p → [⊩ D j {p}] u}) (≤-irrelevant _
_) u⊩Di) of
         λ { (V , SjuV , V⊩Dj) → V , SjuV , λ { {v} ∈V → ∈[⊩ D (suc j) ]
           ⇐ subst (λ p → M , v ⊩ D (suc j) {p}) (≤-irrelevant _ _)
(V⊩Dj ∈V)}}})
        (λ {SyzV → case ¬D▷¬C SyzV of
        ---
        λ { (c , snd , cC) → c , snd , (∈[⊩ C ] ⇐ cC)}})
        of λ { (V , V⊆𝔹 , Swx , RV⊆ℂ) → V , subst (λ x → S w x V) xsn≡x
Swx
        , λ {∈V → ⊩∧ ⇐ ((∈[⊩ B ] ⇒ V⊆𝔹 ∈V) , ⊩□ ⇐
        λ {Rxv → ∈[⊩ C ] ⇒ RV⊆ℂ (_ , ∈V , Rxv)})}}}}}}
    where
    ⊩U : ∀ {x} iu → (i≤n : iu ≤ n)
      → M , x ⊩ RⁿU n iu {i≤n} {C} {D}
```

```
       -----
       → Σ W λ y → Σ W λ z
       → Σ ((j : Nat) → {j < suc iu} → W) λ xs
       → R y z
         × (R (xs 0 {s≤s z≤n}) y)
         × (∀ j → (j<i : j < iu) → R (xs (suc j) {s≤s j<i}) (xs j {s≤s
(<⇒≤ j<i)}))
         × xs iu {≤-reflexive refl} ≡ x
        × (∀ j → (j≤i : j ≤ iu) → M , xs j {s≤s j≤i} ⊩ RⁿU n j {≤-trans
j≤i i≤n} {C} {D})
         × (∀ j → (j<i : j < iu) → M , xs j {s≤s (<⇒≤ j<i)} ⊩ D j {s≤s
(≤-trans (<⇒≤ j<i) i≤n)} ▷ D (suc j) {s≤s (≤-trans j<i i≤n)})
         × (∀ {V} → S y z V → Satisfiable (V ∩ (M ,_⊩ C)))
         × M , z ⊩ D 0 {s≤s z≤n}
       -----
    ⊩U {x} zero i≤n x⊩ = case ⊩◇ ⇒ x⊩ of
       λ { (y , Rxy , y⊩) → case ⊩◁ ⇒ y⊩ of λ { (z , Ryz , z⊩D , pD)
       → y , z , (λ {_ → x}) , Ryz , Rxy , (λ {j ()}) , refl ,
       (λ { .0 z≤n → subst (λ p → M , x ⊩ ◇ (¬' (D zero {p} ▷ ¬' C)))
         (≤-irrelevant _ _) x⊩ }) ,
         (λ {_ ()}) ,
         (λ {k → pD k }) , subst (λ p → M , z ⊩ D 0 {p})
           (≤-irrelevant _ _) z⊩D
       }}
     ⊩U {x} (suc i) i≤n x⊩ = case ⊩◇ ⇒ x⊩ of λ { (x' , Rxx' , x'⊩) →
case ⊩∧ ⇒ x'⊩ of
       λ { (x'⊩D , x'⊩U) → case ⊩U i (≤-pred (≤-step i≤n)) x'⊩U of
       λ { (y , z , xs , Ryz , Rx₀y , Rs , xsi≡x' , xs⊩U , xs⊩D , ¬D▷¬C ,
z⊩D)
       → y , z , extend x xs , Ryz , Rx₀y ,
         R-aux xs Rs (subst (R x) (sym xsi≡x') Rxx') ,
         extend-last x xs , ⊩U-aux xs xs⊩U , ⊩D-aux xs (subst (λ p → M
, p ⊩ D i ▷ D (suc i)) (sym xsi≡x') x'⊩D) xs⊩D ,
         (λ {z → ¬D▷¬C z}) , z⊩D
         }}}
         where
         ⊩D-aux : (xs : (j : Nat) → {j < suc i} → W)
           (as : M , xs i ⊩ D i ▷ D (suc i))
          (⊩D : (j : Nat) → (j<i : j < i) → M , xs j ⊩ D j ▷ D (suc j))
           (j : Nat) (j<i : j < suc i)
          → M , extend x xs j {s≤s (<⇒≤ j<i)} ⊩ D j ▷ D (suc j)
         ⊩D-aux xs xj⊩D ⊩D j j<si with j <? suc i
         ... | no a = ⊥-elim (a j<si)
         ... | yes _ with j <? i
         ... | yes j<i = subst₃ (λ p p1 p2 → M , xs j {p} ⊩ D j {p1} ▷
D (suc j) {p2})
              (≤-irrelevant _ _) (≤-irrelevant _ _) (≤-irrelevant _ _)
(⊩D j j<i)
         ... | no a with j ≟ i
         ... | yes refl = subst₃ (λ p p1 p2 → M , xs j {p} ⊩ D j {p1} ▷
D (suc j) {p2})
```

```
            (≤-irrelevant _ _) (≤-irrelevant _ _) (≤-irrelevant _ _)
xj⊩D
        ... | no y = ⊥-elim (y (≤-antisym (≤-pred j<si) (≤-pred (≰⇒>
a))))
       ⊩U-aux : (xs : (j : Nat) → {j < suc i} → W)
         (xs⊩U : (j : Nat) (j≤i : j ≤ i) → M , xs j {s≤s j≤i} ⊩ RⁿU n
j)
         (j : Nat) (j≤i : j ≤ suc i)
        → M , extend x xs j {s≤s j≤i} ⊩ RⁿU n j {≤-trans j≤i i≤n} {C}
{D}
       ⊩U-aux xs xs⊩U j j≤i with j <? suc i
       ... | yes y = subst₂ (λ p1 p2 → M , xs j {p1} ⊩ RⁿU n j {p2} )
(≤-irrelevant _ _) (≤-irrelevant _ _) (xs⊩U j (≤-pred y))
       ... | no y with j ≟ suc i
        ... | yes refl = subst (λ p → M , x ⊩ RⁿU n j {p} {C} {D})
(≤-irrelevant _ _) x⊩
       ... | no n = ⊥-elim (n (≤-antisym j≤i (≰⇒> (λ z → y (s≤s z)))))
      R-aux : (xs : (j : Nat) → {j < suc i} → W) (Rs : (j : Nat) (j<i
: j < i)
         → R (xs (suc j) {s≤s j<i}) (xs j {s≤s (<⇒≤ j<i)}))
         → R x (xs i {s≤s (≤-reflexive refl)})
         → (j : Nat) (j<i : j < suc i)
         → R (extend x xs (suc j) {s≤s j<i}) (extend x xs j {s≤s (<⇒≤
j<i)})
       R-aux xs Rs Rxx' j (s≤s j<i) with j <? suc i
       ... | (no b) = ⊥-elim (b (s≤s j<i))
       ... | (yes b) with suc j <? suc i
       ... | (yes a) = subst₂ R xs-irrel xs-irrel (Rs j (≤-pred a))
         where
         xs-irrel : ∀ {i p1 p2} → xs i {p1} ≡ xs i {p2}
         xs-irrel {i} {p1} {p2} = cong (λ p → xs i {p}) (≤-irrelevant
p1 p2)
       ... | (no a) with j ≟ i
       ... | (yes refl) = subst (λ p → R x (xs j {p})) (≤-irrelevant
_ _) Rxx'
        ... | (no n) = ⊥-elim (case ≰⇒> a of (λ { (s≤s (s≤s z)) → n
(≤-antisym j<i z)}))

module Rⁿ-completeness
  {W R S}
  {F : FrameL {lzero} {lzero} {lzero} W R S}
  (∈S? : Decidable₃ S)
  (dec : ∀ V → MultiDecidableModel (model {V = V} F))
  (∈SV? : ∀ {w u Y} → S w u Y → Decidable Y)
  where
  open FrameL F

  *⊩Rⁿ : Nat → Set₁
  *⊩Rⁿ n = P.Rⁿ n (F *⊩_)

  pattern a = 0
```

```
 pattern b = 1
 pattern c = 2
 pattern d i = suc (suc (suc i))


 ⊩Rⁿ⇒Rⁿ-condition : (n : Nat) → *⊩Rⁿ (suc n) → Rⁿ-condition F (suc n)
 ⊩Rⁿ⇒Rⁿ-condition n ⊩Rⁿ {w} {y} {z} {𝔸} {𝔹} {ℂ}
   xs 𝔻 Rwxn Rx0y Ryz z∈𝔻₀ Rxs w⊩𝔸▷𝔹 xn⊩D▷A xs⊩D y⊩D◁C
  = case (⊩▷ ⇒ ⊩MP (⊩Rⁿ D Val w) w⊩a▷b) Rwxn (⊩∧ ⇐ (xs⊩U n , xn⊩d▷a))
of
    λ { (V , SwxnV , V⊩b∧□c) → V , (λ {x → [b] ⇒ proj₁ (⊩∧ ⇒ V⊩b∧□c
x)})
    , SwxnV , (λ { (v , v∈V , Rvx) → [c] ⇒ (⊩□ ⇒ proj₂ (⊩∧ ⇒ V⊩b∧□c
v∈V)) Rvx})}
   where
   Val : Valuation F
   Val u a = u ∈ 𝔸
   Val u b = u ∈ 𝔹
   Val u c = u ∈ ℂ
   Val u (d j) with j ≤? n
   ... | yes j≤n = u ∈ 𝔻 j j≤n
   ... | no ¬j≤n = ⊥
   M = model {V = Val} F
   D : (j : Nat) {_ : j < suc n} → Fm
   D j = var (d j)
   open Extended (dec Val) ∈S? ∈SV?
   [a] : ∀ {u} → M , u ⊩ var a ⇔ u ∈ 𝔸
   [a] {u} = equivalence (λ { (var x) → x}) (λ x₁ → var x₁)
   [b] : ∀ {u} → M , u ⊩ var b ⇔ u ∈ 𝔹
   [b] {u} = equivalence (λ { (var x) → x}) (λ x₁ → var x₁)
   [c] : ∀ {u} → M , u ⊩ var c ⇔ u ∈ ℂ
   [c] {u} = equivalence (λ { (var x) → x}) (λ x₁ → var x₁)
   [d] : ∀ j {j≤n u} → M , u ⊩ var (d j) ⇔ u ∈ 𝔻 j j≤n
   [d] j {j≤n} {u} = equivalence ⇨ λ x → var (⇦ x)
    where
    ⇨ : M , u ⊩ var (d j) → 𝔻 j j≤n u
    ⇨ (var x) with j ≤? n
    ... | yes p = subst (λ p → 𝔻 j p u) (≤-irrelevant _ _) x
    ... | no _ = ⊥-elim x
    ⇦ : 𝔻 j j≤n u → d j ∈ Val u
    ⇦ x with j ≤? n
    ... | yes p = subst (λ p → 𝔻 j p u) (≤-irrelevant _ _) x
    ... | no p = ⊥-elim (p j≤n)
   w⊩a▷b : M , w ⊩ var a ▷ var b
   w⊩a▷b = [⊩▷] [a] [b] w⊩𝔸▷𝔹
   x⊩d▷a : M , xs n {≤-reflexive refl} ⊩ var (d n) ▷ var a
   x⊩d▷a = [⊩▷] ([d] n) [a] xn⊩D▷A
   xn⊩d▷a : M , xs n ⊩ var (d n) ▷ var a
   xn⊩d▷a = [⊩▷] ([d] n) [a] xn⊩D▷A
   xs⊩U : ∀ j {j<sn j≤n} → M , xs j {j<sn} ⊩ RⁿU n j {j≤n} {var c} {D}
   xs⊩U zero {j<sn} {j≤n} = ⊩◇ ⇐ (y , (subst (λ p → R (xs zero {p})
y) (≤-irrelevant _ _) Rx0y)
      , ⊩◁ ⇐ (z , Ryz , ([d] 0 ⇐ z∈𝔻₀) , λ {SyzV → case y⊩D◁C SyzV of
       λ { (v , v∈V , v∈ℂ) → v , v∈V , ([c] ⇐ v∈ℂ)}}))
    xs⊩U (suc j) {j<sn} {j≤n} = ⊩◇ ⇐ (xs j {s≤s (<⇒≤ j≤n)} , (subst₂
(λ p p' →
      R (xs (suc j) {p}) (xs j {p'})) (≤-irrelevant _ _) (≤-irrelevant
_ _) (Rxs j j≤n))
      , ⊩∧ ⇐ ([⊩▷] ([d] j) ([d] _) (xs⊩D j j≤n) , xs⊩U j))
```

B.16. GeneralizedVeltmanSemantics/Properties/R_1

The frame condition for the R^1 principle.

```
module GeneralizedVeltmanSemantics.Properties.R₁ where

open import Function.Equivalence using (_⇔_; equivalence; module Equivalence)

open import Agda.Builtin.Nat using (Nat; suc; zero)
open import Agda.Builtin.Unit using (⊤; tt)
open import Agda.Primitive using (Level; lzero; lsuc)
open import Data.Empty using (⊥; ⊥-elim)
open import Data.Product renaming (_,_ to _,_)
open import Data.Sum using (_⊎_; inj₁; inj₂; [_,_])
open import Function using (_∘_; const; case_of_; id)
open import Function.Equality using (_⟨$⟩_)
open import Relation.Binary using (REL; Rel; Transitive)
open import Relation.Nullary using (yes; no; ¬_)
open import Relation.Unary using (Pred; _∈_; _∉_; Decidable; Satisfiable;
_⊆_; _∩_; {_})
open import Relation.Binary using (Irreflexive) renaming (Decidable to
Decidable₂)
open import Relation.Binary.PropositionalEquality using (_≡_; refl;
subst; trans; sym)

open import Formula
open import GeneralizedVeltmanSemantics
open import Base using (_⇒_; _⇐_; Decidable₃)
open import GeneralizedVeltmanSemantics.Properties
  using (module Extended; ⊩¬; ⊮¬; ⊩□; ⊩◇; ⊩MP; ⊩∧; ⊩→¬⊮)
import Principles as P
open import GeneralizedFrame using (module Predicates)

private
  variable
    ℓW ℓR ℓS : Level

R₁-condition : ∀ {W R S} → FrameL {lzero} {lzero} {lzero} W R S → Set _
R₁-condition {W = W} {R = R} {S = S} F =
  ∀ {w x u V C 𝔼} → R w x → R x u → S w u V → IsChoiceSet C x u
  → ∃[ V' ] (V' ⊆ V × S w x V' × R[ V' ] ⊆ C ×
    (∀ {v c} → v ∈ V' → c ∈ C → (∃[ U ] (U ⊆ R⁻¹ x [ 𝔼 ] × R v c × S x
c U))
```

```
    → (∃[ 𝔼' ] (𝔼' ⊆ 𝔼 × S v c 𝔼'))))
  where open FrameL F
        open Predicates F

module R₁-soundness
  {W R S V}
  {M : Model {lzero} {lzero} {lzero} W R S V}
  (M,_*⊩?_ : MultiDecidableModel M)
  (∈S? : Decidable₃ S)
  (S? : S-decidable (Model.F M))
  (∈SV? : ∀ {w u Y} → S w u Y → Decidable Y) where

  open Model M
  open FrameL F
  open Extended M,_*⊩?_ ∈S? ∈SV?

  ⊩R₁ : ∀ {w A B C E D} → S-decidable F → R₁-condition F
    → M , w ⊩ A ▷ B ⇝ (A ◁ C ∧ D ▷ ◇ E) ▷ (B ∧ □ C ∧ D ▷ E)
  ⊩R₁ {w} {A} {B} {C} {E} S-dec c = ⊩⇝ ⇐ λ A▷B → ⊩▷ ⇐ λ {x} Rwx t →
case ⊩∧ ⇒ t of
    λ { (t1 , D▷◇E) → case ⊩¬ ⇒ t1 of λ { (rhd (u , Rxu , uA , A◁C))
    → case (⊩▷ ⇒ A▷B) (R-trans Rwx Rxu) uA of λ { (V , SwuV , VB) →
    case Swu-sat SwuV of λ { (_ , _) →
    case c {𝔼 = Γ} Rwx Rxu SwuV (𝒞 x u A◁C Rxu)
   of λ { (V' , V'⊆V , SwxV' , R[V']⊆Γ , cond) → V' , (SwxV' , λ { {v'}
v'∈V'
    → ⊩∧ ⇐ (VB (V'⊆V v'∈V') , (⊩∧ ⇐ (⊩□ ⇐ (λ {h} Rv'h →
    case ∈K x u h  ⇒ (R[V']⊆Γ (v' , v'∈V' , Rv'h)) of
    λ { (_ , _ , _ , hC) → hC})
    , ⊩▷ ⇐ λ { {c} Rv'c cD → let c∈Γ = R[V']⊆Γ (v' , v'∈V' , Rv'c) in
    case (⊩▷ ⇒ D▷◇E) (K⊆Rx x u c∈Γ) cD
   of λ { (U , SxcU , U◇E) → case cond v'∈V' c∈Γ (U , (λ {u} uU → case
⊩◇ ⇒ U◇E uU of
    λ { (e , Ru'e , e⊩E) → (e , (inE ⇐ e⊩E , Ru'e)) , SwuY⊆Rw SxcU uU
}) , Rv'c , SxcU) of
      λ { (Γ' , Γ'⊆Γ , Sv'cΓ') → Γ' , Sv'cΓ' , λ { {e} z → inE ⇒ Γ'⊆Γ
z}}}}})))})}}}}}
    where
      Γ : 𝕎
      Γ = [⊩ E ]
      inE : ∀ {e} → e ∈ Γ ⇔ (M , e ⊩ E )
      inE = ∈[⊩ E ]
      K : (x u : W) → 𝕎
      K x u y with S-dec x u y
      ... | inj₂ _ = ⊥
      ... | inj₁ _ with M, y ⊩? C
      ... | inj₁ _ = ⊤
      ... | inj₂ _ = ⊥
      ∈K : (x u y : W) → y ∈ K x u ⇔ Σ 𝕎 λ Z → y ∈ Z × S x u Z × M , y
⊩ C
      ∈K x u y = equivalence ⇒ ⇐
```

```
        where
        ⇨ : y ∈ K x u → Σ 𝕎 λ Z → y ∈ Z × S x u Z × M , y ⊩ C
        ⇨ _ with S-dec x u y
        ... | inj₁ (Y , yY , SxuY) with M, y ⊩? C
        ... | inj₁ x = Y , yY , SxuY , x
        ⇦ : (Σ 𝕎 λ Z → y ∈ Z × S x u Z × M , y ⊩ C) → y ∈ K x u
        ⇦ (Z , y∈Z , SxuZ , yC) with S-dec x u y
        ... | inj₂ p = p Z SxuZ y∈Z
        ... | inj₁ p with M, y ⊩? C
        ... | inj₁ _ = tt
        ... | inj₂ p' = ⊩→¬⊮ yC p'
      K⊆Rx : (x u : W) → K x u ⊆ R x
      K⊆Rx x u {z} z∈Γ = case ∈K x u z ⇒ z∈Γ of λ { (Z , z∈Z , SxuZ ,
_) → SwuY⊆Rw SxuZ z∈Z}
      𝒞 : (x u : W)
        → ((Y : W → Set) → Satisfiable Y → ¬ S x u Y ⊎ Satisfiable (Y
∩ (M ,_⊮ ¬' C)))
          → R x u → IsChoiceSet (K x u) x u
      𝒞 x u s Rxu = Rxu , (λ {Y} SxuY → case s Y (Swu-sat SxuY) of
        λ { (inj₁ x) → ⊥-elim (x SxuY) ; (inj₂ (c , c∈Y , c⊩C)) → c ,
c∈Y ,
        ∈K x u c ⇐ (Y , c∈Y , SxuY , ⊮¬ ⇒ c⊩C)})


module R₁-completeness
  {W R S}
  {F : FrameL {lzero} {lzero} {lzero} W R S}
  (∈S? : Decidable₃ S)
  (dec : ∀ V → MultiDecidableModel (model {V = V} F))
  (∈SV? : ∀ {w u Y} → S w u Y → Decidable Y)
  where
  open FrameL F
  open Predicates F

  *⊩R₁ : Set₁
  *⊩R₁ = P.R₁ (F *⊩_)

  pattern a = 0
  pattern b = 1
  pattern c = 2
  pattern d = 3
  pattern e = 4

  ¬R₁-condition : Set _
  ¬R₁-condition = Σ W λ w → Σ W λ x → Σ W λ u → Σ 𝕎 λ V → Σ 𝕎 λ C → Σ
𝕎 λ E →
    R w x × R x u × S w u V × IsChoiceSet C x u × E ⊆ R x ×
      let 𝒱 U = U ⊆ V × S w x U × R[ U ] ⊆ C in
      Σ (∀ {V' : 𝕎} → V' ∈ 𝒱 → Σ W λ vᵤ → Σ W λ cᵤ → Σ 𝕎 λ Zᵤ
      → vᵤ ∈ V' × cᵤ ∈ C × R vᵤ cᵤ × Zᵤ ⊆ R⁻¹ x [ E ] × S x cᵤ Zᵤ
        × (∀ 𝔼' → 𝔼' ⊆ E → ¬ S vᵤ cᵤ 𝔼'))
        λ p →
      let ∈𝒱⇒ : ∀ {U} → U ∈ 𝒱 → Σ W λ vu → Σ W λ cu → Σ 𝕎 λ Zu
              → R vu cu × Zu ⊆ R⁻¹ x [ E ] × S x cu Zu
             × (∀ 𝔼' → 𝔼' ⊆ E → ¬ S vu cu 𝔼')
          ∈𝒱⇒ {U} h = case p h of
            λ { (vu , cu , Zu , fst , snd , a1 , a2 , a3 , a4) → vu , cu
, Zu , a1 , (λ {x → a2 x}) ,
            a3 , a4}
          cᵤ : ∀ {U} → U ∈ 𝒱 → W
          cᵤ u = case ∈𝒱⇒ u of λ { (vu , cu , zu , snd) → cu}
     in
     (∀ (w : W) → (Σ 𝕎 λ U → Σ (U ∈ 𝒱) λ U∈𝒱 → cᵤ U∈𝒱 ≡ w)
      ⊎ (∀ {U} → (U∈𝒱 : U ∈ 𝒱) → ¬ (cᵤ U∈𝒱 ≡ w)))

  ⊩R₁⇒R₁-condition : *⊩R₁ → ¬ ¬R₁-condition
  ⊩R₁⇒R₁-condition R₁ (w , x , u , V , C , E , Rwx , Rxu , SwuV , (_ ,
choice) , E⊆Rx , p , t)
     = case ⊩MP ⊩r1 w⊩a▷b of λ { s → case (⊩▷ ⇒ s) Rwx (⊩∧ ⇐ (x⊩a◁c ,
x⊩d▷◊e)) of
    λ { (U , SwxU , U⊩b∧□c∧c▷d) →
      let
        U∈𝒱 : U ∈ 𝒱
        U∈𝒱 = ∈𝒱' SwxU
                (λ z → proj₁ (⊩∧ ⇒ U⊩b∧□c∧c▷d z))
                λ z → proj₁ (⊩∧ ⇒ proj₂  (⊩∧ ⇒ U⊩b∧□c∧c▷d z))
        vu∈U = proj₁ (proj₂ (proj₂ (proj₂ (p U∈𝒱))))
        Rvc = proj₁ (proj₂ (proj₂ (proj₂ (proj₂ (proj₂ (p U∈𝒱))))))
        ¬Svc𝔼 = proj₂ (proj₂ (proj₂ (proj₂ (proj₂ (proj₂ (proj₂ (proj₂
(p U∈𝒱))))))))
      in
        case (⊩▷ ⇒ proj₂ (⊩∧ ⇒ (proj₂ (⊩∧ ⇒ U⊩b∧□c∧c▷d vu∈U)))) Rvc
          ([d] ⇐ (U , U∈𝒱 , refl)) of
            λ { (𝔼' , Svc𝔼' , 𝔼'⊩E) → ¬Svc𝔼 𝔼' (λ {e∈𝔼' → [e] ⇒ 𝔼'⊩E
e∈𝔼'}) Svc𝔼'}}}
    where
    r1 : Fm
    r1 = (var a ▷ var b) ↝ (((var a ◁ var c) ∧ var d ▷ ◊ var e)
      ▷ (var b ∧ □ var c ∧ (var d ▷ var e)))
    𝒱 : Pred 𝕎 _
    𝒱 U = U ⊆ V × S w x U × R[ U ] ⊆ C
    Val : Valuation F
    Val i a = i ≡ u
    Val i b = i ∈ V
    Val i c = i ∈ C
    Val i e = i ∈ E
    Val i d with t i
    ... | inj₁ x = ⊤
    ... | inj₂ x = ⊥
    Val i (suc (suc (suc (suc (suc _))))) = ⊥
    M = model {V = Val} F
    ⊩r1 : M , w ⊩ r1
    ⊩r1 = R₁ Val w
    ∈𝒱 : ∀ {U} → U ∈ 𝒱 → Σ W λ vu → Σ W λ cu → Σ 𝕎 λ Zu
      → R vu cu × Zu ⊆ R⁻¹ x [ E ] × S x cu Zu
      × (∀ 𝔼' → 𝔼' ⊆ E → ¬ S vu cu 𝔼')
    ∈𝒱 {U} h with p h
    ... | (vu , cu , zu , fst , snd , a1 , a2 , a3) = vu , cu , zu , a1
, a2 , a3
    vᵤ : ∀ {U} → U ∈ 𝒱 → W
    vᵤ u = proj₁ (∈𝒱 u)
    cᵤ : ∀ {U} → U ∈ 𝒱 → W
    cᵤ u = proj₁ (proj₂ (∈𝒱 u))
    Zᵤ : ∀ {U} → U ∈ 𝒱 → 𝕎
    Zᵤ u = proj₁ (proj₂ (proj₂ (∈𝒱 u)))
    Rvᵤcᵤ : ∀ {U} → (U∈𝒱 : U ∈ 𝒱) → R (vᵤ U∈𝒱) (cᵤ U∈𝒱)
    Rvᵤcᵤ p = proj₁ (proj₂ (proj₂ (proj₂ (∈𝒱 p))))
    Zu⊆R⁻¹ₓ[E] : ∀ {U} → (U∈𝒱 : U ∈ 𝒱) → Zᵤ U∈𝒱 ⊆ R⁻¹ x [ E ]
    Zu⊆R⁻¹ₓ[E] U∈𝒱 = proj₁ (proj₂ (proj₂ (proj₂ (proj₂ (∈𝒱 U∈𝒱)))))
    SxcᵤZᵤ : ∀ {U} → (U∈𝒱 : U ∈ 𝒱) → S x (cᵤ U∈𝒱) (Zᵤ U∈𝒱)
   SxcᵤZᵤ U∈𝒱 = proj₁ (proj₂ (proj₂ (proj₂ (proj₂ (proj₂ (∈𝒱 U∈𝒱))))))
    open Extended (dec Val) ∈S? ∈SV?
    [a] : ∀ {y} → M , y ⊩ var a ⇔ y ≡ u
    [a] = equivalence (λ {(var x) → x}) (λ x₁ → var x₁)
    [b] : ∀ {y} → M , y ⊩ var b ⇔ y ∈ V
    [b] = equivalence (λ {(var z) → z}) λ x₁ → var x₁
    [c] : ∀ {y} → M , y ⊩ var c ⇔ y ∈ C
    [c] = equivalence (λ {(var z) → z}) λ x₁ → var x₁
    [e] : ∀ {y} → M , y ⊩ var e ⇔ y ∈ E
    [e] = equivalence (λ {(var z) → z}) λ x₁ → var x₁
    [d] : ∀ {y} → M , y ⊩ var d ⇔ Σ 𝕎 λ U → Σ (U ∈ 𝒱) λ U∈𝒱 → cᵤ U∈𝒱
≡ y
    [d] {y} = equivalence ⇒ λ x₁ → var (⇐ x₁)
      where
      ⇒ : M , y ⊩ var d → Σ 𝕎 (λ U → Σ (U ∈ 𝒱) (λ U∈𝒱 → cᵤ U∈𝒱 ≡ y))
      ⇒ (var x) with t y
      ... | inj₁ x₁ = x₁
      ⇐ : Σ 𝕎 (λ U → Σ (U ∈ 𝒱) (λ U∈𝒱 → cᵤ U∈𝒱 ≡ y)) → d ∈ Val y
      ⇐ x with t y
      ... | inj₁ x₁ = tt
      ⇐ (U , U∈𝒱 , ref) | inj₂ r = case r U∈𝒱 of λ {z → z ref}
    w⊩a⊳b : M , w ⊩ var a ⊳ var b
    w⊩a⊳b = ⊩⊳ ⇐ λ { {y} Rwy ya → case [a] ⇒ ya of λ { refl → V , SwuV
, λ {x → [b] ⇐ x}}}
    x⊩a⊲c : M , x ⊩ var a ⊲ var c
    x⊩a⊲c = ⊩¬ ⇐ (⊮⊳ ⇐ (u , (Rxu , ([a] ⇐ refl , λ {Y (s , s∈Y) SxuY →
      case choice SxuY of λ { (c' , c∈Y , c∈C) → c' , (c∈Y , ⊮¬ ⇐ ([c]
⇐ c∈C))}}))))
    𝒱⊩b : ∀ {U} → U ∈ 𝒱 → U ⊆ (M ,_⊩ var b)
    𝒱⊩b {U} (U⊆V , _) {u'} u'∈U = [b] ⇐ U⊆V u'∈U
    𝒱⊩□c : ∀ {U} → U ∈ 𝒱 → U ⊆ M ,_⊩ □ var c
     𝒱⊩□c {U} (_ , _ , R[U]⊆C) {u'} u'∈U = ⊩□ ⇐ λ { {z} Ru'z → [c] ⇐
R[U]⊆C (u' , u'∈U , Ru'z)}
```

```
    ∈𝒱' : ∀ {U} → S w x U → U ⊆ M ,_⊩ var b → U ⊆ M ,_⊩ □ var c → U ∈ 𝒱
    ∈𝒱' {U} SwxU Ub U□c = (λ z → [b] ⇒ (Ub z)) , SwxU , λ { (c' , c'∈U
, Rcy)
       → [c] ⇒ (⊩□ ⇒ U□c c'∈U) Rcy}
    x⊩d▷◊e : M , x ⊩ var d ▷ ◊ (var e)
    x⊩d▷◊e = ⊩▷ ⇐ λ { {y} Rxy y⊩d → case [d] ⇒ y⊩d of
      λ { (U , U∈𝒱 , refl) → Zᵤ U∈𝒱 , SxcᵤZᵤ U∈𝒱 ,
        λ { ∈Zᵤ → case Zu⊆R⁻¹ₓ[E] U∈𝒱 ∈Zᵤ of λ { ((e' , e'∈E , Rye) ,
Rxe') → ⊩◊ ⇐ (e' , Rye , [e] ⇐ e'∈E)}} }}
```

B.17. GeneralizedVeltmanSemantics/Properties/Verbrugge

```
module GeneralizedVeltmanSemantics.Properties.Verbrugge where

open import Function.Equivalence using (_⇔_; equivalence; module Equivalence)

open import Agda.Builtin.Nat using (Nat; suc; zero)
open import Agda.Builtin.Unit using (⊤; tt)
open import Agda.Primitive using (Level; lzero; lsuc; _⊔_)
open import Data.Empty using (⊥; ⊥-elim)
open import Function using (_$_)
open import Data.Product renaming (_,_ to _,_)
open import Data.Sum using (_⊎_; inj₁; inj₂; [_,_])
open import Function using (_∘_; const; case_of_; id)
open import Function.Equality using (_⟨$⟩_)
open import Relation.Binary using (REL; Rel; Transitive)
open import Relation.Nullary using (yes; no; ¬_)
open import Relation.Unary using (Pred; _∈_; _∉_; Decidable; Satisfiable;
_⊆_; _∩_; {_}; ∅)
open import Relation.Binary using (Irreflexive) renaming (Decidable to
Decidable₂)
open import Relation.Binary.PropositionalEquality using (_≡_; refl;
subst; trans; sym; subst₂)

open import Formula using (Fm; Var; _↝_; ⊥'; _▷_; _◁_; var; ⊤'; ¬'_;
□_; ◊_; _∧_; _∨_; car)
open import Base
open import GeneralizedVeltmanSemantics.Properties
  using (module SemanticsProperties-4;
    module SemanticsProperties-3;
    module SemanticsProperties-L; module PGeneric)
open import GeneralizedFrame
open import GeneralizedFrame.Properties
import OrdinaryFrame as O
import OrdinaryVeltmanSemantics as O
import OrdinaryVeltmanSemantics.Properties as O

private
  variable
    ℓW ℓR ℓS : Level
```

```
module OrdModel
  {ℓW ℓR ℓS}
  (T : ∀ {ℓW ℓS} → (W : Set ℓW) → REL₃ W W (Pred W ℓW) ℓS → Set (lsuc
ℓW ⊔ ℓS))
  {W R S}
  (F : Frame {ℓW} {ℓR} {ℓS} W R S T)
  (V : W → Pred Var lzero)
  where
  open Frame F
  open PGeneric T

  W' : Set _
  W' = ∃[ x ]
        (Σ[ A ∈ (Pred (W × W) (lsuc ℓW ⊔ ℓS ⊔ ℓR)) ]
        ((∀ {u v} → (u , v) ∈ A → ∃[ Y ] (S u x Y × v ∈ Y))
        × (∀ {u V} → S u x V → ∃[ v ] (v ∈ V × (u , v) ∈ A))))

  Amax : W → Pred (W × W) _
  Amax x (w , v) = Σ 𝕎 λ C → S w x C × v ∈ C × (⊥ → R v v)

  R' : Rel W' _
  R' (x , A , _) (y , B , _) = R x y × (∀ {w z} → R w x → (w , z) ∈ B →
(w , z) ∈ A)

  S' : Rel₃ W' _
  S' w'@(w , C , _) x'@(x , A , _) y'@(y , B , _) =
    R' w' x' × R' w' y' × (∀ {v} → (w , v) ∈ B → (w , v) ∈ A)

  fmax : W → W'
  fmax w = w , Amax w , (λ { (C , SuwC , v∈C , _) → C , SuwC , v∈C})
    , λ {SuwV → (proj₁ (Swu-sat SuwV)) , ((proj₂ (Swu-sat SuwV)) ,
    (_ , (SuwV , (proj₂ (Swu-sat SuwV) , ⊥-elim))))}

  module W' where
    x : W' → W
    x = proj₁

    A : W' → Pred (W × W) _
    A = proj₁ ∘ proj₂

    p1 : (w' : W')
       → (∀ {u v} → (u , v) ∈ (A w') → Σ 𝕎 λ Y → S u (proj₁ w') Y × v
∈ Y)
    p1 = proj₁ ∘ proj₂ ∘ proj₂

    p2 : (w' : W') → ∀ {u V} → S u (x w') V → Σ W λ v → v ∈ V × (u , v)
∈ (A w')
    p2 = proj₂ ∘ proj₂ ∘ proj₂

    x∘f≡id : ∀ w → x (fmax w) ≡ w
    x∘f≡id w = refl

  V' : W' → Pred Var lzero
  V' w' v = v ∈ V (W'.x w')

  f-chain : ∀ {a} → InfiniteChain R' a → InfiniteChain R _
  InfiniteChain.b (f-chain x) = W'.x (InfiniteChain.b x)
  InfiniteChain.a<b (f-chain x) = proj₁ (InfiniteChain.a<b x)
  InfiniteChain.tail (f-chain x) = f-chain (InfiniteChain.tail x)

  R'-Noetherian : Noetherian R'
  R'-Noetherian i = R-noetherian (f-chain i)

  F' : O.Frame W' R' S'
  F' = O.frame
    (fmax witness)
    (λ { {u} {v} (x , A) (y , B) → R-trans x y ,
    λ { {v1} {v2} Rwx wz∈A → A Rwx (B (R-trans Rwx x) wz∈A)}})
    R'-Noetherian
    (λ { (fst , fst₁ , snd) → fst , fst₁})
    (λ { {w} {z} (Rwz , snd) →
    (Rwz , λ { {u} {v} Ruw W'Auv → snd Ruw W'Auv}) , (Rwz , snd) , λ z
→ z})
    (λ { ((Rwi , snd1) , _ , snd) (_ , (a , snd3))
      → (Rwi ,
      λ {x x₁ → snd1 x x₁}) , a , λ { x → snd (snd3 x)}})
    λ { (fst , snd) (fst₁ , B) → (fst , snd) ,
    (R-trans fst fst₁ , λ {x y → snd x (B (R-trans x fst) y)}) , B fst}

  M' : O.Model W' R' S' V'
  M' = O.model F'

module PrefaceTheoremAll
  {ℓW ℓR ℓS}
  (T : ∀ {ℓW ℓS} → (W : Set ℓW) → REL₃ W W (Pred W ℓW) ℓS → Set (lsuc
ℓW ⊔ ℓS))
  {W R S}
  (F : Frame {ℓW} {ℓR} {ℓS} W R S T)
  (V : W → Pred Var lzero)
  where
  open OrdModel T F V
  open Frame F
  open PGeneric T

  M : Model W R S V
  M = model {V = V} F

  lemma⇛-type : Set _
  lemma⇛-type =
    ∀ {w x Y} → S w x Y →
    Σ _ λ y → y ∈ Y
    × (∀ {b V} → S b y V
      → Σ _ λ v → v ∈ V
      -- (1)
      × (b ≡ w → S b x ｛ v ｝)
      -- (2)
      × (R b w → S b w ｛ v ｝))

  lemma⇦-type : Set _
  lemma⇦-type =
    ∀ {w b x V} → R w x → S b x V →
    Σ _ λ v → v ∈ V
      -- (1)
      × (b ≡ w → S b x ｛ v ｝)
      -- (2)
      × (R b w → S b w ｛ v ｝)

  module Theorem
    (dec : MultiDecidableModel M)
    (dec' : O.DecidableModel M')
    (∈S? : Decidable₃ S)
    (∈S'? : Decidable₃ S')
    (∈SV? : ∀ {w u Y} → S w u Y → Decidable Y)
    (lemma⇨ : lemma⇨-type)
    (lemma⇦ : lemma⇦-type)
    where

    private
      open Extended dec ∈S? ∈SV?
      module O' = O.Extended dec' ∈S'?

      thm⇨ : ∀ {w C A} → M , w ⊩ A → M' O., w , C ⊩ A
      thm⇦ : ∀ {w C A} → M' O., w , C ⊩ A → M , w ⊩ A
      thm'⇨ : ∀ {w C A} → M , w ⊮ A → M' O., w , C ⊮ A

      thm⇦ x = ⊩⇔¬⊮ ⇐ λ {y → O.⊮→¬⊩ (thm'⇨ y) x}

      thm⇨ (var x) = O.var x
      thm⇨ A@(impl _) = O'.⊩↝ ⇐ λ {wA → thm⇨ ((⊩↝ ⇒ A) (thm⇦ wA))}
      thm⇨ {w} {C'@(C , Cp1 , Cp2)} F@(rhd {D} {E} xl) = O'.⊩▷ ⇐
        λ { {xA} R'wx@(Rwx , Rwx∈) x⊩D
          → case (⊩▷ ⇒ F) Rwx (thm⇦ x⊩D) of
          λ { (V , SwxV , V⊩E) →
          let
              l = lemma⇨ SwxV
              y = proj₁ l
              y∈V = (proj₁ ∘ proj₂) l
              lemma2 = (proj₂ ∘ proj₂) l
              x = W'.x xA
              Rwy : R w y
              Rwy = SwuY⊆Rw SwxV y∈V
              A = W'.A xA
```

```
              B = λ { (u , v) → (Σ _ λ Y
                → S u y Y × v ∈ Y) × (u ≡ w → (w , v) ∈ A) × (R u w →
(u , v) ∈ C) }
              yB∈W' : W'
              yB∈W' = y , B ,
                (λ { {u} {v} ((Y , SuyY , v∈Y) , _) → Y , SuyY , v∈Y}) ,
                 λ { {b} {V} SbyV →
                   let l2 = lemma2 SbyV
                       v = proj₁ l2
                       v∈V = (proj₁ ∘ proj₂) l2
                       lemma-1 = (proj₁ ∘ proj₂ ∘ proj₂) l2
                       lemma-2 = (proj₂ ∘ proj₂ ∘ proj₂) l2
                   in _ , v∈V , (_ , (SbyV , v∈V))
                 , (λ { refl → let
                         Swxv : S w x { v }
                         Swxv = lemma-1 refl
                        in case W'.p2 xA Swxv of λ { (_ , refl , snd) →
snd} })
                 , λ { Rbw → let
                               Sbwz : S b w { v }
                               Sbwz = lemma-2 Rbw
                               bv∈C : (b , v) ∈ C
                               bv∈C = case Cp2 Sbwz of
                                 λ { (_ , refl , snd) → snd}
                              in bv∈C} }
              l1 : ∀ {v} → (w , v) ∈ B → (w , v) ∈ A
              l1 wv∈B = proj₁ (proj₂ wv∈B) refl
              l2 : ∀ {b z} → R b w → (b , z) ∈ B → (b , z) ∈ C
              l2 {b} {z} Rbw bz∈B = (proj₂ ∘ proj₂) bz∈B Rbw
              R'wy : R' (w , C') yB∈W'
              R'wy = Rwy , (λ {Rbw ∈B → l2 Rbw ∈B})
          in
            yB∈W' , (R'wx , R'wy , λ {z → l1 z})
            , thm⇒ (V⊩E y∈V)
            }}

      thm'⇒ (var x) = O.var x
      thm'⇒ {w} {C} (impl {A} {B} a b) = O.impl (thm⇒ a) (thm'⇒ b)
      thm'⇒ bot = O.bot
     thm'⇒ {w} {C'@(C , Cp1 , Cp2)} F@(rhd {D} {E} _) = case ⊩▷ ⇒ F of
        λ { (x , Rwx , x⊩D , px)
        → let A = λ { (u , v) →
                (Σ _ λ Y → S u x Y × v ∈ Y) × (u ≡ w → M , v ⊮ E) × (R
u w → (u , v) ∈ C)}
              xA∈W' : W'
              xA∈W' = x , A ,
                (λ { ((V , SuxV , v∈V) , _) → _ , SuxV , v∈V}) ,
                λ { {b} {V} SbxV →
                  let l = lemma⇐ Rwx SbxV
                      v = proj₁ l
                      v∈V : v ∈ V
                      v∈V = proj₁ ∘ proj₂ $ l
                      l1 = proj₁ ∘ proj₂ ∘ proj₂ $ l
                      l2 = proj₂ ∘ proj₂ ∘ proj₂ $ l
                  in
                  _ , (v∈V , (_ , (SbxV , v∈V))
                , (λ { refl →
                let
                  Swxv : S b x { v }
                  Swxv = l1 refl
                in case px _ (_ , refl) Swxv of
                  λ { (_ , refl , l) → l}
                }) , λ { Rbw →
                  let
                    Suwv : S b w { v }
                    Suwv = l2 Rbw
                    uv∈C : (b , v) ∈ C
                    uv∈C = case Cp2 Suwv of λ { (_ , refl , p) → p}
                  in uv∈C} )}
              R'wx : R' (w , C') xA∈W'
               R'wx = Rwx , (λ { {b} Rbw bz∈A → (proj₂ ∘ proj₂) bz∈A
Rbw})
           in O'.⊮▷ ⇐ (xA∈W' , R'wx , thm⇒ x⊩D , λ { {yB@(y , B , B1 ,
B2)}
            S'wxy →
            let
                Rwy : R w y
                Rwy = (proj₁ ∘ proj₁ ∘ proj₂) S'wxy
                Swyy : S w y { y }
                Swyy = S-quasirefl Rwy
                wy∈B : (w , y) ∈ B
                wy∈B = case B2 Swyy of
                  λ { (_ , refl , snd) → snd}
                wy∈A : (w , y) ∈ A
                wy∈A = (proj₂ ∘ proj₂) S'wxy wy∈B
            in thm'⇒ ((proj₁ ∘ proj₂) wy∈A refl)})}

    theorem : ∀ {w C A} → M , w ⊩ A ⇔ M' O., w , C ⊩ A
    theorem = equivalence thm⇒ thm⇐

module Theorem-4
  {W R S}
  (F : Frame4 {ℓW} {ℓR} {ℓS} W R S)
  (V : W → Pred Var lzero)
  where
  open Frame4 F
  open SemanticsProperties-4

  open Trans-conditions using (Trans-4)
  open OrdModel {ℓW} {ℓR} {ℓS} Trans-4 F V
  open PrefaceTheoremAll Trans-4 F V
```

```
lemma⇐ : lemma⇐-type
lemma⇐ {w} {b} {x} {V} Rwx SbxV =
  let
    qt = quasitrans SbxV
    v = proj₁ qt
    v∈V = proj₁ (proj₂ qt)
    transv : ∀ {Y} → S b v Y → S b x Y
    transv = proj₂ (proj₂ qt)
  in v , v∈V
    -- (1)
    , (
    let
      Rbv : R b v
      Rbv = SwuY⊆Rw SbxV v∈V
      Swvv : S b v ⦃ v ⦄
      Swvv = S-quasirefl Rbv
      Swxv : S b x ⦃ v ⦄
      Swxv = transv Swvv
      in const Swxv)
    -- (2)
    , λ { Rbw →
      let
        Ruv : R b v
        Ruv = SwuY⊆Rw SbxV v∈V
        Suvv : S b v ⦃ v ⦄
        Suvv = S-quasirefl Ruv
        Suxv : S b x ⦃ v ⦄
        Suxv = transv Suvv
        Suwx : S b w ⦃ x ⦄
        Suwx = R-Sw-trans Rbw Rwx
        transx : ∀ {Y} → S b x Y → S b w Y
        transx {Y} = case quasitrans Suwx of
          λ { (_ , refl , p) → p}
        Sbwv : S b w ⦃ v ⦄
        Sbwv = transx Suxv
      in Sbwv}

lemma⇒ : lemma⇒-type
lemma⇒ {w} {x} {Y} SwxY =
  let
    qt = quasitrans SwxY
    y = proj₁ qt
    y∈V = (proj₁ ∘ proj₂) qt
    Rwy : R w y
    Rwy = SwuY⊆Rw SwxY (proj₁ (proj₂ qt))
    transy : ∀ {V} → S w y V → S w x V
    transy = (proj₂ ∘ proj₂) qt
  in y , y∈V ,
    (λ {b} {V} SbyV →
  let v∈V = (proj₁ ∘ proj₂) (quasitrans SbyV)
      v = proj₁ (quasitrans SbyV)
```

```
        transv : ∀ {V} → S b v V → S b y V
        transv = (proj₂ ∘ proj₂) (quasitrans SbyV)
    in v , v∈V ,
      -- (1)
      (λ {refl →
        let
            SwxV : S w x V
            SwxV = transy SbyV
            Rwv : R w v
            Rwv = SwuY⊆Rw SwxV v∈V
            Swvv : S w v { v }
            Swvv = S-quasirefl Rwv
            Swxv : S w x { v }
            Swxv = (transy ∘ transv) Swvv
        in Swxv
        })
      -- (2)
      , λ { Rbw →
        let
          Sbyz : S b w { y }
          Sbyz = R-Sw-trans Rbw Rwy
          transy : ∀ {Y} → S b y Y → S b w Y
          transy {Y} = case quasitrans Sbyz of
            λ { (_ , refl , a) → a {Y}  }
          SbwY : S b w V
          SbwY = transy SbyV
          transv : ∀ {V} → S b v V → S b y V
          transv = (proj₂ ∘ proj₂) (quasitrans SbyV)
          Sbvv : S b v { v }
          Sbvv = S-quasirefl (SwuY⊆Rw SbwY v∈V)
          Sbwv : S b w { v }
          Sbwv = (transy ∘ transv) Sbvv
        in Sbwv
      })

  module _
    (dec : MultiDecidableModel M)
    (dec' : O.DecidableModel M')
    (∈S? : Decidable₃ S)
    (∈S'? : Decidable₃ S')
    (∈SV? : ∀ {w u Y} → S w u Y → Decidable Y)
    where
    open Theorem dec dec' ∈S? ∈S'? ∈SV? lemma⇒ lemma⇐


module Theorem-3
  {W R S}
  (F : Frame3 {ℓW} {ℓR} {ℓS} W R S)
  (V : W → Pred Var lzero)
  where
  open Frame3 F
  open SemanticsProperties-3

  open Trans-conditions using (Trans-3)
  open OrdModel {ℓW} {ℓR} {ℓS} Trans-3 F V
  open PrefaceTheoremAll Trans-3 F V
  open FrameProperties Trans-3 F

  lemma⇐ : lemma⇐-type
  lemma⇐ {w} {b} {x} {V} Rwx SbxV =
    let
      qt = quasitrans SbxV
      v = proj₁ qt
      v∈V = proj₁ (proj₂ qt)
      transv : ∀ {Y} → S b v Y → Σ 𝕎 λ V' → V' ⊆ Y × S b x V'
      transv = proj₂ (proj₂ qt)
    in v , v∈V
      -- (1)
      , (
      let
        Rbv : R b v
        Rbv = SwuY⊆Rw SbxV v∈V
        Swvv : S b v ⦃ v ⦄
        Swvv = S-quasirefl Rbv
        Swxv : S b x ⦃ v ⦄
        Swxv = case transv Swvv of
          λ { (v' , v'⊆⦃v⦄ , l) → S-ext l v'⊆⦃v⦄
          λ { refl → case Swu-sat l of
          λ { (_ , snd) → case v'⊆⦃v⦄ snd of λ {refl → snd}}}}
        in const Swxv)
      -- (2)
      , λ { Rbw →
        let
          Ruv : R b v
          Ruv = SwuY⊆Rw SbxV v∈V
          Suvv : S b v ⦃ v ⦄
          Suvv = S-quasirefl Ruv
          Suxv : S b x ⦃ v ⦄
          Suxv = uncurry S⊆⦃v⦄ (proj₂ (transv Suvv))
          Suwx : S b w ⦃ x ⦄
          Suwx = R-Sw-trans Rbw Rwx
          transx : ∀ {Y} → S b x Y → Σ 𝕎 λ Y' → Y' ⊆ Y × S b w Y'
          transx {Y} = case quasitrans Suwx of
            λ { (_ , refl , p) → p}
          Sbwv : S b w ⦃ v ⦄
          Sbwv = uncurry S⊆⦃v⦄ (proj₂ (transx Suxv))
        in Sbwv}

  lemma⇒ : lemma⇒-type
  lemma⇒ {w} {x} {Y} SwxY =
    let
      qt = quasitrans SwxY
```

```
      y = proj₁ qt
      y∈V = (proj₁ ∘ proj₂) qt
      Rwy : R w y
      Rwy = SwuY⊆Rw SwxY (proj₁ (proj₂ qt))
      transy : ∀ {u} → S w y { u } → S w x { u }
      transy Sbvy = uncurry S⊆{v} ∘ proj₂ ∘ (proj₂ ∘ proj₂) qt $ Sbvy
    in y , y∈V ,
      (λ {b} {V} SbyV →
    let v∈V = (proj₁ ∘ proj₂) (quasitrans SbyV)
        v = proj₁ (quasitrans SbyV)
        transv : ∀ {u} → S b v { u } → S b y { u }
       transv Sbvy = uncurry S⊆{v} ∘ proj₂ ∘ (proj₂ ∘ proj₂ $ quasitrans
SbyV) $ Sbvy
    in v , v∈V ,
      -- (1)
      (λ {refl →
        let
            Rwv : R w v
            Rwv = SwuY⊆Rw SbyV v∈V
            Swvv : S w v { v }
            Swvv = S-quasirefl Rwv
            Swxv : S w x { v }
            Swxv = (transy ∘ transv) Swvv
        in Swxv
        })
      -- (2)
      , λ { Rbw →
        let
          Sbyz : S b w { y }
          Sbyz = R-Sw-trans Rbw Rwy
          transy : ∀ {u} → S b y { u } → S b w { u }
          transy {Y} Sbyu = case quasitrans Sbyz of
            λ { (_ , refl , a) → uncurry S⊆{v} ∘ proj₂ $ a Sbyu }
          Rbv : R b v
          Rbv = SwuY⊆Rw SbyV v∈V
          Sbvv : S b v { v }
          Sbvv = S-quasirefl Rbv
          Sbwv : S b w { v }
          Sbwv = (transy ∘ transv) Sbvv
        in Sbwv
      })

  module _
    (dec : MultiDecidableModel M)
    (dec' : O.DecidableModel M')
    (∈S? : Decidable₃ S)
    (∈S'? : Decidable₃ S')
    (∈SV? : ∀ {w u Y} → S w u Y → Decidable Y)
    where
    open Theorem dec dec' ∈S? ∈S'? ∈SV? lemma⇒ lemma⇐
```

B.18. GeneralizedVeltmanSemantics/Properties/Vukovic

This file contains the unfinished proof of proposition 2.8 in [41].

```
module GeneralizedVeltmanSemantics.Properties.Vukovic where

open import Function.Equivalence using (_⇔_; equivalence; module Equivalence)

open import Agda.Builtin.Nat using (Nat; suc; zero)
open import Agda.Builtin.Unit using (⊤; tt)
open import Agda.Primitive using (Level; lzero; lsuc; _⊔_)
open import Data.Empty using (⊥; ⊥-elim)
open import Function using (_$_)
open import Data.Product renaming (_,_ to _,_)
open import Data.Sum using (_⊎_; inj₁; inj₂; [_,_])
open import Function using (_∘_; const; case_of_; id)
open import Function.Equality using (_⟨$⟩_)
open import Relation.Binary using (REL; Rel; Transitive)
open import Relation.Nullary using (yes; no; ¬_)
open import Relation.Unary using (Pred; _∈_; _∉_; Decidable; Satisfiable;
_⊆_; _∩_; {_}; ∅)
open import Relation.Binary using () renaming (Decidable to Decidable₂)
open import Relation.Binary.PropositionalEquality using (_≡_; refl;
subst; trans; sym; subst₂; _≢_)

open import Formula using (Fm; Var; _↝_; ⊥'; _▷_; _◁_; var; ⊤'; ¬'_;
□_; ◊_; _∧_; _∨_; car)
open import Base
open import GeneralizedVeltmanSemantics.Properties using (module SemanticsProperti
4; module SemanticsProperties-L; module PGeneric)
open import GeneralizedFrame
open import GeneralizedFrame.Properties
import OrdinaryFrame as O
import OrdinaryVeltmanSemantics as O
import OrdinaryVeltmanSemantics.Properties as O

private
  variable
    ℓW ℓR ℓS : Level

module OrdModel
  {ℓW ℓR ℓS}
  (T : ∀ {ℓW ℓS} → (W : Set ℓW) → REL₃ W W (Pred W ℓW) ℓS → Set (lsuc
ℓW ⊔ ℓS))
  {W R S}
  (F : Frame {ℓW} {ℓR} {ℓS} W R S T)
  (V : W → Pred Var lzero)
  where
  open Frame F
  open PGeneric T

  W' : Set (lsuc (lsuc ℓW ⊔ ℓR ⊔ ℓS))
  W' =   ∃[ v ]
       (Σ[ A ∈ (Pred (W × W) (lsuc ℓW ⊔ ℓS ⊔ ℓR)) ]
       ((∀ {x V} → S x v V → Σ[ V' ∈ 𝕎 ]
         (∃[ y ] (y ∈ V' × V' ⊆ R x × V ⊆ V' × (x , y) ∈ A)))
       × (∀ {x y} → (x , y) ∈ A → ∃[ V ] (S x v V × y ∈ V))
       × (∀ {u} → R u v → (u , v) ∈ A)))

  R' : Rel W' (lsuc ℓW ⊔ ℓR ⊔ ℓS)
  R' (w , A , _) (u , B , _) = R w u × (∀ {x} → R x w → ∀ {y} → (x , y)
∈ B → (x , y) ∈ A)

  S' : Rel₃ W' (lsuc ℓW ⊔ ℓR ⊔ ℓS)
  S' w'@(w , A , _) u'@(u , B , _) v'@(v , C , _) =
    R' w' u' × R' w' v' × (∀ {y} → (w , y) ∈ C → (w , y) ∈ B)

  module W' where
    v : W' → W
    v = proj₁

    A : W' → Pred (W × W) _
    A = proj₁ ∘ proj₂

    p1 : (w' : W')
      → ∀ {x V} → S x (v w') V
      → ∃[ V' ] (∃[ y ]
      (y ∈ V' × V' ⊆ R x × V ⊆ V' × (x , y) ∈ A w'))
    p1 = proj₁ ∘ proj₂ ∘ proj₂

    p2 : (w' : W') → ∀ {x y} → (x , y) ∈ A w'
      → Σ 𝕎 λ V → S x (v w') V × y ∈ V
    p2 = proj₁ ∘ proj₂ ∘ proj₂ ∘ proj₂

    p3 : (w' : W') → ∀ {x} → R x (v w') → (x , v w') ∈ A w'
    p3 = proj₂ ∘ proj₂ ∘ proj₂ ∘ proj₂

  Amax : W → Pred (W × W) (lsuc ℓW ⊔ ℓS ⊔ ℓR)
  Amax w (x , y) = Σ 𝕎 (λ V → S x w V × V y × (⊥ → R x x))

  fmax : W → W'
  fmax w = w , Amax w ,
       (λ { {x} {V} SxwV → V , (proj₁ (Swu-sat SxwV) , (proj₂ (Swu-sat
SxwV)
         , ((λ {∈V → SwuY⊆Rw SxwV ∈V}) , ((λ z → z) , (_ , SxwV
         , proj₂ (Swu-sat SxwV) , ⊥-elim)))))})
    , (λ {(a , b , c , _) → a , b , c})
    , λ { Ruw → _ , S-quasirefl Ruw , refl , ⊥-elim}

  Val' : W' → Pred Var lzero
  Val' w' v = v ∈ V (W'.v w')

  f-chain : ∀ {a} → InfiniteChain R' a → InfiniteChain R _
```

```
    InfiniteChain.b (f-chain x) = W'.v (InfiniteChain.b x)
    InfiniteChain.a<b (f-chain x) = proj₁ (InfiniteChain.a<b x)
    InfiniteChain.tail (f-chain x) = f-chain (InfiniteChain.tail x)

    R'-Noetherian : Noetherian R'
    R'-Noetherian i = R-noetherian (f-chain i)

    F' : O.Frame W' R' S'
    F' = O.frame
      (fmax witness)
      (λ { (Rab , p1) (Rbc , p2) → (R-trans Rab Rbc) ,
        (λ {Rxi z → p1 Rxi (p2 (R-trans Rxi Rab) z) })})
      R'-Noetherian
      (λ { (R'wu , R'wv , _) → R'wu , R'wv})
      (λ {R'wu → R'wu , R'wu , (λ {x → x})})
      (λ { (a , (pij , b)) ((_ , rjk) , (c , pjk)) → a , c
        , λ {x → b (pjk x)}})
      λ {R'wu@(Rwu , pwu) R'uv@(Ruv , puv) → R'wu , (R-trans Rwu Ruv ,
        λ { {b} Rbw by∈ → pwu Rbw (puv (R-trans Rbw Rwu) by∈)})
      , λ {x → puv Rwu x}}

    M' : O.Model W' R' S' Val'
    M' = O.model F'

  module PrefaceTheoremAll
    {ℓW ℓR ℓS}
    {W R S}
    (F : FrameL {ℓW} {ℓR} {ℓS} W R S)
    (V : W → Pred Var lzero)
    where
    T = Trans-conditions.Trans-L
    open OrdModel T F V
    open Frame F
    open PGeneric T

    M : Model W R S V
    M = model {V = V} F

    module Theorem
      (dec : MultiDecidableModel M)
      (dec' : O.DecidableModel M')
      (R? : Decidable₂ R)
      (∈S? : Decidable₃ S)
      (∈S'? : Decidable₃ S')
      (∈SV? : ∀ {w u Y} → S w u Y → Decidable Y)
      (_≡?_ : Decidable₂ (_≡_ {_} {W}))
      (monotone : ∀ {w u} {V Z : 𝕎} → S w u V → V ⊆ Z → Z ⊆ R w → S w u Z)
      (quasitrans : Trans-conditions.Trans-L W S)
      where

      private
```

```
 open Extended dec ∈S? ∈SV?
 module O' = O.Extended dec' ∈S'?

 thm⇒ : ∀ {w C A} → M , w ⊩ A → M' O., w , C ⊩ A
 thm⇐ : ∀ {w C A} → M' O., w , C ⊩ A → M , w ⊩ A
 thm'⇒ : ∀ {w C A} → M , w ⊮ A → M' O., w , C ⊮ A

 thm⇐ x = ⊩⇔¬⊮ ⇐ λ {y → O.⊮→¬⊩ (thm'⇒ y) x}

 module F
   (V0 V : 𝕎)
   (v0 u w : W)
   (v0∈V0 : v0 ∈ V0)
   (SwuV0 : S w u V0)
   (Swv0V : S w v0 V)
   where
   f : ∀ {v} → v ∈ V0 → Σ _ λ Vv → S w v Vv
   f {v} v∈V0 with v ≡? v0
   ... | (yes refl) = V , Swv0V
   ... | (no n) = ⦃ v ⦄ , S-quasirefl (SwuY⊆Rw SwuV0 v∈V0)

   f≡v0 : ∀ {v} → v ∈ V → v ∈ proj₁ (f v0∈V0)
   f≡v0 {v} v∈V with v0 ≡? v0
   ... | yes refl = v∈V
   ... | no n = ⊥-elim (n refl)

 -- thm⇒ = {!!}
 thm⇒ (var x) = O.var x
 thm⇒ A@(impl _) = O'.⊩⇝ ⇐ λ {wA → thm⇒ ((⊩⇝ ⇒ A) (thm⇐ wA))}
thm⇒ {w} {A'@(A , Ap1 , Ap2 , Ap3)} D▷E@(rhd {D} {E} xl) = O'.⊩▷ ⇐
   λ { {𝔹@(u , B , _)} R'wu@(Rwu , Rwu∈) u⊩D
     → case (⊩▷ ⇒ D▷E) Rwu (thm⇐ u⊩D) of
     λ { (V0 , SwuV0 , V0⊩E) →
   let v0 = proj₁ (Swu-sat SwuV0)
       v0∈V0 = proj₂ (Swu-sat SwuV0)
       𝔸 = w , A'
       C : Pred (W × W) (lsuc ℓW ⊔ ℓS ⊔ ℓR)
       C = λ { (x , y) →
         (y ≡ v0 × R x v0) ⊎
         (w ≡ x × (w , y) ∈ B × Σ _ λ V' → S x v0 V' × y ∈ V') ⊎
         (¬ (R x w) × w ≢ x × Σ _ λ V' → S x v0 V' × y ∈ V') ⊎
         R x w × (x , y) ∈ A × Σ _ λ V' → S x v0 V' × y ∈ V'}
       V0-v = V0 ∩ λ x → x ≢ v0
       ℂ : W'
       ℂ = v0 , C
         , (λ {x} {V} Sxv0V →
         let
           case-w≡x : w ≡ x → Σ 𝕎 λ V' → Σ _ λ y →
             y ∈ V' × V' ⊆ R x × V ⊆ V' × (x , y) ∈ C
           case-w≡x = λ {refl →
             let
```

```
          open F V0 V v0 u w v0∈V0 SwuV0 Sxv0V
          SwuV∪V0-v0 = quasitrans SwuV0 f
          pB = W'.p1 𝔹 SwuV∪V0-v0
          V' = proj₁ pB
          y = proj₁ ∘ proj₂ $ pB
          y∈V' : y ∈ V'
          y∈V' = proj₁ ∘ proj₂ ∘ proj₂ $ pB
          V'⊆Rx : V' ⊆ R x
          V'⊆Rx = proj₁ ∘ proj₂ ∘ proj₂ ∘ proj₂ $ pB
          V∪V0-v0⊆V' : _ ⊆ V'
          V∪V0-v0⊆V' = proj₁ ∘ proj₂ ∘ proj₂ ∘ proj₂ ∘ proj₂
$ pB
          wy∈B : (w , y) ∈ B
          wy∈B = proj₂ ∘ proj₂ ∘ proj₂ ∘ proj₂ ∘ proj₂ $ pB
          pB2 = W'.p2 𝔹 wy∈B
          V⊆V' : V ⊆ V'
          V⊆V' = λ {∈V → V∪V0-v0⊆V' (v0 , v0∈V0 , f≡v0 ∈V)}
          Swv0⋃ : S w v0 _
          Swv0⋃ = monotone Sxv0V (λ { {v} v∈V → v0 , v0∈V0 ,
f≡v0 v∈V})
           λ {z → V'⊆Rx (V∪V0-v0⊆V' z)}
          Swv0V' : S w v0 V'
          Swv0V' = monotone Swv0⋃ V∪V0-v0⊆V' V'⊆Rx
          wy∈C : (w , y) ∈ C
         wy∈C = inj₂ (inj₁ (refl , wy∈B , _ , Swv0V' , y∈V'))
         in V' , y , y∈V' , V'⊆Rx , V⊆V' , wy∈C}
        case-w≠x : w ≢ x → ¬ R x w → Σ 𝕎 λ V' → Σ _ λ y →
         y ∈ V' × V' ⊆ R x × V ⊆ V' × (x , y) ∈ C
        case-w≠x = λ {w≠x ¬Rwx →
         let
          V⊆C : V ⊆ λ {y → (x , y) ∈ C}
          V⊆C = λ { {y} y∈V → inj₂ (inj₂ (inj₁
           (¬Rwx , (w≠x , V , Sxv0V , y∈V))))}
          y∈V = proj₂ $ Swu-sat Sxv0V
         in V , _ , y∈V , SwuY⊆Rw Sxv0V , (λ i → i) , V⊆C y∈V}
        case-Rxw : R x w → Σ 𝕎 λ V' → Σ _ λ y →
         y ∈ V' × V' ⊆ R x × V ⊆ V' × (x , y) ∈ C
        case-Rxw = λ { Rxw →
         let
          Rwv0 = SwuY⊆Rw SwuV0 v0∈V0
          Sxwv0 : S x w { v0 }
          Sxwv0 = R-Sw-trans Rxw Rwv0
          SwxV : S x w V
         SwxV = S-ext (quasitrans Sxwv0 λ {refl → _ , Sxv0V})
           (λ { (_ , refl , p) → p})
           λ {v∈V → v0 , refl , v∈V}
          p = W'.p1 𝔸 SwxV
          V' = proj₁ p
          y∈V' = proj₁ ∘ proj₂ ∘ proj₂ $ p
          V'⊆Rx = proj₁ ∘ proj₂ ∘ proj₂ ∘ proj₂ $ p
          V⊆V' = proj₁ ∘ proj₂ ∘ proj₂ ∘ proj₂ ∘ proj₂ $ p
```

```
                  xy∈A : (x , _) ∈ A
                  xy∈A = proj₂ ∘ proj₂ ∘ proj₂ ∘ proj₂ ∘ proj₂ $ p
                  SxwV' : S x v0 V'
                  SxwV' = monotone Sxv0V V⊆V' V'⊆Rx
                 in V' , _ , y∈V' , V'⊆Rx , V⊆V'
                     , inj₂ (inj₂ (inj₂ (Rxw , xy∈A , (_ , (SxwV' ,
y∈V'))))))}
              in
              case w ≡? x , R? x w of
                λ { (yes refl , _) → case-w≡x refl
                ; (no w≢x , no ¬Rxw) → case-w≢x w≢x ¬Rxw
                ; (no w≢x , yes Rxw) → case-Rxw Rxw } )
            , (λ { (inj₁ (refl , Rxv0)) → _ , S-quasirefl Rxv0 , refl ;
                (inj₂ (inj₁ (refl , wy∈ , V , Sxv0V , ∈V))) → _ , Sxv0V
, ∈V ;
                (inj₂ (inj₂ (inj₁ (¬Rxw , w≢x , V , Sxv0V , ∈V)))) → _
, Sxv0V , ∈V ;
                (inj₂ (inj₂ (inj₂ (Rxw , ∈A , V , Sxv0V , ∈V)))) → _ ,
Sxv0V , ∈V})
              , λ { Rxv0 → inj₁ (refl , Rxv0)}
            Rwv0 : R w v0
            Rwv0 = SwuY⊆Rw SwuV0 v0∈V0
            C→A : ∀ {x y} → R x w → (x , y) ∈ C → (x , y) ∈ A
            C→A = λ { {x} {y} Rxw xy∈C@(inj₁ (refl , Rxv0))
                → case W'.p2 ℂ xy∈C of λ { (V , SV , ∈V) → {!Ap3!}} ;
            Rxw (inj₂ (inj₁ (refl , wy∈B , _))) → ⊥-elim (R-Irreflexive
{F = F} refl Rxw) ;
              Rxw (inj₂ (inj₂ (inj₁ x))) → ⊥-elim (proj₁ x Rxw) ;
              Rxw (inj₂ (inj₂ (inj₂ (_ , xy∈A , _)))) → xy∈A }
            wv0∈B : (w , v0) ∈ B
            wv0∈B = {!!}
            C→B : ∀ {y} → (w , y) ∈ C → (w , y) ∈ B
            C→B = λ { {y} xy∈C@(inj₁ (refl , Rwv0)) → wv0∈B ;
              (inj₂ (inj₁ (refl , wy∈B , _))) → wy∈B ;
              (inj₂ (inj₂ (inj₁ (_ , w≢w , _)))) → ⊥-elim (w≢w refl) ;
               (inj₂ (inj₂ (inj₂ (Rww , _)))) → ⊥-elim (R-Irreflexive
{F = F} refl Rww) }
            R'AB : R' 𝔸 𝔹
            R'AB = Rwu , Rwu∈
            R'AC : R' 𝔸 ℂ
            R'AC = Rwv0 , (λ { {x} Rxw xy∈C → C→A Rxw xy∈C})
        in ℂ , ((R'AB , R'AC , λ {x → C→B x}) , thm⇨ (V0⊩E v0∈V0))
           }}

      thm'⇨ (var x) = O.var x
      thm'⇨ {w} {C} (impl {A} {B} a b) = O.impl (thm⇨ a) (thm'⇨ b)
      thm'⇨ bot = O.bot
      thm'⇨ {w} {A , Ap1 , Ap2} DE@(rhd {D} {E} _) = case ⊮▷ ⇒ DE of
        λ { (u , Rwu , u⊩D , pu) →
          let open Aux u Rwu u⊩D (λ {SV → pu _ (Swu-sat SV) SV})
          in O'.⊮▷ ⇐ (𝔹 , {!!})
```

```
    }
    where
    module Aux
     (u : W)
     (Rwu : R w u)
     (u⊩D : M , u ⊩ D)
     (pu : ∀ {V} → S w u V → Satisfiable (V ∩ (M ,_⊮ E)))
     where
     B : Pred (W × W) (lsuc ℓW ⊔ ℓS ⊔ ℓR)
     B (x , y) =
         (y ≡ u × R x u)
       ⊎ (w ≡ x × M , y ⊮ E × Σ 𝕎 λ V → S w u V × y ∈ V)
       ⊎ (¬ R x w × w ≢ x × Σ 𝕎 λ V → S x u V × y ∈ V)
       ⊎ (R x w × (x , y) ∈ A × Σ 𝕎 λ V → S x u V × y ∈ V)
     p2 : ∀ {x y} → (x , y) ∈ B → Σ 𝕎 λ V → S x u V × y ∈ V
     p2 (inj₁ (refl , Rxu)) = _ , S-quasirefl Rxu , refl
    p2 (inj₂ (inj₁ (refl , y⊮E , V , SwuV , y∈V))) = V , SwuV , y∈V
     p2 (inj₂ (inj₂ (inj₁ (¬Rxw , x≠w , V , SxuV , y∈V)))) = V ,
SxuV , y∈V
     p2 (inj₂ (inj₂ (inj₂ (Rxw , xy∈A , V , SxuV , y∈V)))) = V ,
SxuV , y∈V
     p1 : ∀ {x V} → S x u V → Σ 𝕎 λ V' → Σ W λ y
       → y ∈ V' × V' ⊆ R x × V ⊆ V' × (x , y) ∈ B
     p1 {x} {V} SxuV with w ≡? x
     ... | yes refl = case pu SxuV of
       λ { (y , y∈V , yE) → V , _ , y∈V , (λ {z → SwuY⊆Rw SxuV z
}) , (λ {z → z})
      , inj₂ (inj₁ (refl , yE , _ , SxuV , y∈V))}
     ... | no w≠x with R? x w
    ... | no ¬Rxw = V , _ , (proj₂ (Swu-sat SxuV) , (λ {z → SwuY⊆Rw
SxuV z })
      , (λ {z → z})
       , (inj₂ (inj₂ (inj₁ (¬Rxw , (w≠x , (_ , (SxuV , (proj₂
(Swu-sat SxuV)))))))))
     ... | yes Rxw = V' , y , y∈V' , (λ {z → V'⊆Rx z}) , (λ {z →
V⊆V' z})
      , inj₂ (inj₂ (inj₂ (Rxw , (xy∈A , _ , (SxuV' , y∈V')))))
      where
      SxwV : S x w V
      SxwV = S-trans Rxw Rwu SxuV
       where open FrameLProperties F
      p : Σ 𝕎 λ V' → Σ _ λ y
       → y ∈ V' × V' ⊆ R x × V ⊆ V' × (x , y) ∈ A
      p = Ap1 SxwV
      V' = proj₁ p
      y = proj₁ ∘ proj₂ $ p
      y∈V' = proj₁ ∘ proj₂ ∘ proj₂ $ p
      V'⊆Rx = proj₁ ∘ proj₂ ∘ proj₂ ∘ proj₂ $ p
      V⊆V' = proj₁ ∘ proj₂ ∘ proj₂ ∘ proj₂ ∘ proj₂ $ p
      xy∈A = proj₂ ∘ proj₂ ∘ proj₂ ∘ proj₂ ∘ proj₂ $ p
      SxuV' : S x u V'
```

```
            SxuV' = monotone SxuV V⊆V' V'⊆Rx
          𝔹 : W'
          𝔹 = u , B
            , p1
            , p2
            , inj₁ ∘ (refl ,_)
          c0 : ∀ {x} → R x w → R x u → (x , u) ∈ A
          c0 Rxw Rxu = case Ap1 (R-Sw-trans Rxw Rwu) of
            λ { (V' , y , y∈V' , V'⊆Rx , u∈V' , xy∈A) → {!!}}
          l1 : ∀ {x y} → R x w → (x , y) ∈ B → (x , y) ∈ A
          l1 Rxw (inj₁ (refl , Rxu)) = c0 Rxw Rxu
          l1 Rxw (inj₂ (inj₁ (refl , snd))) = ⊥-elim (R-Irreflexive {F
= F} refl Rxw)
          l1 Rxw (inj₂ (inj₂ (inj₁ x))) = ⊥-elim (proj₁ x Rxw)
          l1 Rxw (inj₂ (inj₂ (inj₂ y))) = proj₁ ∘ proj₂ $ y

    theorem : ∀ {w C A} → M , w ⊩ A ⇔ M' O., w , C ⊩ A
    theorem = equivalence thm⇒ thm⇐
```

B.19. GeneralizedVeltmanSemantics/Properties

```
module GeneralizedVeltmanSemantics.Properties where

open import Function.Equivalence using (_⇔_; equivalence; module Equivalence)

open import Agda.Builtin.Nat using (Nat; suc; zero)
open import Agda.Builtin.Unit using (⊤; tt)
open import Agda.Primitive using (Level; lzero; lsuc; _⊔_)
open import Data.Empty using (⊥; ⊥-elim)
open import Data.Product using (Σ; proj₁; proj₂; _×_) renaming (_,_ to
_,_)
open import Data.Sum using (_⊎_; inj₁; inj₂; [_,_])
open import Function using (_∘_; const; case_of_; id)
open import Function.Equality using (_⟨$⟩_)
open import Relation.Binary using (REL; Rel; Transitive)
open import Relation.Nullary using (yes; no; ¬_)
open import Relation.Unary using (Pred; _∈_; _∉_; Decidable; Satisfiable;
_⊆_; _∩_; {_})
open import Relation.Binary using (Irreflexive) renaming (Decidable to
Decidable₂)
open import Relation.Binary.PropositionalEquality using (_≡_; refl)

open import Formula using (Fm; Var; _↣_; ⊥'; _▷_; _◁_; var; ⊤'; ¬'_;
□_; ◇_; _∧_; _∨_; car)
import GeneralizedVeltmanSemantics as G
open import GeneralizedFrame using (module Trans-conditions)
open import Base
import Principles as P

module PGeneric
  (T : ∀ {ℓW ℓS} (W : Set ℓW) → REL₃ W W (Pred W ℓW) ℓS → Set (lsuc ℓW
⊔ ℓS))
  where
  open G.Generic T public

  R-Irreflexive : ∀ {ℓW ℓR ℓS W R S T} → {F : G.Frame {ℓW} {ℓR} {ℓS} W
R S T} → Irreflexive _≡_ R
  R-Irreflexive {F = F} {x} refl Rxx = R-noetherian (infiniteRefl Rxx)
    where open G.Frame F

  module _
    {ℓW ℓR ℓS}
    {W R S V}
    {M : Model {ℓW} {ℓR} {ℓS} W R S V} where
    open Model M
    open G.Frame F

    ⊩⊥ : ∀ {w} → ¬ (M , w ⊩ ⊥')
    ⊩⊥ = λ ()

    ⊮→¬⊩ : ∀ {w A} → M , w ⊮ A → ¬ (M , w ⊩ A)
    ⊮→¬⊩ (var x) (var x₁) = x x₁
    ⊮→¬⊩ (impl x a) (impl (inj₁ x₁)) = ⊮→¬⊩ x₁ x
    ⊮→¬⊩ (impl x a) (impl (inj₂ y)) = ⊮→¬⊩ a y
    ⊮→¬⊩ (rhd (u , Rwu , fst , snd)) (rhd z) with z Rwu
    ... | inj₁ x = ⊮→¬⊩ x fst
    ... | inj₂ (Y , SwuY , p)
      = case snd Y (Swu-sat SwuY) of
        λ { (inj₁ x) → x SwuY ; (inj₂ (y , w , z))
        → ⊮→¬⊩ z (p w)}

    ⊩→¬⊮ : ∀ {w A} → M , w ⊩ A → ¬ (M , w ⊮ A)
    ⊩→¬⊮ x y = ⊮→¬⊩ y x

    ⊩MP : ∀ {w A B} → M , w ⊩ A ⇾ B → M , w ⊩ A → M , w ⊩ B
    ⊩MP (impl (inj₁ x)) y = ⊥-elim (⊩→¬⊮ y x)
    ⊩MP (impl (inj₂ x)) y = x

    ⊩¬ : ∀  {w A} → (M , w ⊩ ¬' A) ⇔ (M , w ⊮ A)
    ⊩¬ {w} {A} = equivalence ⇒ ⇐
      where ⇒ : M , w ⊩ ¬' A → M , w ⊮ A
            ⇒ (impl (inj₁ x)) = x
            ⇐ : M , w ⊮ A → M , w ⊩ ¬' A
            ⇐ (var x) = impl (inj₁ (var x))
            ⇐ (impl x x₁) = impl (inj₁ (impl x x₁))
            ⇐ (rhd x) = impl (inj₁ (rhd x))
            ⇐ bot = impl (inj₁ bot)

    ⊩T : ∀ {w} → M , w ⊩ T'
    ⊩T = impl (inj₁ bot)

    ⊮T : ∀ {w} → ¬ (M , w ⊮ T')
```

```
⊮⊤ (impl x x₁) = ⊩→¬⊮ x x₁

⊮¬ : ∀ {w A} → M , w ⊮ ¬' A ⇔ M , w ⊩ A
⊮¬ {w} {A} = equivalence ⇨ ⇦
  where
    ⇨ : M , w ⊮ ¬' A → M , w ⊩ A
    ⇨ (impl x x₁) = x
    ⇦ : M , w ⊩ A → M , w ⊮ ¬' A
    ⇦ x = impl x bot

⊩¬¬ : ∀ {w A} → M , w ⊩ ¬' ¬' A ⇔ M , w ⊩ A
⊩¬¬ {w} {A} = equivalence ⇨ ⇦
  where
  ⇨ : M , w ⊩ ¬' ¬' A → M , w ⊩ A
  ⇨ (impl (inj₁ (impl x x₁))) = x
  ⇦ : M , w ⊩ A → M , w ⊩ ¬' ¬' A
  ⇦ x = impl (inj₁ (impl x bot))

⊮¬¬ : ∀  {w A} → M , w ⊮ ¬' ¬' A ⇔ M , w ⊮ A
⊮¬¬ {w} {A} = equivalence ⇨ ⇦
  where
  ⇨ : M , w ⊮ ¬' ¬' A → M , w ⊮ A
  ⇨ x = ⊩¬ ⇒ (⊮¬ ⇒ x)
  ⇦ : M , w ⊮ A → M , w ⊮ ¬' ¬' A
  ⇦ x = ⊮¬ ⇐ (⊩¬ ⇐ x)

⊩∧ : ∀ {w A B} → M , w ⊩ A ∧ B ⇔ (M , w ⊩ A × M , w ⊩ B)
⊩∧ {w} {A} {B} = equivalence ⇨ ⇦
  where
  ⇨ : M , w ⊩ A ∧ B → M , w ⊩ A × M , w ⊩ B
  ⇨ (impl (inj₁ (impl x (impl x₁ y)))) = x , x₁
  ⇦ : M , w ⊩ A × M , w ⊩ B → M , w ⊩ A ∧ B
  ⇦ (fst , snd) = impl (inj₁ (impl fst (impl snd bot)))

⊮∧ : ∀ {w A B} → M , w ⊮ A ∧ B ⇔ (M , w ⊮ A ⊎ M , w ⊮ B)
⊮∧ {w} {A} {B} = equivalence ⇨ ⇦
  where
  ⇨ : M , w ⊮ A ∧ B → (M , w ⊮ A ⊎ M , w ⊮ B)
  ⇨ (impl (impl (inj₁ x)) x₁) = inj₁ x
  ⇨ (impl (impl (inj₂ (impl (inj₁ x)))) x₁) = inj₂ x
  ⇦ : (M , w ⊮ A ⊎ M , w ⊮ B) → M , w ⊮ A ∧ B
  ⇦ (inj₁ x) = impl (impl (inj₁ x)) bot
  ⇦ (inj₂ y) = impl (impl (inj₂ (⊩¬ ⇐ y))) bot

⊩∨ : ∀ {w A B} → M , w ⊩ A ∨ B ⇔ (M , w ⊩ A ⊎ M , w ⊩ B)
⊩∨ {w} {A} {B} = equivalence ⇨ ⇦
  where
  ⇨ : M , w ⊩ A ∨ B → (M , w ⊩ A ⊎ M , w ⊩ B)
  ⇨ (impl (inj₁ (impl x x₁))) = inj₁ x
  ⇨ (impl (inj₂ y)) = inj₂ y
  ⇦ : (M , w ⊩ A ⊎ M , w ⊩ B) → M , w ⊩ A ∨ B
```

```
    ⇐ (inj₁ x) = impl (inj₁ (⊮¬ ⇐ x))
    ⇐ (inj₂ y) = impl (inj₂ y)

  ⊩□ : ∀ {w A} → M , w ⊩ □ A ⇔ (∀ {v} → R w v → M , v ⊩ A)
  ⊩□ {w} {A} = equivalence ⇒ ⇐
    where
    ⇒ : M , w ⊩ □ A → (∀ {v} → R w v → M , v ⊩ A)
    ⇒ (rhd f) wRv with f wRv
    ... | (inj₁ x) = ⊮¬ ⇒ x
    ... | inj₂ (Y , SwvY , x) = case Swu-sat SwvY of
      λ { (u , snd) → ⊥-elim (⊩⊥ (x snd)) }
    ⇐ : (∀ {v} → R w v → M , v ⊩ A) → M , w ⊩ □ A
    ⇐ x = rhd λ wRu → inj₁ (⊮¬ ⇐ (x wRu))

  ⊮□ : ∀ {w A} → M , w ⊮ □ A ⇔ (Σ (W) λ u → R w u × M , u ⊮ A)
  ⊮□ {w} {A} = equivalence ⇒ ⇐
    where
    ⇒ : M , w ⊮ □ A → (Σ (W) λ u → R w u × M , u ⊮ A)
    ⇒ (rhd (u , wRu , u⊩¬A , snd)) = u , wRu , ⊩¬ ⇒ u⊩¬A
    ⇐ : (Σ W λ u → R w u × M , u ⊮ A) → M , w ⊮ □ A
    ⇐ (u , wRu , u⊮A) = rhd (u , (wRu , ⊩¬ ⇐ u⊮A
      , λ { Y (v , p) → inj₂ (v , p , bot)}))

  ⊩◇ : ∀  {w A} → M , w ⊩ ◇ A ⇔ (Σ (W) λ u → R w u × M , u ⊩ A)
  ⊩◇ {w} {A} = equivalence ⇒ ⇐
    where
    ⇒ : M , w ⊩ ◇ A → Σ W λ u → R w u × M , u ⊩ A
    ⇒ (impl (inj₁ (rhd (u , m , u⊩¬¬A , snd)))) = u , m , ⊩¬¬ ⇒ u⊩¬¬A
    ⇐ : (Σ W λ u → R w u × M , u ⊩ A) → M , w ⊩ ◇ A
    ⇐ (u , wRu , snd) = impl (inj₁ (rhd (u , (wRu , ⊩¬¬ ⇐ snd , λ _ s
→ inj₂ (proj₁ s , proj₂ s , bot)))))

  ⊮◇ : ∀ {w A} → M , w ⊮ ◇ A ⇔ (∀ {u} → R w u → M , u ⊮ A)
  ⊮◇ {w} {A} = equivalence ⇒ ⇐
    where
    ⇒ : M , w ⊮ ◇ A → (∀ {u} → R w u → M , u ⊮ A)
    ⇒ (impl (rhd x) y) wRu with x wRu
    ... | inj₁ z = ⊮¬¬ ⇒ z
    ... | inj₂ (Y , z , k) = ⊥-elim (⊩⊥ (k (proj₂ (Swu-sat z))))
    ⇐ : (∀ {u} → R w u → M , u ⊮ A) → M , w ⊮ ◇ A
    ⇐ x = impl (rhd (λ wRu → inj₁ (⊮¬¬ ⇐ (x wRu)))) bot

  ⊩4' : ∀ {w A} → M , w ⊩ □ A → M , w ⊩ □ □ A
  ⊩4' (rhd x) = ⊩□ ⇐ λ {u} wRu → rhd (λ {v} uRv →
    case x {v} (R-trans wRu uRv) of λ { (inj₁ x) → inj₁ x
        ; (inj₂ (Y , SY , snd)) → ⊥-elim (⊩⊥ (snd (proj₂ (Swu-sat
SY))))})

  ⊩⤳⇒ : ∀ {w A B} → M , w ⊩ A ⤳ B → (M , w ⊩ A → M , w ⊩ B)
  ⊩⤳⇒ (impl (inj₁ x)) y = ⊥-elim (⊮→¬⊩ x y)
  ⊩⤳⇒ (impl (inj₂ b)) y = b
```

```
module Extended
  (M,_*⊩?_ : MultiDecidableModel M)
  (∈S? : Decidable₃ S)
  (∈SV? : ∀ {w u Y} → S w u Y → Decidable Y) where

  infix 5 M,_⊩?_
  M,_⊩?_ : DecidableModel M
  M,_⊩?_ = SingleDecidableModel M,_*⊩?_

  open Model M

  [⊩_] : Fm → 𝕎
  [⊩ A ] y with M, y ⊩? A
  ... | inj₁ p = ⊤ × (⊥ → W)
  ... | inj₂ p = ⊥ × (⊥ → W)

  ∈[⊩_] : ∀ {y} A → y ∈ [⊩ A ] ⇔ M , y ⊩ A
  ∈[⊩_] {y} A = equivalence ⇒ ⇐
    where
    ⇒ : y ∈ [⊩ A ] → M , y ⊩ A
    ⇒ x with M, y ⊩? A
    ... | inj₁ k = k
    ⇐ : M , y ⊩ A → y ∈ [⊩ A ]
    ⇐ fst with M, y ⊩? A
    ... | inj₁ x = tt , λ ()
    ... | inj₂ y = ⊩→¬⊮ fst y , λ ()

  ⊩⇝ : ∀ {w A B} → M , w ⊩ A ⇝ B ⇔ (M , w ⊩ A → M , w ⊩ B)
  ⊩⇝ {w} {A} {B} = equivalence ⊩⇝⇒ ⇐
    where
    ⇐ : (M , w ⊩ A → M , w ⊩ B) → M , w ⊩ A ⇝ B
    ⇐ x with M, w ⊩? A
    ... | inj₁ z = impl (inj₂ (x z))
    ... | inj₂ y = impl (inj₁ y)

  -- ⊮⇝ : ∀ {w A B} → M , w ⊮ A ⇝ B ⇔ (M , w ⊩ A × M , w ⊮ B)
  -- ⊮⇝ {w} {A} {B} = equivalence ⇒ ⇐
  --   where
  --   ⇒ : M , w ⊮ A ⇝ B → (M , w ⊩ A × M , w ⊮ B)
  --   ⇒ x = {!!}
  --   ⇐ : (M , w ⊩ A × M , w ⊮ B) → M , w ⊮ A ⇝ B
  --   ⇐ = {!!}

  ⊩▷ : ∀ {w A B} → M , w ⊩ A ▷ B ⇔
    (∀ {u} → R w u → M , u ⊩ A → Σ 𝕎 λ Y → S w u Y × Y ⊆ M ,_⊩ B)
  ⊩▷ {w} {A} {B} = equivalence ⇒ ⇐
    where
    ⇒ : M , w ⊩ A ▷ B → (∀ {u} → R w u → M , u ⊩ A
      → Σ 𝕎 λ Y
      → S w u Y × Y ⊆ M ,_⊩ B)
```

```
        ⇨ (rhd x) {u} wRu uA with x wRu
        ⇨ (rhd x) {u} wRu uA | inj₁ z = ⊥-elim (⊩→¬⊮ uA z)
        ⇨ (rhd x) {u} wRu uA | inj₂ (fst , fst₁ , snd) = fst , (fst₁ ,
snd)
        ⇦ : (∀ {u} → R w u → M , u ⊩ A → Σ 𝕎 λ Y → S w u Y
          × Y ⊆ M ,_⊩ B) → M , w ⊩ A ▷ B
        ⇦ x = rhd (λ {u} wRu → [ (λ x₁ → inj₂ (x wRu x₁)) , inj₁ ] (M,
u ⊩? A))

      ⊮▷ : ∀ {w A B} → M , w ⊮ A ▷ B ⇔
        Σ W (λ u → R w u × M , u ⊩ A × ∀ Y → Satisfiable Y → S w u Y →
(Satisfiable (Y ∩ (M ,_⊮ B))))
      ⊮▷ {w} {A} {B} = equivalence ⇨ ⇦
        where
        ⇨ : M , w ⊮ A ▷ B → Σ W (λ u → R w u × M , u ⊩ A × ((Y : 𝕎) →
              Satisfiable Y → S w u Y → Satisfiable (Y ∩ (M ,_⊮ B))))
        ⇨ (rhd (u , Rwu , uA , snd)) = u , Rwu , uA , (λ {Y x x₁ → case
snd Y x of
          λ { (inj₁ x) → ⊥-elim (x x₁) ; (inj₂ y) → y}})
        ⇦ : Σ W (λ u → R w u × M , u ⊩ A × ((Y : 𝕎) →
             Satisfiable Y → S w u Y → Satisfiable (Y ∩ (M ,_⊮ B)))) →
          M , w ⊮ A ▷ B
        ⇦ (u , Rwu , uA , snd) = rhd (u , Rwu , uA , λ Y x → case ∈S? w
u Y of
          λ { (yes p) → inj₂ (snd Y x p) ; (no p) → inj₁ p})

      ⊩◁ : ∀ {w A B} → M , w ⊩ A ◁ B ⇔ Σ W λ x → R w x × M , x ⊩ A × ∀
{V} → S w x V → Σ W λ b → b ∈ V × M , b ⊩ B
      ⊩◁ {w} {A} {B} = equivalence ⇨ ⇦
        where
        ⇨ : M , w ⊩ A ◁ B → Σ W λ x → R w x × M , x ⊩ A × ∀ {V} → S w x
V → Σ W λ b → b ∈ V × M , b ⊩ B
        ⇨ A◁B = case ⊩¬ ⇒ A◁B of λ {z → case ⊮▷ ⇒ z of
          λ { (u , Rwu , uA , snd) → u , Rwu , uA , λ {V} SwuV → case
snd V (Swu-sat SwuV) SwuV of
          λ { (b , fst , snd) → b , (fst , (⊮¬ ⇒ snd)) }}}
        ⇦ : (Σ W λ x → R w x × M , x ⊩ A × ∀ {V} → S w x V → Σ W λ b →
b ∈ V × M , b ⊩ B) → M , w ⊩ A ◁ B
        ⇦ (u , Rwu , uA , p) = ⊩¬ ⇐ (⊮▷ ⇐ (u , Rwu , uA , (λ Y satY SwuY
→ case p SwuY of
          λ { (y , y∈Y , yB) → y , (y∈Y , ⊮¬ ⇐ yB)})))

      ⊩4 : ∀ {w A} → M , w ⊩ □ A ⇝ □ □ A
      ⊩4 = ⊩⇝ ⇐ ⊩4'

      ⊩⇔¬⊮ : ∀ {w A} → M , w ⊩ A ⇔ (¬ M , w ⊮ A)
      ⊩⇔¬⊮ {w} {A} = equivalence ⊩→¬⊮ ⇦
        where
        ⇦ : (M , w ⊮ A → ⊥) → M , w ⊩ A
        ⇦ x = [ id , (λ y → ⊥-elim (x y)) ] (M, w ⊩? A)
```

```
⊮⇔¬⊩ : ∀ {w A} → M , w ⊮ A ⇔ (¬ M , w ⊩ A)
⊮⇔¬⊩ {w} {A} = equivalence ⊮→¬⊩ ⇦
  where
  ⇦ : ¬ M , w ⊩ A → M , w ⊮ A
  ⇦ x = [ (λ y → ⊥-elim (x y)) , id ] (M, w ⊩? A)

⊩K : ∀ {w A B} → M , w ⊩ □ (A ↝ B) ↝ □ A ↝ □ B
⊩K {w} {A} {B} = ⊩↝ ⇐ λ x → ⊩↝ ⇐ λ y → ⊩□ ⇐
  λ {u} wRu → ⊩MP ((⊩□ ⇒ x) wRu) ((⊩□ ⇒ y) wRu)


⊩J1 : ∀ {w A B} → M , w ⊩ □ (A ↝ B) ↝ A ▷ B
⊩J1 {w} {A} {B} = ⊩↝ ⇐ λ x → rhd λ { {u} Rwu → case M, u ⊩? A of
  λ { (inj₁ k) → inj₂ (⦃ u ⦄ , S-quasirefl Rwu ,
  λ {refl → ⊩MP ((⊩□ ⇒ x) Rwu) k})
  ; (inj₂ y) → inj₁ y}}


⊩J3 : ∀ {w A B C} → M , w ⊩ A ▷ C ∧ B ▷ C ↝ (A ∨ B) ▷ C
⊩J3 {w} {A} {B} {C} = ⊩↝ ⇐ λ x → ⊩▷ ⇐ λ Rwu uA∨B → case ⊩∧ ⇒ x of
  λ {(a , b) → case ⊩∨ ⇒ uA∨B of
  λ { (inj₁ uA) → (⊩▷ ⇒ a) Rwu uA ;
  (inj₂ uB) → (⊩▷ ⇒ b) Rwu uB} }

⊩J4 : ∀ {w A B} → M , w ⊩ A ▷ B ↝ ◊ A ↝ ◊ B
⊩J4 = ⊩↝ ⇐ λ x → ⊩↝ ⇐ λ y → ⊩◊ ⇐ (case ⊩◊ ⇒ y of
  λ { (u , Rwu , snd) → case (⊩▷ ⇒ x) Rwu snd of
  λ { (Y , SwuY , YB) → case Swu-sat SwuY of
  λ { (v , vY) → v , SwuY⊆Rw SwuY vY , YB vY}}})

⊩J5 : ∀ {w A} → M , w ⊩ ◊ A ▷ A
⊩J5 = ⊩▷ ⇐ λ {u} Rwu u⊩◊A → case ⊩◊ ⇒ u⊩◊A of
  λ { (v , Ruv , vA) → ⦃ v ⦄ , R-Sw-trans Rwu Ruv
    , λ {refl → vA}}

[⊩▷] : ∀ {A B w} {𝔸 𝔹 : 𝕎}
  → (∀ {u} → M , u ⊩ A ⇔ u ∈ 𝔸)
  → (∀ {u} → M , u ⊩ B ⇔ u ∈ 𝔹)
  → (∀ {u} → R w u → u ∈ 𝔸 → Σ 𝕎 λ V → S w u V × V ⊆ 𝔹)
  → M , w ⊩ A ▷ B
 [⊩▷] [A] [B] A▷B = ⊩▷ ⇐ λ { {u} Rwu u⊩A → case A▷B Rwu ([A] ⇒
u⊩A) of
  λ { (V , SwuV , V⊆𝔹) → V , SwuV , (λ {z → [B] ⇐ V⊆𝔹 z})}}

module Properties-Trans where

  Trans-4 Trans-7 Trans-8 Trans-3 : Set _
  Trans-4 = ∀ {x u Y} → S x u Y → Σ W λ y → Σ (y ∈ Y) λ y∈Y → ∀
{Y'} → S x y Y' → S x u Y'

  Trans-7 = ∀ {u x y Y Y'} → S x u Y → y ∈ Y → S x y Y' → y ∉ Y'
```

```
          → Σ 𝕎 λ Y'' → Y'' ⊆ Y' × S x u Y''

        Trans-8 = ∀ {u x y Y Y'} → S x u Y → y ∈ Y → S x y Y' → y ∉ Y'
→ S x u Y'

        Trans-3 = ∀ {x u Y} → S x u Y → Σ W λ y → Σ (y ∈ Y) λ y∈Y → ∀
{Y'} → S x y Y' → Σ 𝕎 λ Y''
          → Y'' ⊆ Y' × S x u Y''

        4⇒3 : Trans-4 → Trans-3
        4⇒3 t SxuY = case t SxuY of λ { (y , fst₁ , snd) → y , fst₁ ,
          λ { {Y'} SxyY' → Y' , (λ x → x) , snd SxyY'}}

       ⊩J2-T3 : ∀ {w A B C} → Trans-3 → M , w ⊩ (A ▷ B ∧ B ▷ C) ⇾ A ▷ C
        ⊩J2-T3 {w} {A} {B} {C} t = ⊩⇾ ⇐ λ x → ⊩▷ ⇐ λ {u} Rwu uA → case
⊩∧ ⇒ x of
            λ { (A▷B , B▷C) → case (⊩▷ ⇒ A▷B) Rwu uA of
            λ { (Y , SwuY , YB ) → case t SwuY of
             λ { (y , y∈Y , snd) → case (⊩▷ ⇒ B▷C) (SwuY⊆Rw SwuY y∈Y)
(YB y∈Y) of
            λ { (Y' , SwyY' , Y'⊩C) → case snd SwyY' of
             λ { (Y'' , Y''⊆Y' , SwuY'') → Y'' , SwuY'' , λ {x₁ → Y'⊩C
(Y''⊆Y' x₁)}}}}}}

       ⊩J2-T4 : ∀ {w A B C} → Trans-4 → M , w ⊩ (A ▷ B ∧ B ▷ C) ⇾ A ▷ C
        ⊩J2-T4 {w} {A} {B} {C} t = ⊩J2-T3 (4⇒3 t)

       -- NOTE: you must avoid using S-quasitrans in this proof! this
is hacky and
       -- unelegant, but doing it properly would require a lot of work.
       ⊩J2-T8 : ∀ {w A B C} → Trans-8 → M , w ⊩ (A ▷ B ∧ B ▷ C) ⇾ A ▷ C
        ⊩J2-T8 {w} {A} {B} {C} t = ⊩⇾ ⇐ λ x → ⊩▷ ⇐ λ {u} Rwu uA → case
⊩∧ ⇒ x of
            λ { (A▷B , B▷C) → case (⊩▷ ⇒ A▷B) Rwu uA of
            λ { (Y , SwuY , YB ) → case M, Y *⊩? C of
            λ { (inj₁ x) → Y , SwuY , λ {x₁ → x x₁} ;
           (inj₂ (y , y∈Y , y⊮C)) → case (⊩▷ ⇒ B▷C) (SwuY⊆Rw SwuY y∈Y)
(YB y∈Y) of
            λ { (Y' , SwyY' , Y'⊆C) → Y' , t SwuY y∈Y SwyY' (λ {y∈Y' →
⊮→¬⊩ y⊮C (Y'⊆C y∈Y')}) ,
            λ {x₁ → Y'⊆C x₁}}}}}

  module Extended2
    {ℓW ℓR ℓS}
    {W R S V}
    {M : Model {ℓW} {ℓR} {ℓS} W R S V}
    (M,_*⊩?_ : MultiDecidableModel M)
    (∈S? : Decidable₃ S)
    (∈SV? : ∀ {w u Y} → S w u Y → Decidable Y) where

    infix 5 M,_⊩?_
    M,_⊩?_ : DecidableModel M
    M,_⊩?_ = SingleDecidableModel M,_*⊩?_

    open Model M
    open Extended M,_*⊩?_ ∈S? ∈SV?
    open G.Frame F

    L-chain : ∀ {w u A} → R w u → M , u ⊩ A → M , w ⊩ □ (□ A ⇝ A) →
InfiniteChain R w
    InfiniteChain.b (L-chain {w} {u} Rwu uA uF) = u
    InfiniteChain.a<b (L-chain {w} {u} Rwu uA uF) = Rwu
    InfiniteChain.tail (L-chain {w} {u} Rwu uA uF) =
      case (⊩□ ⇒ uF) Rwu of
      λ { (impl (inj₁ x)) → case ⊮□ ⇒ x of
        λ { (v , Ruv , vA) → L-chain Ruv vA ((⊩□ ⇒ ⊩4' uF) Rwu)}
      ; (impl (inj₂ y)) → ⊥-elim (⊩→¬⊮ y uA) }

    ⊩L : ∀ {w A} → M , w ⊩ □ (□ A ⇝ A) ⇝ □ A
    ⊩L {w} {A} = ⊩⇝ ⇐ λ x → ⊩□ ⇐ λ {u} Rwu → ⊩⇔¬⊮ ⇐
      λ ¬A → R-noetherian (L-chain Rwu ¬A x)

module _ where
  open PGeneric Trans-conditions.Trans-L
  module ExtendedT2
    {ℓW ℓR ℓS}
    {W R S V}
    {M : Model {ℓW} {ℓR} {ℓS} W R S V}
    (M,_*⊩?_ : MultiDecidableModel M)
    (∈S? : Decidable₃ S)
    (∈SV? : ∀ {w u Y} → S w u Y → Decidable Y)
    where
    open Extended M,_*⊩?_ ∈S? ∈SV?
    open Model M
    open G.Frame F

    ⊩J2 : ∀ {w A B C} → M , w ⊩ (A ▷ B ∧ B ▷ C) ⇝ A ▷ C
    ⊩J2 {w} {A} {B} {C} = ⊩⇝ ⇐ λ x → ⊩▷ ⇐ λ {u} Rwu uA → case ⊩∧ ⇒ x of
      λ { (A▷B , B▷C) → case (⊩▷ ⇒ A▷B) Rwu uA of
      λ { (Y , SwuY , YB ) → case M, Y *⊩? C of
      λ { (inj₁ x) → Y , SwuY , λ { {v} k → x k}
      ; (inj₂ (v , v∈Y , v¬C)) → case (⊩▷ ⇒ B▷C) (SwuY⊆Rw SwuY v∈Y) (YB
v∈Y) of
      λ { (Z , SwvZ , ZC) → _ , let
        f : ∀ {y} → y ∈ Y → Σ 𝕎 λ Z → S w y Z × Z ⊆ M ,_⊩ C
        f {y} y∈Y = case (⊩▷ ⇒ B▷C) (SwuY⊆Rw SwuY y∈Y) (YB y∈Y) of
          λ { (Zy , fst , snd) → Zy , fst , λ {z → snd z}}
        f' : ∀ {y} → y ∈ Y → Σ 𝕎 λ Z → S w y Z
        f' y∈Y = proj₁ (f y∈Y) , (proj₁ (proj₂ (f y∈Y))) in
        quasitrans SwuY f' , λ { {x} (y , y∈Y , x∈Z) → proj₂ (proj₂ (f
y∈Y)) x∈Z}}}}}
```

```
module _ where
  open PGeneric Trans-conditions.Trans-L public
  open ExtendedT2 public
module SemanticsProperties-4 = PGeneric Trans-conditions.Trans-4
module SemanticsProperties-3 = PGeneric Trans-conditions.Trans-3
module SemanticsProperties-L = PGeneric Trans-conditions.Trans-L
```

B.20. GeneralizedVeltmanSemantics

```
module _ where

open import Agda.Builtin.Nat using (Nat; suc; _+_)
open import Agda.Primitive using (Level; lzero; lsuc; _⊔_)
open import Data.Empty using (⊥; ⊥-elim)
open import Data.List using (List)
open import Data.List.Relation.Unary.All using (All)
open import Data.Product
open import Data.Sum using (_⊎_; inj₁; inj₂)
open import Function using (_∘_; case_of_)
open import Relation.Binary using (REL; Rel; Transitive; Reflexive)
open import Relation.Binary.PropositionalEquality using (_≡_; _≢_; refl)
open import Relation.Nullary using (¬_)
open import Relation.Unary using (Pred; _∈_; _∉_; Decidable; {_}; _∩_;
_⊆_; Satisfiable)

open import Formula using (Fm; Var; _↝_; ⊥'; _▷_; var; ¬'_)
open import Base using (Noetherian; REL₃; Rel₃)
open import GeneralizedFrame using (FrameL; Frame; module Trans-conditions)
public

module Generic
  (T : ∀ {ℓW ℓS} (W : Set ℓW) → REL₃ W W (Pred W ℓW) ℓS → Set (lsuc ℓW
⊔ ℓS))
  where

  Valuation : ∀ {ℓW ℓR ℓS W R S} → Frame {ℓW} {ℓR} {ℓS} W R S T → Set
(lsuc lzero ⊔ ℓW)
  Valuation {W = W} F = REL W Var lzero

  record Model
    {ℓW ℓR ℓS}
    (W : Set ℓW)
    (R : Rel W ℓR)
    (S : REL₃ _ _ _ ℓS)
    (V : REL W Var lzero)
    : Set (lsuc ℓW ⊔ ℓR ⊔ ℓS) where
    constructor model
    field
      F : Frame {ℓW} {ℓR} {ℓS} W R S T
  infix 5 _,_⊮_
  data _,_⊮_ {ℓW ℓR ℓS W R S V} (M : Model {ℓW} {ℓR} {ℓS} W R S V) (w
: W)
    : Fm → Set (lsuc ℓW ⊔ ℓR ⊔ ℓS)

  infix 5 _,_⊩_
  data _,_⊩_ {ℓW ℓR ℓS W R S V} (M : Model {ℓW} {ℓR} {ℓS} W R S V) (w
: W) : Fm → Set (lsuc ℓW ⊔ ℓR ⊔ ℓS)

  data _,_⊩_ {ℓW} {ℓR} {ℓS} {W} {R} {S} {V} M w where
    var : ∀ {a : Var} → a ∈ V w → M , w ⊩ var a
    impl : ∀ {A B} → M , w ⊮ A ⊎ M , w ⊩ B → M , w ⊩ A ⇝ B
    rhd : ∀ {A B} →
      (∀ {u} → R w u → M , u ⊮ A ⊎ (∃[ Y ] (S w u Y × (Y ⊆ M ,_⊩ B))))
      → M , w ⊩ A ▷ B

  data _,_⊮_ {ℓW} {ℓR} {ℓS} {W} {R} {S} {V} M w where
    var : ∀ {a : Var} → a ∉ V w → M , w ⊮ var a
    impl : ∀ {A B} → M , w ⊩ A → M , w ⊮ B → M , w ⊮ A ⇝ B
    rhd : ∀ {A B} →
      ∃[ u ] (R w u × M , u ⊩ A
       × ∀ Y → Satisfiable Y → (¬ S w u Y) ⊎ (Satisfiable (Y ∩ (M ,_⊮
B))))
      → M , w ⊮ A ▷ B
    bot : M , w ⊮ ⊥'

  DecidableModel : ∀ {ℓW ℓR ℓS W R S V} → Model {ℓW} {ℓR} {ℓS} W R S V
    → Set (lsuc ℓW ⊔ ℓR ⊔ ℓS)
  DecidableModel M = ∀ w A → M , w ⊩ A ⊎ M , w ⊮ A

  MultiDecidableModel : ∀ {ℓW ℓR ℓS W R S V} → Model {ℓW} {ℓR} {ℓS} W
R S V
    → Set (lsuc ℓW ⊔ ℓR ⊔ ℓS ⊔ lsuc ℓW)
  MultiDecidableModel {ℓW = ℓW} {W = W} M =
    ∀ (Y : Pred W ℓW) A → Y ⊆ M ,_⊩ A ⊎ Satisfiable (Y ∩ (M ,_⊮ A))

  SingleDecidableModel : ∀ {ℓW ℓR ℓS W R S V} → {M : Model {ℓW} {ℓR}
{ℓS} W R S V}
    → MultiDecidableModel M → DecidableModel M
  SingleDecidableModel x w A = case x { w } A of
    λ { (inj₁ x) → inj₁ (x refl) ; (inj₂ (u , refl , snd)) → inj₂ snd}

  S-decidable : ∀ {ℓW ℓR ℓS W R S} → (F :  Frame {ℓW} {ℓR} {ℓS} W R S
T)
    → Set (lsuc ℓW ⊔ ℓS)
  S-decidable {S = S} F = ∀ x y u → (Σ 𝕎 λ U → u ∈ U × S x y U) ⊎ (∀ U
→ S x y U → u ∉ U)
    where open Frame F

-- Frame validity
  infix 5 _*⊩_
  _*⊩_ : ∀ {ℓW ℓR ℓS W R S} → Frame {ℓW} {ℓR} {ℓS} W R S T → Fm → Set
(lsuc ℓW ⊔ ℓR ⊔ ℓS)
  F *⊩ A = ∀ val w → model {V = val} F , w ⊩ A

  infix 5 _*⊮_
  _*⊮_ : ∀ {ℓW ℓR ℓS W R S} → Frame {ℓW} {ℓR} {ℓS} W R S T → Fm → Set
(lsuc ℓW ⊔ ℓR ⊔ ℓS)
  F *⊮ A = Σ (Valuation F × _) λ { (val , w) → model {V = val} F , w ⊮
A}
    where open Frame F

  _,_⊩*_ : ∀ {ℓW ℓR ℓS W R S V} → (M : Model {ℓW} {ℓR} {ℓS} W R S V)
(w : W) → List Fm → Set _
  M , w ⊩* Π = All (M , w ⊩_) Π

module _ where open Generic Trans-conditions.Trans-L public

module T1 where open Generic Trans-conditions.Trans-1 public
module T2 where open Generic Trans-conditions.Trans-2 public
module T3 where open Generic Trans-conditions.Trans-3 public
module T4 where open Generic Trans-conditions.Trans-4 public
module T5 where open Generic Trans-conditions.Trans-5 public
module T6 where open Generic Trans-conditions.Trans-6 public
module T7 where open Generic Trans-conditions.Trans-7 public
module T8 where open Generic Trans-conditions.Trans-6 public
```

B.21. IL

```
module IL where

open import OrdinaryVeltmanSemantics using (Model; _,_⊩_; impl)
open import Relation.Binary.PropositionalEquality using (_≡_; refl)
open import Data.Sum using (_⊎_; inj₁; inj₂)
open import Data.List using (List; []; _∷_)
open import Data.List.Membership.Propositional using (_∈_)

open import Formula using (Fm; _▷_; _⇝_; □_; _∧_; _∨_; ◇_; ¬'_; ⊥')

infix 5 _⊢_
data _⊢_ (Π : List Fm) : Fm → Set where
  Ax : ∀ {A} → A ∈ Π → Π ⊢ A
  -- classical axioms
  C1 : ∀ {A B} → Π ⊢ A ⇝ (B ⇝ A)
  C2 : ∀ {A B C} → Π ⊢ (A ⇝ (B ⇝ C)) ⇝ ((A ⇝ B) ⇝ (A ⇝ C))
  C3 : ∀ {A B} → Π ⊢ (¬' A ⇝ ¬' B) ⇝ (B ⇝ A)
  -- other axioms
  K : ∀ {A B} → Π ⊢ (□ (A ⇝ B)) ⇝ (□ A ⇝ □ B)
  L : ∀ {A} → Π ⊢ □ (□ A ⇝ A) ⇝ □ A
  J1 : ∀ {A B} → Π ⊢ □ (A ⇝ B) ⇝ A ▷ B
  J2 : ∀ {A B C} → Π ⊢ A ▷ B ∧ B ▷ C ⇝ A ▷ C
  J3 : ∀ {A B C} → Π ⊢ (A ▷ C ∧ B ▷ C) ⇝ (A ∨ B) ▷ C
  J4 : ∀ {A B} → Π ⊢ A ▷ B ⇝ ◇ A ⇝ ◇ B
  J5 : ∀ {A} → Π ⊢ ◇ A ▷ A
  MP : ∀ {A B} → Π ⊢ A ⇝ B → Π ⊢ A → Π ⊢ B
  nec : ∀ {A} → [] ⊢ A → Π ⊢ □ A
```

B.22. IL/Edsl

```
module IL.Edsl where

import Agda.Builtin.Unit as U
import Data.Empty as Empty
import Data.Fin as Fin
import Data.List as Lst
import Data.Maybe as M
import Function as Fun
import Relation.Binary.PropositionalEquality as Eq
open Eq using (_≢_)
open import Relation.Nullary
open Eq using (_≡_; refl; subst)
open Fin using (Fin; inject+; fromℕ; inject₁; zero; suc; #_)
open Fun using (id)
open Lst using (List; []; _∷_)
open M using (nothing; just; Maybe)
open import Agda.Builtin.Equality
open import Agda.Builtin.FromNat
open import Data.List.Relation.Unary.Any using () renaming (here to
here'; there to there')
open import Agda.Builtin.Nat
open import Data.Bool using (true; false) renaming (_∨_ to _||_ ; _∧_
to _&_)
open import Data.Nat using (_≤?_)
open import Data.List.Membership.Propositional using (_∈_)
open import Data.Product
open import Formula
open import IL

instance
  NumNat : Number Nat
  NumNat .Number.Constraint _ = U.⊤
  NumNat .Number.fromNat    m = m

pattern _! a = var a

here : ∀ {A : Fm} {Σ : List Fm} → A ∈ (A ∷ Σ)
here {A} {Σ} = here' refl

there : ∀ {A : Fm} {Σ B} → A ∈ Σ → A ∈ (B ∷ Σ)
there x = there' x

_!_ : ∀ {A : Set} → (L : List A) → (i : Nat) → Maybe A
[] ! _ = nothing
```

```
(x :: l) ! zero = just x
(x :: l) ! suc i = l ! i

makeMember : ∀ {Σ i A} → Σ ! i ≡ just A → A ∈ Σ
makeMember {a :: l} {zero} refl = here
makeMember {a :: l} {suc i} x = there (makeMember {l} {i} x)

instance
  NumMember : ∀ {A Σ} → Number (A ∈ Σ)
  NumMember {A} {Σ} .Number.Constraint i = Σ ! i ≡ just A
  NumMember {A} {Σ} .Number.fromNat i {{p}} = makeMember p

_≤_ : (m n : Nat) → Set
zero  ≤ n     = U.⊤
suc m ≤ zero  = Empty.⊥
suc m ≤ suc n = m ≤ n

fromN≤ : (m n : Nat) → m ≤ n → Fin (suc n)
fromN≤ zero    _       _  = zero
fromN≤ (suc _) zero    ()
fromN≤ (suc m) (suc n) p  = suc (fromN≤ m n p)

instance
  NumFin : ∀ {n} → Number (Fin (suc n))
  NumFin {n} .Number.Constraint m         = m ≤ n
  NumFin {n} .Number.fromNat    m {{m≤n}} = fromN≤ m n m≤n

data Single {A : Set} : A → Set where
  single : (n : A) → Single n

instance
  NumSing : ∀ {n} → Number (Single {Nat} n)
  NumSing {n} .Number.Constraint m = n ≡ m
  NumSing .Number.fromNat m ⟦ refl ⟧ = single m

data HilbertProof : List Fm → Fm → Nat → Set
lookup-all : ∀ {Σ A n} → HilbertProof Σ A n → Fin (suc n) → List Fm ×
(Fm × Nat)
lookup-may : ∀ {Σ A n} → HilbertProof Σ A n → Nat → Maybe Fm

lookup : ∀ {Σ A n} → HilbertProof Σ A n → Fin (suc n) → Fm
lookup H i = proj₁ (proj₂ (lookup-all H i))

compile-instr : ∀ {n Σ A} → (H : HilbertProof Σ A n) → (i : Fin (suc
n)) → Σ ⊢ lookup H i

data HilbertRef {Σ A n} (H : HilbertProof Σ A n) (fB : Fm) (f : Fm →
Fm) : Set where
  ref : (i : Nat) → M.map f (lookup-may H i) ≡ just fB → HilbertRef H
fB f
```

```
data HilbertProof where
  begin : ∀ {Σ A} → Σ ⊢ A → HilbertProof Σ A 0
  by : ∀ {Σ A B n} → Σ ⊢ B → HilbertProof Σ A n → HilbertProof Σ B (suc
n)
  Ax : ∀ {Σ B n} → (A : Fm) → HilbertProof Σ B n → HilbertProof (A ::
Σ) A (suc n)
  nec : ∀ {Σ n □A C} (H : HilbertProof [] C n) (i : HilbertRef H (□A)
□_)
    → HilbertProof Σ (□A) (suc n)
  MP : ∀ {n Σ A B C} (H : HilbertProof Σ C n) → HilbertRef H (A ⇝ B)
id → HilbertRef H A id → HilbertProof Σ B (suc n)

lookup-all {Σ} {A} {n} x zero = Σ , A , n
lookup-all (by x H) (suc i) = lookup-all H i
lookup-all (MP {A} {B} H i' j') (suc i) = lookup-all H i
lookup-all (begin x) (suc ())
lookup-all (Ax x y) (suc h) = lookup-all y h
lookup-all (nec H j) (suc i) = lookup-all H i

-- nothing if the result would be a negative number
_-≥_ : (n m : Nat) → Maybe (Fin (suc n))
n -≥ zero = just (fromℕ n)
zero -≥ suc m = nothing
suc n -≥ suc m = M.map inject₁ (n -≥ m)

lookup-may {Σ} {A} {n} H i = M.map (lookup H) (n -≥ i)

is-just-map : ∀ {A B : Set} {may : Maybe A} {f : A → B}
  → M.is-just (M.map f may) ≡ M.is-just may
is-just-map {A} {B} {just _} = refl
is-just-map {A} {B} {nothing} = refl

-≥-is-just : ∀ {n i} → M.is-just (n -≥ i) ≡ true → i ≤ n
-≥-is-just {zero} {zero} x = U.tt
-≥-is-just {suc n} {zero} x = U.tt
-≥-is-just {suc n} {suc i} x = -≥-is-just {n} {i} (Eq.trans (Eq.sym
is-just-map) x)

is-just-≡ : ∀ {A : Set} {a : Maybe A} {b} → a ≡ just b → M.is-just a ≡
true
is-just-≡ {_} {just _} x = refl

subst-⊢ : ∀ {Σ A A'} → A ≡ A' → Σ ⊢ A → Σ ⊢ A'
subst-⊢ {Σ} x y = subst (λ x →  Σ ⊢ x) x y

l-weakening : {Σ : List Fm} {A B : Fm} (Π : Σ ⊢ A) → (B :: Σ) ⊢ A
l-weakening (Ax x) = Ax (there x)
l-weakening C1 = C1
l-weakening C2 = C2
l-weakening C3 = C3
l-weakening K = K
```

```
l-weakening L = L
l-weakening J1 = J1
l-weakening J2 = J2
l-weakening J3 = J3
l-weakening J4 = J4
l-weakening J5 = J5
l-weakening (MP A B) = MP (l-weakening A) (l-weakening B)
l-weakening (nec k) = nec k

l-weakening[] : {Σ : List Fm} {A : Fm} (Π : [] ⊢ A) → Σ ⊢ A
l-weakening[] C1 = C1
l-weakening[] C2 = C2
l-weakening[] C3 = C3
l-weakening[] K = K
l-weakening[] L = L
l-weakening[] J1 = J1
l-weakening[] J2 = J2
l-weakening[] J3 = J3
l-weakening[] J4 = J4
l-weakening[] J5 = J5
l-weakening[] (MP a b) = MP (l-weakening[] a) (l-weakening[] b)
l-weakening[] (nec p) = nec p

A⇝A : ∀ {Σ A} → Σ ⊢ A ⇝ A
A⇝A {Σ} {A} = MP (MP C2 C1) (C1 {B = A})

injective-just : ∀ {A : Set} {a b : A} → just a ≡ just b → a ≡ b
injective-just refl = refl

_-Fin_ : (n m : Nat) → Fin (suc n)
n -Fin zero = fromℕ n
zero -Fin suc m = zero
suc n -Fin suc m = inject₁ (n -Fin m)

lookup-may-just : ∀ {n i A B Σ} {H : HilbertProof Σ B n}
  → lookup-may H i ≡ just A
  → lookup H (n -Fin i) ≡ A
lookup-may-just {n} {i} {A} {_} {Σ} {H} p = injective-just p'
  where
    aux-i≤n : ∀ {n i Σ A B} {H : HilbertProof Σ B n} → lookup-may H i
≡ just A → i ≤ n
   aux-i≤n {n} {i} x = -≥-is-just {n} {i} (Eq.trans (Eq.sym is-just-map)
(is-just-≡ x))
    aux--≥ : ∀ {n i} → i ≤ n → n -≥ i ≡ just (n -Fin i)
    aux--≥ {n} {zero} x = refl
    aux--≥ {suc n} {suc i} x = Eq.cong (M.map inject₁) (aux--≥ {n} {i}
x)
    i≤n : i ≤ n
    i≤n = aux-i≤n {n} {i} p
    n-i-just : n -≥ i ≡ just (n -Fin i)
    n-i-just = aux--≥ {n} {i} i≤n
    p-rewrite : ∀ {n i B Σ} {H : HilbertProof Σ B n}
      → n -≥ i ≡ just (n -Fin i)
      → M.map (lookup H) (n -≥ i) ≡ just A
      → M.map (lookup H) (just (n -Fin i)) ≡ just A
    p-rewrite x p rewrite x = p
    p' : just (lookup H (n -Fin i)) ≡ just A
    p' rewrite (p-rewrite {n} {i} n-i-just p) = refl

compile-nec-aux : ∀ {A : Set} {m : Maybe Fm} {a : Fm} {f} → M.map f m
≡ just a
  → (∀ {b} → (f b ≢ a)) → A
compile-nec-aux {A} {m} {a} {f} x k = Empty.⊥-elim (k (injective-just
  (subst (λ {z → M.map f z ≡ just a}) (proj₂ (aux {m = m} x)) x)))
  where aux : ∀ {m} → M.map f m ≡ just a → Σ _ λ b → m ≡ just b
        aux {just i} x = i , refl

compile-nec : ∀ {□A n C Σ}
  → (H : HilbertProof [] C n) → HilbertRef H □A □_ → Σ ⊢ □A
compile-nec {v !} H (ref i x) = absurd! {_} {lookup-may H i} {v} x
  where
  absurd! : ∀ {A : Set} {a b} → M.map □_ a ≡ just (b !) → A
  absurd! {_} {just y} {b} x with injective-just (Eq.sym x)
  ... | ()
compile-nec {a ⇝ b} H (ref i x) = absurd⇝ {_} {lookup-may H i} {a} {b}
x
  where
  absurd⇝ : ∀ {A : Set} {a b c} → M.map □_ a ≡ just (b ⇝ c) → A
  absurd⇝ {A} {just y} {b} x with injective-just (Eq.sym x)
  ... | ()
compile-nec {⊥'} H (ref i x) = absurd⊥ {_} {lookup-may H i} x
  where
  absurd⊥ : ∀ {A : Set} {a} → M.map □_ a ≡ just ⊥' → A
  absurd⊥ {_} {just y} x with injective-just (Eq.sym x)
  ... | ()
compile-nec {(A ⇝ ⊥') ▷ ⊥'} {n} H (ref i x) =
   nec (subst-⊢ (lookup-may-just {n} {i} (aux□ x)) (compile-instr H (n
-Fin i)))
  where
  aux□ : ∀ {a b} → M.map □_ a ≡ just (¬' b ▷ ⊥') → a ≡ just b
  aux□ {just a} refl = refl
compile-nec {(_ !) ▷ ⊥'} H (ref i x) = compile-nec-aux {m = lookup-may
H i} x λ ()
compile-nec {⊥' ▷ ⊥'} H (ref i x) = compile-nec-aux {m = lookup-may H
i} x λ ()
compile-nec {(A ⇝ (_ !)) ▷ ⊥'} H (ref i x) = compile-nec-aux {m =
lookup-may H i} x λ ()
compile-nec {(A ⇝ B ⇝ C) ▷ ⊥'} H (ref i x) = compile-nec-aux {m =
lookup-may H i} x λ ()
compile-nec {(A ⇝ B ▷ C) ▷ ⊥'} H (ref i x) = compile-nec-aux {m =
lookup-may H i} x λ ()
compile-nec {(A ▷ B) ▷ ⊥'} H (ref i x) = compile-nec-aux {m = lookup-may
H i} x λ ()
compile-nec {A ▷ (_ !)} H (ref i x) = compile-nec-aux {m = lookup-may
H i} x λ ()
compile-nec {A ▷ (B ⇝ C)} H (ref i x) = compile-nec-aux {m = lookup-may
H i} x λ ()
compile-nec {A ▷ (C ▷ D)} H (ref i x) = compile-nec-aux {m = lookup-may
H i} x λ ()

compile-mp : ∀ {A B n Σ C}
  → (H : HilbertProof Σ C n) → HilbertRef H (A ⇝ B) id → HilbertRef H
A id
  → Σ ⊢ B
compile-mp {A} {B} {n} {Σ} H (ref i pi) (ref j pj) = MP Σ⊢A⇝B Σ⊢A
  where
  map-id : ∀ {A : Set} {may : Maybe A} → M.map id may ≡ may
  map-id {A} {just x₁} = refl
  map-id {A} {nothing} = refl
  pi' : lookup-may H i ≡ just (A ⇝ B)
  pi' = Eq.trans (Eq.sym map-id) pi
  pj' : lookup-may H j ≡ just A
  pj' = Eq.trans (Eq.sym map-id) pj
  Σ⊢A : Σ ⊢ A
  Σ⊢A = subst-⊢ (lookup-may-just {n} {j} pj') (compile-instr H (n -Fin
j))
  Σ⊢A⇝B : Σ ⊢ A ⇝ B
   Σ⊢A⇝B = subst-⊢ (lookup-may-just {n} {i} pi') (compile-instr H (n
-Fin i))

compile-instr (begin x) zero = x
compile-instr (begin x) (suc ())
compile-instr (by x H) zero = x
compile-instr (by x H) (suc i) = compile-instr H i
compile-instr (MP {n} H i j) zero = compile-mp H i j
compile-instr (MP H _ _) (suc l) = compile-instr H l
compile-instr (Ax A x) zero = Ax here
compile-instr (Ax A H) (suc i) = l-weakening (compile-instr H i)
compile-instr {suc n} {Σ} (nec {Σ} {n} {_} {□A} H i) zero = compile-nec
H i
compile-instr (nec {_} {n} H i) (suc j) = l-weakening[] (compile-instr
H j)

compile : ∀ {n Σ A} → HilbertProof Σ A n → Σ ⊢ A
compile H = compile-instr H zero

injective-□ : ∀ {A B} → □ A ≡ □ B → A ≡ B
injective-□ refl = refl

instance
  NumHilbertRef : ∀ {A Σ n} {H : HilbertProof Σ A n} {fB} {f} → Number
(HilbertRef H fB f)
  NumHilbertRef {A} {Σ} {n} {H} {fB} {f} .Number.Constraint t = M.map
```

```
f (lookup-may H t) ≡ just fB
  NumHilbertRef {A} {Σ} {n} {H} .Number.fromNat t ⟦ x ⟧ = ref t x

begin[_]_By_ : ∀ {Σ} → Single {Nat} 0 → (B : Fm) → Σ ⊢ B → HilbertProof
Σ B 0
begin[ z ] B By p = begin p

infixl 10 _[_]_By_
_[_]_By_ : ∀ {Σ C n} → (H : HilbertProof Σ C n) → Single {Nat} (suc n)
→ (B : Fm) → Σ ⊢ B → HilbertProof Σ B (suc n)
H [ n ] B By p = by p H

infixl 10 _[_]_ByNec_
_[_]_ByNec_ : ∀ {C n}
  → (H : HilbertProof [] C n) → Single {Nat} (suc n)
  → (□B : Fm) → HilbertRef H □B □_
  → HilbertProof [] □B (suc n)
H [ i ] □A ByNec ix = nec H ix

infixl 10 _[_]_ByMP_,_
_[_]_ByMP_,_ : ∀ {Σ C A n} → (H : HilbertProof Σ C n) → Single {Nat}
(suc n)
  → (B : Fm) (i : HilbertRef H (A ⤳ B) id) (j : HilbertRef H A id) →
HilbertProof Σ B (suc n)
H [ n ] B ByMP i , j = MP H i j

infix 0 _■
_■ : ∀ {n Σ A} → (H : HilbertProof Σ A n) → Σ ⊢ A
H ■ = compile H

⊢A⤳A : ∀ {A} → [] ⊢ A ⤳ A
⊢A⤳A {A} =
  begin[ 0 ] (A ⤳ ((A ⤳ A) ⤳ A)) ⤳ ((A ⤳ (A ⤳ A)) ⤳ (A ⤳ A)) By C2
       [ 1 ] A ⤳ ((A ⤳ A) ⤳ A)                                   By C1
       [ 2 ] A ⤳ (A ⤳ A)                                         By C1
      [ 3 ] (A ⤳ (A ⤳ A)) ⤳ A ⤳ A                            ByMP 0 , 1
      [ 4 ] A ⤳ A                                             ByMP 3 , 2
      ■

⊢A▷A : ∀ {A} → [] ⊢ A ▷ A
⊢A▷A {A} =
  begin[ 0 ] A ⤳ A                  By ⊢A⤳A
       [ 1 ] □ (A ⤳ A)              ByNec 0
       [ 2 ] □ (A ⤳ A) ⤳ (A ▷ A)  By J1
       [ 3 ] A ▷ A                  ByMP 2 , 1
       ■

⊢A▷A' : ∀ {A} → HilbertProof [] (A ▷ A) 3
⊢A▷A' {A} =
  begin[ 0 ] A ⤳ A                  By ⊢A⤳A
       [ 1 ] □ (A ⤳ A)              ByNec 0
```

```
    [ 2 ] □ (A ⇾ A) ⇾ (A ▷ A)  By J1
    [ 3 ] A ▷ A                 ByMP 2 , 1
```

B.23. IL/Properties

```
module IL.Properties where

open import Function.Equivalence using (_⇔_; equivalence; module Equivalence)

open import Agda.Builtin.Unit using (⊤; tt)
open import Agda.Primitive using (Level; lzero; lsuc; _⊔_)
open import Data.Empty using (⊥; ⊥-elim)
open import Data.List using (map; List; []; _∷_)
open import Data.List.Membership.Propositional using (_∈_)
open import Data.List.Relation.Unary.All using (All; []; _∷_)
open import Relation.Binary using (REL; Rel; Transitive)
import Data.List.Relation.Unary.Any as Any
open import Data.Product using (Σ; proj₁; proj₂; _×_) renaming (_,_ to
_,_)
open import Data.Sum using (_⊎_; inj₁; inj₂; [_,_])
open import Function using (_∘_; const; case_of_; id)
open import Relation.Binary.PropositionalEquality using (_≡_; refl)
open import Relation.Unary using (Pred; Decidable)
open import Data.List.Relation.Unary.Any using () renaming (here to
here'; there to there')

import GeneralizedVeltmanSemantics as G
open import Base using (_⇒_; _⇐_; Decidable₃; Rel₃; REL₃)
open import Classical using () renaming (_⊢_ to _⊢c_; Fm to Fmc)
open import Formula using (Fm; _▷_; _⇾_; □_; _∧_; _∨_; _↔_; ◇_; ¬'_;
⊥'; ⊤'; Var)
open import IL
import OrdinaryVeltmanSemantics as O

here : ∀ {A : Fm} {Σ : List Fm} → A ∈ (A ∷ Σ)
here {A} {Σ} = here' refl

there : ∀ {A : Fm} {Σ B} → A ∈ Σ → A ∈ (B ∷ Σ)
there x = there' x

fmc : Fmc → Fm
fmc (Fmc.var x) = Fm.var x
fmc Fmc.⊥' = ⊥'
fmc (x Fmc.⇾ y) = (fmc x) ⇾ fmc y

weak : ∀ {A B Π} → Π ⊢ A → (B ∷ Π) ⊢ A
weak (_⊢_.Ax x) = _⊢_.Ax (there x)
weak C1 = C1
weak C2 = C2
weak C3 = C3
weak K = K
weak L = L
weak J1 = J1
weak J2 = J2
weak J3 = J3
weak J4 = J4
weak J5 = J5
weak (MP x y) = MP (weak x) (weak y)
weak (nec x) = nec x

⊢c→⊢ : ∀ {A Π} → Π ⊢c A → map fmc Π ⊢ fmc A
⊢c→⊢ _⊢c_.C1 = C1
⊢c→⊢ _⊢c_.C2 = C2
⊢c→⊢ _⊢c_.C3 = C3
⊢c→⊢ (_⊢c_.MP x x₁) = MP (⊢c→⊢ x) (⊢c→⊢ x₁)
⊢c→⊢ (_⊢c_.Ax (Any.here refl)) = _⊢_.Ax here
⊢c→⊢ (_⊢c_.Ax (Any.there x)) = weak (⊢c→⊢ (_⊢c_.Ax x))

val : (Var → Fm) → Fm → Fm
val f (Fm.var v) = f v
val f ⊥' = ⊥'
val f (a ↝ b) = val f a ↝ val f b
val f (a ▷ b) = val f a ▷ val f b

-- structurality of ⊢
⊢struct : ∀ {A Π} → (f : Var → Fm) → Π ⊢ A → map (val f) Π ⊢ val f A
⊢struct f (Ax (here' refl )) = Ax here
⊢struct f (Ax (there' x)) = weak (⊢struct f (Ax x))
⊢struct f C1 = C1
⊢struct f C2 = C2
⊢struct f C3 = C3
⊢struct f K = K
⊢struct f L = L
⊢struct f J1 = J1
⊢struct f J2 = J2
⊢struct f J3 = J3
⊢struct f J4 = J4
⊢struct f J5 = J5
⊢struct f (MP x x₁) = MP (⊢struct f x) (⊢struct f x₁)
⊢struct f (nec x) = nec (⊢struct f x)

deduction : ∀ {Π A B} → Π ⊢ A ↝ B ⇔ (A :: Π) ⊢ B
deduction = equivalence ⇒ ⇐
  where
  ⇐ : ∀ {Π A B} → (A :: Π) ⊢ B → Π ⊢ A ↝ B
  ⇐ {_} {A} {B} (Ax (here' refl )) = MP (MP C2 C1) (C1 {_} {_} {⊥'})
  ⇐ (Ax (there' x)) = MP C1 (Ax x)
  ⇐ C1 = MP C1 C1
  ⇐ C2 = MP C1 C2
  ⇐ C3 = MP C1 C3
  ⇐ K = MP C1 K
  ⇐ L = MP C1 L
  ⇦ J1 = MP C1 J1
  ⇦ J2 = MP C1 J2
  ⇦ J3 = MP C1 J3
  ⇦ J4 = MP C1 J4
  ⇦ J5 = MP C1 J5
  ⇦ (MP x y) = MP (MP C2 (⇦ x)) (⇦ y)
  ⇦ (nec x) = MP C1 (nec x)
  ⇨ : ∀ {Π A B} → Π ⊢ A ↝ B → (A :: Π) ⊢ B
  ⇨ x = MP (weak x) (Ax here)

cut : ∀ {Π A B} → Π ⊢ B → (B :: Π) ⊢ A → Π ⊢ A
cut x (Ax (here' refl)) = x
cut x (Ax (there' z)) = Ax z
cut x C1 = C1
cut x C2 = C2
cut x C3 = C3
cut x K = K
cut x L = L
cut x J1 = J1
cut x J2 = J2
cut x J3 = J3
cut x J4 = J4
cut x J5 = J5
cut x (MP y y₁) = MP (cut x y) (cut x y₁)
cut x (nec y) = nec y

⊢A↝A : ∀ {A Π} → Π ⊢ A ↝ A
⊢A↝A = deduction ⇐ (Ax here)

A↝A : ∀ {Π A} → Π ⊢ A ↝ A
A↝A {Π} {A} = MP (MP (C2 {Π} {A} {A ↝ A} {A}) (C1 {Π} {A} {A ↝ A})) (C1
{Π} {A} {A})

⊢A↝A' : ∀ {A Π} → Π ⊢ A ↝ A
⊢A↝A' {A} = MP (MP C2 C1) (C1 {B = A})

⊢A▷A : ∀ {A Π} → Π ⊢ A ▷ A
⊢A▷A {A} = MP J1 (nec ⊢A↝A)

trans : ∀ {A B C Π} → Π ⊢ (A ↝ B) ↝ (B ↝ C) ↝ A ↝ C
trans {A} {B} {C} = deduction ⇐ (deduction ⇐ (deduction ⇐ (MP (Ax (there
here))
  (MP (Ax (there (there here))) (Ax here)))))

⊢⟦A↝B⟧↝⟦B↝C⟧↝A↝C : ∀ {A B C Π} → Π ⊢ (A ↝ B) ↝ (B ↝ C) ↝ A ↝ C
⊢⟦A↝B⟧↝⟦B↝C⟧↝A↝C = trans

⊢A↝¬¬A : ∀ {Π A} → Π ⊢ A ↝ ¬' ¬' A
⊢A↝¬¬A = deduction ⇐ (deduction ⇐ (MP (Ax here) (Ax (there here))))

⊢¬¬A↝A : ∀ {A Π} → Π ⊢ (¬' ¬' A) ↝ A
```

```
⊢¬¬A⤳A {A} = MP C3 ⊢A⤳¬¬A

⊢⟦A⤳B⟧⤳¬B⤳¬A : ∀ {A B Π} → Π ⊢ (A ⤳ B) ⤳ ¬' B ⤳ ¬' A
⊢⟦A⤳B⟧⤳¬B⤳¬A = deduction ⇐ (deduction ⇐ (deduction ⇐ (MP (Ax (there
here))
   (MP (Ax (there (there here))) (Ax here)))))

⊢A⤳⊤ : ∀ {Π A} → Π ⊢ A ⤳ ⊤'
⊢A⤳⊤ = deduction ⇐ (deduction ⇐ (Ax here))

⊢⊥⤳A : ∀ {Π A} → Π ⊢ ⊥' ⤳ A
⊢⊥⤳A = MP C3 ⊢A⤳⊤

⊢¬A⤳A⤳B : ∀ {A B Π} → Π ⊢ ¬' A ⤳ A ⤳ B
⊢¬A⤳A⤳B = MP (MP C2 (MP C1 C3)) C1

⊢⤳ : ∀ {A B Π} → (Π ⊢ ¬' A ⊎ Π ⊢ B) → Π ⊢ A ⤳ B
⊢⤳ (inj₁ x) = MP C3 (MP C1 x)
⊢⤳ (inj₂ y) = MP C1 y

⊢A∧B⤳A : ∀ {Π A B} → Π ⊢ A ∧ B ⤳ A
⊢A∧B⤳A = MP (MP trans (MP ⊢⟦A⤳B⟧⤳¬B⤳¬A ⊢¬A⤳A⤳B)) ⊢¬¬A⤳A

⊢A∧B⤳B : ∀ {Π A B} → Π ⊢ A ∧ B ⤳ B
⊢A∧B⤳B = MP (MP trans (MP ⊢⟦A⤳B⟧⤳¬B⤳¬A C1)) ⊢¬¬A⤳A

⊢⟦A⤳B⤳C⟧⤳B⤳A⤳C : ∀ {A B C Π} → Π ⊢ (A ⤳ B ⤳ C) ⤳ B ⤳ A ⤳ C
⊢⟦A⤳B⤳C⟧⤳B⤳A⤳C = deduction ⇐ (deduction ⇐ (deduction ⇐ (cut
   (MP (Ax (there (there here))) (Ax here)) (MP (Ax here)
   (Ax (there (there here)))))))

⊢A⤳B⤳A∧B : ∀ {Π A B} → Π ⊢ A ⤳ B ⤳ A ∧ B
⊢A⤳B⤳A∧B = deduction ⇐ (deduction ⇐ (deduction ⇐ (cut (MP (Ax here)
   (Ax (there (there here)))) (MP (Ax here)
   (Ax (there (there here)))))))

⊢∧ : ∀ {Π A B} → Π ⊢ A ∧ B ⇔ (Π ⊢ A × Π ⊢ B)
⊢∧ {Π} {A} {B} = equivalence ⇒ ⇐
  where
  ⇐ : (Π ⊢ A × Π ⊢ B) → Π ⊢ A ∧ B
  ⇐ (fst , snd) = MP (MP ⊢A⤳B⤳A∧B fst) snd
  ⇒ : Π ⊢ A ∧ B → (Π ⊢ A × Π ⊢ B)
  ⇒ x = MP ⊢A∧B⤳A x , MP ⊢A∧B⤳B x

⊢A⤳A∨B : ∀ {A B Π} → Π ⊢ A ⤳ A ∨ B
⊢A⤳A∨B = deduction ⇐ (deduction ⇐ (MP ⊢⊥⤳A (MP (Ax here)
   (Ax (there here)))))

K1 : ∀ {Π A} → Π ⊢ A ▷ (A ∨ ◊ A) × Π ⊢ (A ∨ ◊ A) ▷ A
K1 {Π} {A} = ⇒ , ⇐
  where
  ⇨ : Π ⊢ A ▷ (A ∨ ◊ A)
  ⇨ = MP J1 (nec ⊢A⇾A∨B)
  ⇦ : Π ⊢ (A ∨ ◊ A) ▷ A
  ⇦ = MP J3 (⊢∧ ⇐ (⊢A▷A , J5))

A⇾B⇒□A⇾□B : ∀ {A B Π} → [] ⊢ A ⇾ B → Π ⊢ □ A ⇾ □ B
A⇾B⇒□A⇾□B x = MP K (nec x)

A⇔B⇒□A⇔□B : ∀ {A B Π} → [] ⊢ A ⇔ B → Π ⊢ □ A ⇔ □ B
A⇔B⇒□A⇔□B x = ⊢∧ ⇐ (case ⊢∧ ⇒ x of λ {(fst , snd) → A⇾B⇒□A⇾□B fst ,
A⇾B⇒□A⇾□B snd})

⊢□⟦A∧B⟧⇔⟦□A∧□B⟧ : ∀ {A B Π} → Π ⊢ □ (A ∧ B) ⇔ (□ A ∧ □ B)
⊢□⟦A∧B⟧⇔⟦□A∧□B⟧ {A} {B} {Π} = ⊢∧ ⇐ (⇨ , ⇦)
  where
  ⇨ : Π ⊢ □ (A ∧ B) ⇾ □ A ∧ □ B
  ⇨ = deduction ⇐ cut (MP (A⇾B⇒□A⇾□B ⊢A∧B⇾A) (Ax here))
     (cut (MP (A⇾B⇒□A⇾□B ⊢A∧B⇾B) (Ax (there here)))
    (⊢∧ ⇐ ((Ax (there here)) , (Ax here))))
  ⇦ : Π ⊢ □ A ∧ □ B ⇾ □ (A ∧ B)
   ⇦ = deduction ⇐ MP (MP K (MP (A⇾B⇒□A⇾□B  ⊢A⇾B⇾A∧B) (MP ⊢A∧B⇾A (Ax
here))))
     (MP ⊢A∧B⇾B (Ax here))

A⇾B⇒◊A⇾◊B : ∀ {A B Π} → [] ⊢ A ⇾ B → Π ⊢ ◊ A ⇾ ◊ B
A⇾B⇒◊A⇾◊B x = deduction ⇐ (deduction ⇐
  MP (Ax (there (here))) (MP (A⇾B⇒□A⇾□B (MP ⊢⟦A⇾B⟧⇾¬B⇾¬A x)) (Ax here)))

A⇔B⇒◊A⇔◊B : ∀ {A B Π} → [] ⊢ A ⇔ B → Π ⊢ ◊ A ⇔ ◊ B
A⇔B⇒◊A⇔◊B x = ⊢∧ ⇐ (case ⊢∧ ⇒ x of λ {(fst , snd) → A⇾B⇒◊A⇾◊B fst ,
A⇾B⇒◊A⇾◊B snd})

⊢¬⟦A∧B⟧⇔¬A∨¬B : ∀ {A B Π} → Π ⊢ ¬' (A ∧ B) ⇔ ¬' A ∨ ¬' B
⊢¬⟦A∧B⟧⇔¬A∨¬B {A} {B} {Π} = ⊢∧ ⇐ (⇨ , ⇦)
  where
  ⇨ : Π ⊢ ¬' (A ∧ B) ⇾ ¬' A ∨ ¬' B
  ⇨ = deduction ⇐ (deduction ⇐  MP (MP ⊢¬¬A⇾A (Ax (there here)))
     (MP ⊢¬¬A⇾A (Ax here)))
  ⇦ : Π ⊢ ¬' A ∨ ¬' B ⇾ ¬' (A ∧ B)
  ⇦ = deduction ⇐ (deduction ⇐ MP (MP (Ax (there here))
    (MP ⊢A⇾¬¬A (MP ⊢A∧B⇾A (Ax here)))) (MP ⊢A∧B⇾B (Ax here)))

L2 : ∀ {A B Π} → Π ⊢ □ (A ⇾ B) ⇾ (□ A ⇾ □ B)
L2 = K

L4 : ∀ {Π A} → Π ⊢ □ (□ A ⇾ A) ⇾ □ A
L4 = L

⟦A∨¬'B⟧⇾⟦A∧B∨¬'B⟧ : ∀ {A B Π} → Π ⊢ (A ∨ ¬' B) ⇾ (A ∧ B ∨ ¬' B)
⟦A∨¬'B⟧⇾⟦A∧B∨¬'B⟧ = deduction ⇐ (deduction ⇐ (deduction ⇐ (
   (cut (MP C3 (Ax (there (there here))))
   (cut (MP (Ax here) (Ax (there here)))
   (cut (MP ⊢¬¬A↝A (Ax (there (there (there here)))))
   (cut (MP (Ax here) (Ax (there here))) (MP (Ax here)
   (Ax (there (there (there (there here)))))))))))


All∈ : ∀ {ℓ₂} {P : Pred Fm ℓ₂} {L a} → a ∈ L → All P L → P a
All∈ (here' refl) (px ∷ ay) = px
All∈ (there' x) (px ∷ ay) = All∈ x ay

variable
  ℓW ℓR ℓS : Level

module OrdSoundness
  {W : Set lzero}
  {R : Rel W lzero}
  {S : Rel₃ W lzero}
  {V : REL W Var lzero}
  {M : O.Model W R S V}
  (M,_⊩?_ : O.DecidableModel M)
  (∈?S : Decidable₃ S)
  where

  open import OrdinaryVeltmanSemantics
   using (Model; _,_⊩_; _,_⊮_; impl; rhd; bot; DecidableModel; _,_⊩*_)
  open import OrdinaryVeltmanSemantics.Properties
   using (⊩MP; ⊩¬; ⊮→¬⊩; ⊩□; ⊩◇; ⊮¬; module Extended; module Extended2)

  open Extended M,_⊩?_ ∈?S
  open Extended2 M,_⊩?_ ∈?S

  soundness : ∀ {w A Π} → Π ⊢ A → M , w ⊩* Π → M , w ⊩ A
  soundness {w} (C1 {A}) p = ⊩↝ ⇐ λ x → impl (inj₂ x)
  soundness {w} (C2 {A} {B} {C}) p = ⊩↝ ⇐ λ x → ⊩↝ ⇐ λ y → ⊩↝ ⇐ λ z →
⊩MP (⊩MP x z) (⊩MP y z)
  soundness {w} (C3 {A} {B}) p = ⊩↝ ⇐
    λ {x → ⊩↝ ⇐ λ y → case x of λ { (impl (inj₁ x)) → ⊮¬ ⇒ x
    ; (impl (inj₂ z)) → ⊥-elim (⊮→¬⊩ (⊩¬ ⇒ z) y)}}
  soundness {w} K p = ⊩K
  soundness {w} L p = ⊩L
  soundness {w} J1 p = ⊩J1
  soundness {w} J2 p = ⊩J2
  soundness {w} J3 p = ⊩J3
  soundness {w} J4 p = ⊩J4
  soundness {w} J5 p = ⊩J5
  soundness {w} (MP x y) p = ⊩MP (soundness x p) (soundness y p)
  soundness {w} (nec x) p = ⊩□ ⇐ λ {u} wRu → soundness x []
  soundness {w} (Ax x) p = All∈ x p

module GenSoundness-All
  {W : Set ℓW}
  {S : REL₃ W W (Pred W ℓW) ℓS}
  (T : ∀ {ℓW ℓS} (W : Set ℓW) → REL₃ W W (Pred W ℓW) ℓS → Set (lsuc ℓW
⊔ ℓS))
  where
   open import GeneralizedVeltmanSemantics.Properties using (module
PGeneric)
  open PGeneric T
  module Soundness
    {W R S V}
    {M : Model {lzero} {lzero} {lzero} W R S V}
    (M,_*⊩?_ : MultiDecidableModel M)
    (∈S? : Decidable₃ S)
    (∈SV? : ∀ {w u Y} → S w u Y → Decidable Y)
    (⊩J2 : ∀ {w A B C} → M , w ⊩ (A ▷ B ∧ B ▷ C) ↝ A ▷ C)
    where

    open Extended M,_*⊩?_ ∈S? ∈SV?
    open Extended2 M,_*⊩?_ ∈S? ∈SV?

    soundness : ∀ {w A Π} → Π ⊢ A → M , w ⊩* Π → M , w ⊩ A
    soundness {w} (C1 {A}) p = ⊩↝ ⇐ λ x → impl (inj₂ x)
    soundness {w} C2 p = ⊩↝ ⇐ λ x → ⊩↝ ⇐ λ y → ⊩↝ ⇐ λ z → ⊩MP (⊩MP x
z) (⊩MP y z)
     soundness {w} C3 p = ⊩↝ ⇐ λ {x → ⊩↝ ⇐ λ y → case x of λ { (impl
(inj₁ x)) → ⊩¬ ⇒ x
      ; (impl (inj₂ z)) → ⊥-elim (⊮→¬⊩ (⊩¬ ⇒ z) y)}}
    soundness {w} K p = ⊩K
    soundness {w} L p = ⊩L
    soundness {w} J1 p = ⊩J1
    soundness {w} J2 p = ⊩J2
    soundness {w} J3 p = ⊩J3
    soundness {w} J4 p = ⊩J4
    soundness {w} J5 p = ⊩J5
    soundness {w} (MP x y) p = ⊩MP (soundness x p) (soundness y p)
    soundness {w} (nec x) p = ⊩□ ⇐ λ {u} wRu → soundness x []
    soundness {w} (Ax x) p = All∈ x p

module GenSoundness-L
  {W R S V}
  {M : G.Model {lzero} {lzero} {lzero} W R S V}
  (M,_*⊩?_ : G.MultiDecidableModel M)
  (∈S? : Decidable₃ S)
  (∈SV? : ∀ {w u Y} → S w u Y → Decidable Y)
  where
   open import GeneralizedVeltmanSemantics.Properties using (module
Extended; module Extended2; module ExtendedT2; _,_⊩*_; _,_⊩_)
  open import GeneralizedFrame using (module Trans-conditions)
  open Trans-conditions using (Trans-L)

  open ExtendedT2 M,_*⊩?_ ∈S? ∈SV?
 open GenSoundness-All {W = W} {S = S} Trans-L using (module Soundness)
```

```
  open Soundness M,_*⊩?_ ∈S? ∈SV? ⊩J2 renaming (soundness to s)

  soundness : ∀ {w A Π} → Π ⊢ A → M , w ⊩* Π → M , w ⊩ A
  soundness {w} A p = s A p
```

B.24. OrdinaryFrame

```
module OrdinaryFrame where

open import Agda.Builtin.Nat using (Nat; suc; _+_)
open import Agda.Primitive using (Level; lzero; lsuc; _⊔_)
open import Data.Empty using (⊥; ⊥-elim)
open import Data.List using (List)
open import Data.List.Relation.Unary.All using (All)
open import Data.Product using (Σ; proj₁; proj₂; _×_; _,_)
open import Data.Sum using (_⊎_; inj₁; inj₂)
open import Relation.Nullary using (yes; no; ¬_)
open import Function using (_∘_; case_of_; _$_)
open import Relation.Binary using (REL; Rel; Transitive; Reflexive)
renaming (Decidable to Decidable₂)
open import Relation.Binary.PropositionalEquality using (_≡_; _≢_; refl;
subst; cong)
open import Relation.Nullary using (¬_)
open import Relation.Unary using (Pred; _∈_; _∉_; Decidable; {_}; _∩_;
_⊆_; Satisfiable)

open import Formula using (Fm; Var; _↝_; ⊥'; _▷_; var; ¬'_)
open import Base

private
  variable
    ℓW ℓR ℓS : Level

record Frame (W : Set ℓW) (R : Rel W ℓR) (S : Rel₃ W ℓS)
  : Set (ℓW ⊔ ℓR ⊔ ℓS) where
  constructor frame
  field
    witness : W
    R-trans : Transitive R
    R-noetherian : Noetherian R
    Sw⊆R[w]² : ∀ {w u v} → S w u v → R w u × R w v
    Sw-refl : ∀ {w u} → R w u → S w u u
    Sw-trans : ∀ {w} → Transitive (S w)
    R-Sw-trans : ∀ {w u v} → R w u → R u v → S w u v
```

B.25. OrdinaryVeltmanSemantics/Finite

```
module OrdinaryVeltmanSemantics.Finite where

open import Function.Equivalence using (_⇔_; equivalence; module Equivalence)
open import Function.Bijection using (Bijective; Bijection)
open import Function.Surjection using (Surjective)

open import Agda.Builtin.Nat using (Nat; suc; zero)
open import Agda.Builtin.Unit using (⊤; tt)
open import Agda.Primitive using (Level; lzero; lsuc)
open import Data.Empty using (⊥; ⊥-elim)
open import Data.Fin using (Fin; fromℕ; inject₁; zero; suc; lower₁;
toℕ)
open import Data.Fin.Properties using (all?; any?; ¬∀→∃¬)
open import Data.List using (List; []; _∷_; map)
open import Data.List.Membership.Propositional using (_∈_)
open import Data.List.Relation.Unary.All using (All; _∷_; [])
open import Data.List.Relation.Unary.Any using (Any; here; there) renaming
(map to Any-map)
open import Data.Product using (∃; Σ; proj₁; proj₂; _×_) renaming (_,_
to _,_)
open import Data.Sum using (_⊎_; inj₁; inj₂)
open import Function using (_∘_; const; case_of_; id)
open import Function.Equality using (_⟨$⟩_)
open import Relation.Binary using (REL; Rel; Transitive) renaming
(Decidable to Decidable₂)
open import Relation.Binary.PropositionalEquality using (_≡_; _≢_; refl;
setoid; subst; subst₂; sym; cong; trans)
open import Relation.Binary.HeterogeneousEquality using (_≅_; ≅-to-
type-≡) renaming (subst to hsubst;
  trans to htrans; sym to hsym; subst₂ to hsubst₂; refl to hrefl; cong
to hcong)
open import Relation.Nary using (substₙ)
open import Relation.Nullary using (Dec; yes; no; ¬_)
open import Relation.Nullary.Negation using (¬∃→∀¬)
open import Relation.Unary using (Pred; Decidable; _∩_; _∪_; _⊆_; ∁)
renaming (_⇒_ to _P⇒_)
open import Relation.Unary.Properties using (_∩?_; _∪?_; _×?_; ∁?)

open import Formula using (Fm; Var; _↝_; ⊥'; _▷_; var; ⊤'; ¬'_; □_; ◇_;
_∧_; _∨_; car)
open import OrdinaryVeltmanSemantics using (Frame; Model; _,_⊮_; _,_⊩_;
impl;
  var; rhd; bot; _*⊩_; _*⊮_; Valuation; model; DecidableModel)
open import OrdinaryVeltmanSemantics.Properties using (⊮→¬⊩; ⊩→¬⊮)
open import Base
open import OrdinaryFrame

private
  variable
    ℓW ℓR ℓS : Level

Finite : ∀ {W} {R} {S} → Pred (Frame {ℓW} {ℓR} {ℓS} W R S) ℓW
Finite {W = W} F = Σ Nat λ n → Bijection (setoid W) (setoid (Fin n))

module DecideModel
```

```
  {W : Set ℓW}
  {R : Rel W ℓR}
  {S : Rel₃ W ℓR}
  {V : REL W Var lzero}
  (M : Model W R S V)
  (fin : Finite (Model.F M))
  (V? : Decidable₂ V)
  (S? : Decidable₃ S)
  (R? : Decidable₂ R) where
  F = Model.F M
  open Frame F

  n : Nat
  n = proj₁ fin

  bij : Bijection (setoid W) (setoid (Fin n))
  bij = proj₂ fin

  f : W → Fin n
  f w = Bijection.to bij ⟨$⟩ w

  g : Fin n → W
   g m = Surjective.from (Bijective.surjective (Bijection.bijective
bij)) ⟨$⟩ m

  fg : ∀ x → f (g x) ≡ x
 fg = Surjective.right-inverse-of (Bijective.surjective (Bijection.bijective
bij))

  gf : ∀ x → g (f x) ≡ x
  gf = Bijective.left-inverse-of (Bijection.bijective bij)

  R' : Rel (Fin n) _
  R' x y = R (g x) (g y)

  R'? : Decidable₂ R'
  R'? x y with R? (g x) (g y)
  ... | yes t = yes t
  ... | no n = no n

  S' : Rel₃ (Fin n) _
  S' x y z = S (g x) (g y) (g z)

  S'? : Decidable₃ S'
  S'? x y z with S? (g x) (g y) (g z)
  ... | yes p = yes p
  ... | no p = no p

  fR : ∀ {w u} → R w u → R' (f w) (f u)
  fR {w} {u} = subst₂ R (sym (gf w)) (sym (gf u))
```

```
fR2 : ∀ {w u} → R w u → R (g (f w)) (g (f u))
fR2 {w} {u} = subst₂ R (sym (gf w)) (sym (gf u))

R'-trans : Transitive R'
R'-trans = Frame.R-trans F

≡-subst₂-removable : ∀ {p a} {A : Set a} (R : Rel A p) {x y x' y'}
  (eq1 : x ≡ y) (eq2 : x' ≡ y') z → subst₂ R eq1 eq2 z ≅ z
≡-subst₂-removable R₁ refl refl z = hrefl

g-chain : ∀ {a} → InfiniteChain R' a → InfiniteChain R (g a)
InfiniteChain.b (g-chain x) = g (InfiniteChain.b x)
InfiniteChain.a<b (g-chain x) = InfiniteChain.a<b x
InfiniteChain.tail (g-chain x) = g-chain (InfiniteChain.tail x)

R'-noetherian : Noetherian R'
R'-noetherian x = R-noetherian (g-chain x)

S'w⊆R[w]² : ∀ {w u v} → S' w u v → R' w u × R' w v
S'w⊆R[w]² = Frame.Sw⊆R[w]² F

S'w-refl : ∀ {w u} → R' w u → S' w u u
S'w-refl = Frame.Sw-refl F

S'w-trans : ∀ {w} → Transitive (S w)
S'w-trans = Frame.Sw-trans F

R'-S'w-trans : ∀ {w u v} → R' w u → R' u v → S' w u v
R'-S'w-trans = Frame.R-Sw-trans F

F' : Frame (Fin n) R' S'
F' = frame (f (Frame.witness F)) R'-trans R'-noetherian S'w⊆R[w]²
    S'w-refl S'w-trans R'-S'w-trans

V' : Valuation F'
V' w x = V (g w) x

V'? : Decidable₂ V'
V'? x y with V? (g x) y
... | yes p = yes p
... | no p = no p

M' : Model (Fin n) R' S' V'
M' = model {V = V'} F'

g⊮ : ∀ {w A} → M' , w ⊮ A → M , g w ⊮ A
g⊩ : ∀ {w A} → M' , w ⊩ A → M , g w ⊩ A

g⊮ (var x) = var x
g⊮ (impl x y) = impl (g⊩ x) (g⊮ y)
g⊮ {w} (rhd {A} {B} (u , Ru , uA , snd)) = rhd (g u , Ru , g⊩ uA
  , λ v → case snd (f v) of λ {
  (inj₁ x) → inj₁ λ z → x (subst (S (g w) (g u)) (sym (gf _)) z) ;
  (inj₂ y) → inj₂ (subst (M ,_⊮ B) (gf _) (g⊮ y))})
g⊮ bot = bot

g⊩ (var x) = var x
g⊩ (impl (inj₁ x)) = impl (inj₁ (g⊮ x))
g⊩ (impl (inj₂ y)) = impl (inj₂ (g⊩ y))
g⊩ (rhd {A} x) = rhd (λ {u} y → case x (aux y) of
  λ { (inj₁ x) → inj₁ (subst (M ,_⊮ A) (gf u) (g⊮ x))
  ; (inj₂ (v , Sv , snd)) → inj₂ (g v , (aux' Sv , g⊩ snd))})
  where
  aux : ∀ {w u} → R (g w) u → R' w (f u)
  aux = subst₂ R refl (sym (gf _))
  aux' : ∀ {w u v} → S' w (f u) v → S (g w) u (g v)
  aux' {w} {u} {v} = subst (λ a → S (g w) a (g v)) (gf u)

M',_⊩?D_ : Decidable₂ (M' ,_⊩_)
M',_⊮?D_ : Decidable₂ (M' ,_⊮_)
M',_⊩?_ : DecidableModel M'

M', w ⊩?D A with M', w ⊩? A
... | inj₁ p = yes p
... | inj₂ p = no (⊮→¬⊩ p)

M', w ⊮?D A with M', w ⊩? A
... | inj₁ p = no (⊩→¬⊮ p)
... | inj₂ p = yes p


M', w ⊩? var x with V'? w x
... | yes y = inj₁ (var y)
... | no n = inj₂ (var n)
M', w ⊩? ⊥' = inj₂ bot
M', w ⊩? (A ⇝ B) with M', w ⊩? A | M', w ⊩? B
... | inj₂ y | b = inj₁ (impl (inj₁ y))
... | inj₁ x | inj₂ b = inj₂ (impl x b)
... | inj₁ x | inj₁ b = inj₁ (impl (inj₂ b))
M', w ⊩? (A ▷ B) = case A▷B of
  λ { (yes p) → inj₁ (rhd (λ {u} r → case M', u ⊩? A of
      λ { (inj₂ x) → inj₁ x
      ; (inj₁ y) → inj₂ (case p u of (
        λ { (inj₁ (inj₁ x)) → ⊥-elim (x r)
        ; (inj₁ (inj₂ t)) → ⊥-elim (⊩→¬⊮ y t)
        ; (inj₂ t) → t}))})) ;
    (no p) → inj₂ (rhd (case ¬∀→∃¬ n P' P? p of
      λ { (u , snd) → u , (case aux P1? P2? snd of
        λ { (p1 , p2) → case aux P11? P12? p1 of
          λ { (fst , snd) → ¬¬-elim {u} {R' w} (R'? w) fst
          , (case M', u ⊩? A of (
            λ { (inj₁ x) → x ; (inj₂ y) → ⊥-elim (snd y)}))
```

```
              , λ v → case aux2 (S'? w u) (M',_⊩?D B) (¬∃→∀¬ p2 v) of
                λ { (inj₁ x) → inj₁ x ; (inj₂ y) → inj₂ (case M', v ⊩?
B of (
                λ { (inj₁ x) → ⊥-elim (y x)
                  ; (inj₂ y) → y})) }}})})) }
    where
    P11 P12 P2 : Pred (Fin n) _
    P11 = 𝒞 (R' w)
    P12 = M' ,_⊮ A
    P1 = P11 ∪ P12
    P2 = (λ u → ∃ (λ v → S' w u v × M' , v ⊩ B))
    P1? : Decidable P1
    P11? : Decidable P11
    P11? = 𝒞? (R'? w)
    P12? : Decidable P12
    P12? = M',_⊮?D A
    P1? = P11? ∪? P12?
    P2? : Decidable P2
    P2? = λ u → any? (S'? w u ∩? (M',_⊩?D B))
    P' : Pred (Fin n) _
    P' = P1 ∪ P2

    P? : Decidable P'
    P? = P1? ∪? P2?

    aux : ∀ {u} → {P Q : Pred (Fin n) _} → Decidable P → Decidable Q
      → ¬ ((P ∪ Q) u) → ¬ P u × ¬ Q u
    aux {u} P? Q? ¬u = (λ x → ¬u (inj₁ x)) , λ x → ¬u (inj₂ x)

    aux2 : ∀ {u} → {P Q : Pred (Fin n) _} → Decidable P → Decidable Q
      → ¬ ((P ∩ Q) u) → ¬ P u ⊎ ¬ Q u
    aux2 {u} P? Q? ¬u with P? u | Q? u
    ... | no p | _ = inj₁ p
    ... | yes p | no n = inj₂ (λ x → ¬u (p , x))
    ... | yes p | yes n = inj₁ (λ x → ¬u (x , n))

    ¬¬-elim : ∀ {u} → {P : Pred (Fin n) _} → Decidable P → ¬ ¬ P u → P u
    ¬¬-elim {u} P? k with P? u
    ... | yes p = p
    ... | no p = ⊥-elim (k p)

    A▷B : Dec (∀ u → P' u)
    A▷B = all? P?

  -- Decides the original model
  M,_⊩?_ : DecidableModel M
  M, w ⊩? A with M', f w ⊩? A
  ... | inj₁ p = inj₁ (subst (M ,_⊩ A) (gf _) (g⊩ p))
  ... | inj₂ p = inj₂ (subst (M ,_⊮ A) (gf _) (g⊮ p))
```

B.26. OrdinaryVeltmanSemantics/Properties/M

```
module OrdinaryVeltmanSemantics.Properties.M where

open import Function.Equivalence using (_⇔_; equivalence; module Equivalence)

open import Agda.Builtin.Nat using (Nat; suc; zero)
open import Agda.Builtin.Unit using (⊤; tt)
open import Agda.Primitive using (Level; lzero; lsuc; _⊔_)
open import Data.Empty using (⊥; ⊥-elim)
open import Data.Product using (Σ; proj₁; proj₂; _×_) renaming (_,_ to
_,_)
open import Data.Sum using (_⊎_; inj₁; inj₂; [_,_])
open import Function using (_∘_; const; case_of_; id)
open import Function.Equality using (_⟨$⟩_)
open import Relation.Binary using (REL; Rel; Transitive)
open import Relation.Nullary using (yes; no; ¬_)
open import Relation.Unary using (Pred; _∈_; _∉_; Decidable; Satisfiable;
_⊆_; _∩_; {_})
open import Relation.Binary using (Irreflexive) renaming (Decidable to
Decidable₂)
open  import  Relation.Binary.PropositionalEquality  using  (_≡_;  refl;
subst; trans; sym)

open import Formula using (Fm; Var; _↝_; ⊥'; _▷_; var; ⊤'; ¬'_; □_; ◇_;
_∧_; _∨_; car)
open import OrdinaryVeltmanSemantics using (Frame; Model; _,_⊮_; _,_⊩_;
impl; var; rhd; bot; _*⊩_; _*⊮_; Valuation; model; DecidableModel)
open import OrdinaryVeltmanSemantics.Properties using (module Extended;
⊩∧; ⊩□; R-Irreflexive
  ; ⊩→¬⊮)
open import Base using (_⇒_; _⇐_; Decidable₃)
import Principles as P

private
  variable
    ℓW ℓR ℓS : Level

M-condition : ∀ {W R S} → Frame {ℓW} {ℓR} {ℓS} W R S → Set (ℓW ⊔ ℓR ⊔
ℓS)
M-condition {W = W} {R = R} {S = S} F = ∀ {w x y z} → S w x y → R y z
→ R x z
  where open Frame F


module M-soundness
  {W R S V}
  {M : Model {ℓW} {ℓR} {ℓS} W R S V}
  (M,_⊩?_ : DecidableModel M)
  (∈?S : Decidable₃ S)
  where
  open Model M
  open Frame F
  open Extended M,_⊩?_ ∈?S

  ⊩M : ∀ {w A B C} → M-condition (Model.F M) → M , w ⊩ A ▷ B ⇝ (A ∧ □
C) ▷ (B ∧ □ C)
  ⊩M {w} {A} {B} {C} c = ⊩⇝ ⇐ λ A▷B → ⊩▷ ⇐ λ Rwu x → case ⊩∧ ⇒ x of λ
{ (uA , u□C) →
    case (⊩▷ ⇒ A▷B) Rwu uA of λ { (z , Swuz , snd) → z , Swuz , (⊩∧ ⇐
(snd , (⊩□ ⇐
    (λ {v} Rzv → (⊩□ ⇒ u□C) (c Swuz Rzv)))))}}


module M-completeness
  {W R S}
  {F : Frame {lzero} {lzero} {lzero} W R S}
  (_≟_ : Decidable₂ {_} {W} _≡_)
  (R? : Decidable₂ R)
  (∈?S : Decidable₃ S)
  (dec : ∀ V → DecidableModel (model {V = V} F))
  where
  open Frame F

  F*⊩M : Set _
  F*⊩M = P.M (F *⊩_)

  -- If a frame satisfies the M principle for any valuation, then it
must
  -- satisfy the M condition.
  *⊩M : F*⊩M → M-condition F
  *⊩M f {w} {w₁} {w₂} {w₃} Sww₁w₂ Rw₂w₃ with R? w₁ w₃
  ... | yes y = y
  ... | no ¬Rw₁w₃ =
   -- ⊥-elim (case (⊩▷ ⇒ (⊩⇝ ⇒ (f (var a) (var b) (var c)) V w) w⊩A▷B)
Rww₁ w₁⊩A∧□C of
    ⊥-elim (case (⊩▷ ⇒ (⊩⇝ ⇒ f V w) w⊩A▷B) Rww₁ w₁⊩A∧□C of
    λ { (u , Sww₁w₂ , w₂⊩B∧□C) → case only-w₂⊩B (proj₁ (⊩∧ ⇒ w₂⊩B∧□C)
) of
    λ {refl →  let chk : model {V = V} F , w₂ ⊩ □ var c
                   chk = proj₂ (⊩∧ ⇒ w₂⊩B∧□C)
                in ⊩→¬⊮ ((⊩□ ⇒ chk) Rw₂w₃) (var w₃⊮C)}
    })
    where
    pattern a = 0
    pattern b = 1
    pattern c = 2
    V : Valuation F
    V z v with z ≟ w₁
    ... | yes refl = case v of λ { a → ⊤ ; (suc x) → ⊥}
    ... | no _ with z ≟ w₂
    ... | yes refl = case v of λ { a → ⊥ ; b → ⊤ ; (suc (suc x)) → ⊥}
    ... | no _ with z ≟ w₃
    ... | yes refl = ⊥
    ... | no _ with R? w₁ z
    ... | yes y = case v of λ {a → ⊥ ; b → ⊥ ; (suc (suc x)) → ⊤}
    ... | no _ = ⊥
    open Extended (dec V) ∈?S
    Rww₁ : R w w₁
    Rww₁ = proj₁ (Sw⊆R[w]² Sww₁w₂)
    w₃⊮C : c ∉ V w₃
    w₃⊮C x with w₃ ≟ w₁
    ... | yes refl = x
    ... | no _ with w₃ ≟ w₂
    ... | yes refl = x
    ... | no _ with w₃ ≟ w₃
    ... | yes refl = x
    ... | no n = n refl
    only-w₁⊩A : ∀ {u} → model {V = V} F , u ⊩ var a → u ≡ w₁
    only-w₁⊩A {u} (var x) with u ≟ w₁
    ... | yes refl = refl
    ... | no _ with u ≟ w₂
    ... | yes refl = ⊥-elim x
    ... | no _ with u ≟ w₃
    ... | yes refl = ⊥-elim x
    ... | no _ with R? w₁ u
    ... | yes r = ⊥-elim x
    ... | no _ = ⊥-elim x
    only-w₂⊩B : ∀ {u} → model {V = V} F , u ⊩ var b → u ≡ w₂
    only-w₂⊩B {u} (var x) with u ≟ w₁
    ... | yes refl = ⊥-elim x
    ... | no _ with u ≟ w₂
    ... | yes refl = refl
    ... | no _ with u ≟ w₃
    ... | yes refl = ⊥-elim x
    ... | no _ with R? w₁ u
    ... | yes r = ⊥-elim x
    ... | no nr = ⊥-elim x
    w₂⊩B : b ∈ V w₂
    w₂⊩B with w₂ ≟ w₁
    ... | yes refl = ⊥-elim (¬Rw₁w₃ Rw₂w₃)
    ... | no _ with w₂ ≟ w₂
    ... | yes refl = tt
    ... | no n = ⊥-elim (n refl)
    w⊩A▷B : model {V = V} F , w ⊩ var a ▷ var b
     w⊩A▷B = ⊩▷ ⇐ λ {u} x uA → case only-w₁⊩A uA of λ { refl → w₂ ,
Sww₁w₂ , var w₂⊩B}
    w₁⊩A : a ∈ V w₁
    w₁⊩A with w₁ ≟ w₁
    ... | yes refl = tt
    ... | no n = ⊥-elim (n refl)
    w₁⊩□C' : ∀ {u} → R w₁ u → c ∈ V u
    w₁⊩□C' {u} x with u ≟ w₁
```

```
    ... | yes refl = ⊥-elim (R-Irreflexive F refl x)
    ... | no _ with u ≟ w₂
    ... | yes refl = ¬Rw₁w₃ (R-trans x Rw₂w₃)
    ... | no _ with u ≟ w₃
    ... | yes refl = ¬Rw₁w₃ x
    ... | no _ with R? w₁ u
    ... | yes r = tt
    ... | no nr = nr x
    w₁⊩□C : model {V = V} F , w₁ ⊩ □ var c
    w₁⊩□C = ⊩□ ⇐ λ Rw₁v → var (w₁⊩□C' Rw₁v)
    w₁⊩A∧□C : model {V = V} F , w₁ ⊩ var a ∧ □ var c
    w₁⊩A∧□C = ⊩∧ ⇐ (var w₁⊩A , w₁⊩□C)
```

B.27. OrdinaryVeltmanSemantics/Properties/M_0

```
module OrdinaryVeltmanSemantics.Properties.M₀ where

open import Function.Equivalence using (_⇔_; equivalence; module Equivalence)

open import Agda.Builtin.Nat using (Nat; suc; zero)
open import Agda.Builtin.Unit using (⊤; tt)
open import Agda.Primitive using (Level; lzero; lsuc; _⊔_)
open import Data.Empty using (⊥; ⊥-elim)
open import Data.Product using (Σ; proj₁; proj₂; _×_) renaming (_,_ to
_,_)
open import Data.Sum using (_⊎_; inj₁; inj₂; [_,_])
open import Function using (_∘_; const; case_of_; id)
open import Function.Equality using (_⟨$⟩_)
open import Relation.Binary using (REL; Rel; Transitive)
open import Relation.Nullary using (yes; no; ¬_)
open import Relation.Unary using (Pred; _∈_; _∉_; Decidable; Satisfiable;
_⊆_; _∩_; {_})
open import Relation.Binary using (Irreflexive) renaming (Decidable to
Decidable₂)
open import Relation.Binary.PropositionalEquality using (_≡_; refl;
subst; trans; sym)

open import Formula using (Fm; Var; _↝_; ⊥'; _▷_; var; ⊤'; ¬'_; □_; ◇_;
_∧_; _∨_; car)
open import OrdinaryVeltmanSemantics using (Frame; Model; _,_⊮_; _,_⊩_;
impl; var; rhd; bot; _*⊩_; _*⊮_; Valuation; model; DecidableModel)
open import OrdinaryVeltmanSemantics.Properties using (module Extended;
⊩∧; ⊩□;
  R-Irreflexive; ⊩◇; ⊩→¬⊮; ⊩MP)
open import Base using (_⇒_; _⇐_; Decidable₃)
import Principles as P

private
  variable
    ℓW ℓR ℓS : Level
```

```
M₀-condition : ∀ {W R S} → Frame {ℓW} {ℓR} {ℓS} W R S → Set (ℓW ⊔ ℓR ⊔
ℓS)
M₀-condition {W = W} {R = R} {S = S} F = ∀ {w x y z} → R w x → R x y →
S w y z → (∀ {u} → R z u → R x u)
  where open Frame F

module M₀-soundness
  {W R S V}
  {M : Model {ℓW} {ℓR} {ℓS} W R S V}
  (M,_⊩?_ : DecidableModel M)
  (S? : Decidable₃ S)
  where

  open Model M
  open Frame F
  open Extended M,_⊩?_ S?

  ⊩M₀ : ∀ {w} → M₀-condition (Model.F M) → P.M₀ (M , w ⊩_)
  ⊩M₀ {w} c {A} {B} = ⊩⇝ ⇐ λ A▷B → ⊩▷ ⇐ λ { {x} Rwx x∧ →
    case ⊩∧ ⇒ x∧ of
    λ { (x◇A , x□C) → case ⊩◇ ⇒ x◇A of λ { (y , Rxy , yA) →
      case (⊩▷ ⇒ A▷B) (R-trans Rwx Rxy) yA of
    λ { (z , Swyz , zB) → z , Sw-trans (R-Sw-trans Rwx Rxy) Swyz ,
      ⊩∧ ⇐ (zB , (⊩□ ⇐ (λ {p → (⊩□ ⇒ x□C) (c Rwx Rxy Swyz p)})))}}}}


module M₀-cond
  {W R S}
  {F : Frame {lzero} {lzero} {lzero} W R S}
  (dec : ∀ V → DecidableModel (model {V = V} F))
  (S? : Decidable₃ S)
  where
  open Frame F

  F*⊩M₀ : Set₁
  F*⊩M₀ = P.M₀ (F *⊩_)

  pattern a = 0
  pattern b = 1
  pattern c = 2

  *⊩M₀ : F*⊩M₀ → M₀-condition F
  *⊩M₀ ⊩M₀ {w} {x} {y} {z} Rwx Rxy Swyz =
    case (⊩▷ ⇒ ⊩MP (⊩M₀ Val w) w⊩a▷b) Rwx x⊩◇a∧□c of
    λ { (z' , Swxz , snd) → case proj₁ (⊩∧ ⇒ snd) of
    λ { (var refl) → (λ {Rzu → case (⊩□ ⇒ proj₂ (⊩∧ ⇒ snd)) Rzu of
    λ {(var x₁) → x₁}})} }
    where
    Val : Valuation F
    Val w a = w ≡ y
    Val w b = w ≡ z
    Val w c = R x w
    Val w (suc (suc (suc _))) = ⊥
    M = model {V = Val} F
    open Extended {M = M} (dec Val) S?
    [a] : ∀ {w} → M , w ⊩ var a ⇔ w ≡ y
    [a] = equivalence (λ { (var x) → x}) λ {z → var z}
    w⊩a▷b : M , w ⊩ var a ▷ var b
    w⊩a▷b = ⊩▷ ⇐ λ { Rwy ya → case [a] ⇒ ya of
      λ {refl → z , (Swyz , (var refl))}}
    x⊩◊a∧□c : M , x ⊩ ◊ var a ∧ □ var c
    x⊩◊a∧□c = ⊩∧ ⇐ (⊩◊ ⇐ (y , Rxy , var refl) , ⊩□ ⇐
      λ {x₁ → var x₁})
```

B.28. OrdinaryVeltmanSemantics/Properties/P$_0$

```
module OrdinaryVeltmanSemantics.Properties.P₀ where

open import Function.Equivalence using (_⇔_; equivalence; module Equivalence)

open import Agda.Builtin.Nat using (Nat; suc; zero)
open import Agda.Builtin.Unit using (⊤; tt)
open import Agda.Primitive using (Level; lzero; lsuc; _⊔_)
open import Data.Empty using (⊥; ⊥-elim)
open import Data.Product using (Σ; proj₁; proj₂; _×_) renaming (_,_ to
_,_)
open import Data.Sum using (_⊎_; inj₁; inj₂; [_,_])
open import Function using (_∘_; const; case_of_; id)
open import Function.Equality using (_⟨$⟩_)
open import Relation.Binary using (REL; Rel; Transitive)
open import Relation.Nullary using (yes; no; ¬_)
open import Relation.Unary using (Pred; _∈_; _∉_; Decidable; Satisfiable;
_⊆_; _∩_; {_})
open import Relation.Binary using (Irreflexive) renaming (Decidable to
Decidable₂)
open import Relation.Binary.PropositionalEquality using (_≡_; refl;
subst; trans; sym)

open import Formula using (Fm; Var; _↝_; ⊥'; _▷_; var; ⊤'; ¬'_; □_; ◊_;
_∧_; _∨_; car)
open import OrdinaryVeltmanSemantics using (Frame; Model; _,_⊮_; _,_⊩_;
impl;
  var; rhd; bot; _*⊩_; _*⊮_; Valuation; model; DecidableModel)
open import OrdinaryVeltmanSemantics.Properties using (module Extended;
⊩∧; ⊩□;
  R-Irreflexive; ⊩◊; ⊩→¬⊮; ⊩MP)
open import Base using (_⇒_; _⇐_; Decidable₃)
import Principles as P

private
  variable
    ℓW ℓR ℓS : Level
```

```
P₀-condition : ∀ {W R S} → Frame {ℓW} {ℓR} {ℓS} W R S → Set (ℓW ⊔ ℓR ⊔
ℓS)
P₀-condition {W = W} {R = R} {S = S} F = ∀ {w x y z u} → R w x → R x y
→ S w y z → R z u → S x y u
  where open Frame F

module P₀-soundness
  {W R S V}
  {M : Model {ℓW} {ℓR} {ℓS} W R S V}
  (M,_⊩?_ : DecidableModel M)
  (S? : Decidable₃ S)
  where

  open Model M
  open Frame F
  open Extended M,_⊩?_ S?

  ⊩P₀ : ∀ {w} → P₀-condition (Model.F M) → P.P₀ (M , w ⊩_)
  ⊩P₀ {w} c {A} {B} = ⊩↝ ⇐ λ A▷◊B → ⊩□ ⇐ λ { {x} Rwx → ⊩▷ ⇐ λ { {y} Rxy
y⊩A
    → case (⊩▷ ⇒ A▷◊B) (R-trans Rwx Rxy) y⊩A of λ { (z , Swyz , z⊩◊B)
→ case ⊩◊ ⇒ z⊩◊B of
    λ { (u , Rzu , uB) → u , (c Rwx Rxy Swyz Rzu) , uB}}}}


module P₀-cond
  {W R S}
  {F : Frame {lzero} {lzero} {lzero} W R S}
  (dec : ∀ V → DecidableModel (model {V = V} F))
  (S? : Decidable₃ S)
  where
  open Frame F

  F*⊩P₀ : Set₁
  F*⊩P₀ = P.P₀ (F *⊩_)

  pattern a = 0
  pattern b = 1

  *⊩P₀ : F*⊩P₀ → P₀-condition F
  *⊩P₀ ⊩P₀ {w} {x} {y} {z} {u} Rwx Rxy Swyz Rzu = case (⊩▷ ⇒ x⊩a▷b) Rxy
([a] ⇐ refl) of
    λ { (z' , Sxyz , z⊩b) → case [b] ⇒ z⊩b of λ {refl → Sxyz}}
    where
    Val : Valuation F
    Val w a = w ≡ y
    Val w b = w ≡ u
    Val w (suc (suc _)) = ⊥
    M = model {V = Val} F
    open Extended {M = M} (dec Val) S?
    [a] : ∀ {w} → M , w ⊩ var a ⇔ w ≡ y
    [a] = equivalence (λ { (var x) → x}) λ {z → var z}
    [b] : ∀ {w} → M , w ⊩ var b ⇔ w ≡ u
    [b] = equivalence (λ { (var x) → x}) λ {z → var z}
    w⊩a▷◇b : M , w ⊩ var a ▷ ◇ var b
    w⊩a▷◇b = ⊩▷ ⇐ λ { {x} Rwx x⊩a → case [a] ⇒ x⊩a of
      λ { refl → z , Swyz , ⊩◇ ⇐ (u , Rzu , [b] ⇐ refl)} }
    w⊩□a▷b : M , w ⊩ □ (var a ▷ var b)
    w⊩□a▷b = ⊩MP (⊩P₀ Val w) w⊩a▷◇b
    x⊩a▷b : M , x ⊩ var a ▷ var b
    x⊩a▷b = (⊩□ ⇒ w⊩□a▷b) Rwx
```

B.29. OrdinaryVeltmanSemantics/Properties/R

```
module OrdinaryVeltmanSemantics.Properties.R where

open import Function.Equivalence using (_⇔_; equivalence; module Equivalence)

open import Agda.Builtin.Nat using (Nat; suc; zero)
open import Agda.Builtin.Unit using (⊤; tt)
open import Agda.Primitive using (Level; lzero; lsuc; _⊔_)
open import Data.Empty using (⊥; ⊥-elim)
open import Data.Product using (Σ; proj₁; proj₂; _×_) renaming (_,_ to
_,_)
open import Data.Sum using (_⊎_; inj₁; inj₂; [_,_])
open import Function using (_∘_; const; case_of_; id)
open import Function.Equality using (_⟨$⟩_)
open import Relation.Binary using (REL; Rel; Transitive)
open import Relation.Nullary using (yes; no; ¬_)
open import Relation.Unary using (Pred; _∈_; _∉_; Decidable; Satisfiable;
_⊆_; _∩_; {_})
open import Relation.Binary using (Irreflexive) renaming (Decidable to
Decidable₂)
open import Relation.Binary.PropositionalEquality using (_≡_; refl;
subst; trans; sym)

open import Formula using (Fm; Var; _↝_; ⊥'; _▷_; var; ⊤'; ¬'_; □_; ◇_;
_∧_; _∨_; car)
open import OrdinaryVeltmanSemantics using (Frame; Model; _,_⊮_; _,_⊩_;
impl;
  var; rhd; bot; _*⊩_; _*⊮_; Valuation; model; DecidableModel)
open import OrdinaryVeltmanSemantics.Properties using (module Extended;
⊩∧; ⊩□;
  R-Irreflexive; ⊩◇; ⊩→¬⊮; ⊩MP)
open import Base using (_⇒_; _⇐_; Decidable₃)
import Principles as P

private
  variable
    ℓW ℓR ℓS : Level
```

```
R-condition : ∀ {W R S} → Frame {ℓW} {ℓR} {ℓS} W R S → Set (ℓW ⊔ ℓR ⊔
ℓS)
R-condition {W = W} {R = R} {S = S} F = ∀ {w x y z u} → R w x → R x y
→ S w y z → R z u → S x y u
  where open Frame F

module R-soundness
  {W R S V}
  {M : Model {ℓW} {ℓR} {ℓS} W R S V}
  (M,_⊩?_ : DecidableModel M)
  (S? : Decidable₃ S)
  where

  open Model M
  open Frame F
  open Extended M,_⊩?_ S?

  ⊩R : ∀ {w} → R-condition (Model.F M) → P.R (M , w ⊩_)
  ⊩R {w} c {A} {B} = ⊩⇝ ⇐ λ A▷B → ⊩▷ ⇐ λ { {x} Rwx x⊩ → case ⊩◁ ⇒ x⊩ of
    λ { (y , Rxy , yA , snd) → case (⊩▷ ⇒ A▷B) (R-trans Rwx Rxy) yA of
      λ { (z , Swyz , z⊩B) → z , Sw-trans (R-Sw-trans Rwx Rxy) Swyz ,
⊩∧ ⇐ (z⊩B , ⊩□ ⇐
      λ { {u} Rzu → snd (c Rwx Rxy Swyz Rzu)})}}}


module R-cond
  {W R S}
  {F : Frame {lzero} {lzero} {lzero} W R S}
  (dec : ∀ V → DecidableModel (model {V = V} F))
  (S? : Decidable₃ S)
  where
  open Frame F

  F*⊩R : Set₁
  F*⊩R = P.R (F *⊩_)

  pattern a = 0
  pattern b = 1
  pattern c = 2

  *⊩R : F*⊩R → R-condition F
  *⊩R ⊩R {w} {x} {y} {z} {u} Rwx Rxy Swyz Rzu = case (⊩▷ ⇒ (⊩MP (⊩R Val
w) w⊩a▷b)) Rwx x⊩¬a▷¬c of
    λ { (z' , Swxz , snd) → case ⊩∧ ⇒ snd of
     λ { (fst , snd) → case [b] ⇒ fst of λ {refl → [c] ⇒ (⊩□ ⇒ snd)
Rzu}}}
    where
    Val : Valuation F
    Val w a = w ≡ y
    Val w b = w ≡ z
    Val w c = S x y w
    Val w (suc (suc (suc _))) = ⊥
    M = model {V = Val} F
    open Extended {M = M} (dec Val) S?
    [a] : ∀ {w} → M , w ⊩ var a ⇔ w ≡ y
    [a] = equivalence (λ { (var x) → x}) λ {z → var z}
    [b] : ∀ {w} → M , w ⊩ var b ⇔ w ≡ z
    [b] = equivalence (λ { (var x) → x}) λ {z → var z}
    [c] : ∀ {w} → M , w ⊩ var c ⇔ S x y w
    [c] = equivalence (λ { (var x) → x}) λ {z → var z}
    w⊩a▷b : M , w ⊩ var a  ▷ var b
    w⊩a▷b = ⊩▷ ⇐ λ { Rwy y → case [a] ⇒ y of λ {refl → z , Swyz , var
refl}}
    x⊩¬a▷¬c : M , x ⊩ ¬' (var a ▷ ¬' (var c))
    x⊩¬a▷¬c = ⊩◁ ⇐ (y , Rxy , var refl , λ { Sxyv → var Sxyv})
```

B.30. OrdinaryVeltmanSemantics/Properties

```
module OrdinaryVeltmanSemantics.Properties where

open import Function.Equivalence using (_⇔_; equivalence; module Equivalence)

open import Agda.Builtin.Nat using (Nat; suc; zero)
open import Agda.Builtin.Unit using (⊤; tt)
open import Agda.Primitive using (Level; lzero; lsuc; _⊔_)
open import Data.Empty using (⊥; ⊥-elim)
open import Data.Product renaming (_,_ to _,_)
open import Data.Sum using (_⊎_; inj₁; inj₂; [_,_])
open import Function using (_∘_; const; case_of_; id)
open import Function.Equality using (_⟨$⟩_)
open import Relation.Binary using (REL; Rel; Transitive)
open import Relation.Nullary using (yes; no; ¬_)
open import Relation.Unary using (Pred; _∈_; _∉_; Decidable)
open import Relation.Binary using (Irreflexive) renaming (Decidable to
Decidable₂)
open import Relation.Binary.PropositionalEquality using (_≡_; refl)

open import Formula using (Fm; Var; _↝_; ⊥'; _▷_; var; ⊤'; ¬'_; □_; ◇_;
_∧_; _∨_; car)
open import OrdinaryVeltmanSemantics
open import Base

private
  variable
    ℓW ℓR ℓS : Level
    W : Set ℓW
    R : Rel W ℓR
    S : Rel₃ W ℓS
    V : REL W Var lzero

module _
  {M : Model W R S V} where
  open Model M
  open Frame F

  ⊩⊥ : ∀ {w} → ¬ (M , w ⊩ ⊥')
  ⊩⊥ = λ ()

  ⊮→¬⊩ : ∀ {w A} → M , w ⊮ A → ¬ (M , w ⊩ A)
  ⊮→¬⊩ (var x) (var x₂) = x x₂
  ⊮→¬⊩ (impl x y) (impl (inj₁ z)) = ⊮→¬⊩ z x
  ⊮→¬⊩ (impl x y) (impl (inj₂ z)) = ⊮→¬⊩ y z
  ⊮→¬⊩ (rhd (u' , wRu' , u'⊩A , snd)) (rhd z) with z wRu'
  ... | inj₁ u'⊮A with snd u'
  ... | inj₁ x = x (Sw-refl wRu')
  ... | inj₂ y = ⊮→¬⊩ u'⊮A u'⊩A
  ⊮→¬⊩ (rhd (u' , wRu' , mm , snd)) (rhd z) | inj₂ (v , u'Sv , vB) with
snd v
  ... | inj₁ ¬u'Sv = ¬u'Sv u'Sv
  ... | inj₂ r = ⊮→¬⊩ r vB

  ⊩→¬⊮ : ∀ {w A} → M , w ⊩ A → ¬ (M , w ⊮ A)
  ⊩→¬⊮ x y = ⊮→¬⊩ y x

  ⊩MP : ∀ {w A B} → M , w ⊩ A ⇝ B → M , w ⊩ A → M , w ⊩ B
  ⊩MP (impl (inj₁ x)) y = ⊥-elim (⊩→¬⊮ y x)
  ⊩MP (impl (inj₂ x)) y = x

  ⊩¬ : ∀ {w A} → (M , w ⊩ ¬' A) ⇔ (M , w ⊮ A)
  ⊩¬ {w} {A} = equivalence ⇨ ⇦
    where ⇨ : M , w ⊩ ¬' A → M , w ⊮ A
          ⇨ (impl (inj₁ x)) = x
          ⇦ : M , w ⊮ A → M , w ⊩ ¬' A
          ⇦ (var x) = impl (inj₁ (var x))
          ⇦ (impl x x₁) = impl (inj₁ (impl x x₁))
          ⇦ (rhd x) = impl (inj₁ (rhd x))
          ⇦ bot = impl (inj₁ bot)

  ⊩⊤ : ∀ {w} → M , w ⊩ ⊤'
  ⊩⊤ = impl (inj₁ bot)

  ⊮⊤ : ∀ {w} → ¬ (M , w ⊮ ⊤')
  ⊮⊤ (impl x x₁) = ⊩→¬⊮ x x₁

  ⊮¬ : ∀ {w A} → M , w ⊮ ¬' A ⇔ M , w ⊩ A
  ⊮¬ {w} {A} = equivalence ⇨ ⇦
    where
      ⇨ : M , w ⊮ ¬' A → M , w ⊩ A
      ⇨ (impl x x₁) = x
      ⇦ : M , w ⊩ A → M , w ⊮ ¬' A
      ⇦ x = impl x bot

  ⊩¬¬ : ∀ {w A} → M , w ⊩ ¬' ¬' A ⇔ M , w ⊩ A
  ⊩¬¬ {w} {A} = equivalence ⇨ ⇦
    where
    ⇨ : M , w ⊩ ¬' ¬' A → M , w ⊩ A
    ⇨ (impl (inj₁ (impl x x₁))) = x
    ⇦ : M , w ⊩ A → M , w ⊩ ¬' ¬' A
    ⇦ x = impl (inj₁ (impl x bot))

  ⊮¬¬ : ∀ {w A} → M , w ⊮ ¬' ¬' A ⇔ M , w ⊮ A
  ⊮¬¬ {w} {A} = equivalence ⇨ ⇦
    where
    ⇨ : M , w ⊮ ¬' ¬' A → M , w ⊮ A
    ⇨ x = ⊩¬ ⇒ (⊮¬ ⇒ x)
    ⇦ : M , w ⊮ A → M , w ⊮ ¬' ¬' A
    ⇦ x = ⊮¬ ⇐ (⊩¬ ⇐ x)

  ⊩∧ : ∀ {w A B} → M , w ⊩ A ∧ B ⇔ (M , w ⊩ A × M , w ⊩ B)
  ⊩∧ {w} {A} {B} = equivalence ⇨ ⇦
    where
    ⇨ : M , w ⊩ A ∧ B → M , w ⊩ A × M , w ⊩ B
    ⇨ (impl (inj₁ (impl x (impl x₁ y)))) = x , x₁
    ⇦ : M , w ⊩ A × M , w ⊩ B → M , w ⊩ A ∧ B
    ⇦ (fst , snd) = impl (inj₁ (impl fst (impl snd bot)))

  ⊮∧ : ∀ {w A B} → M , w ⊮ A ∧ B ⇔ (M , w ⊮ A ⊎ M , w ⊮ B)
  ⊮∧ {w} {A} {B} = equivalence ⇨ ⇦
    where
    ⇨ : M , w ⊮ A ∧ B → (M , w ⊮ A ⊎ M , w ⊮ B)
    ⇨ (impl (impl (inj₁ x)) x₁) = inj₁ x
    ⇨ (impl (impl (inj₂ (impl (inj₁ x)))) x₁) = inj₂ x
    ⇦ : (M , w ⊮ A ⊎ M , w ⊮ B) → M , w ⊮ A ∧ B
    ⇦ (inj₁ x) = impl (impl (inj₁ x)) bot
    ⇦ (inj₂ y) = impl (impl (inj₂ (⊩¬ ⇐ y))) bot

  ⊩∨ : ∀ {w A B} → M , w ⊩ A ∨ B ⇔ (M , w ⊩ A ⊎ M , w ⊩ B)
  ⊩∨ {w} {A} {B} = equivalence ⇨ ⇦
    where
    ⇨ : M , w ⊩ A ∨ B → (M , w ⊩ A ⊎ M , w ⊩ B)
    ⇨ (impl (inj₁ (impl x x₁))) = inj₁ x
    ⇨ (impl (inj₂ y)) = inj₂ y
    ⇦ : (M , w ⊩ A ⊎ M , w ⊩ B) → M , w ⊩ A ∨ B
    ⇦ (inj₁ x) = impl (inj₁ (⊮¬ ⇐ x))
    ⇦ (inj₂ y) = impl (inj₂ y)

  ⊩□ : ∀ {w A} → M , w ⊩ □ A ⇔ (∀ {v} → R w v → M , v ⊩ A)
  ⊩□ {w} {A} = equivalence ⇨ ⇦
    where
    ⇨ : M , w ⊩ □ A → (∀ {v} → R w v → M , v ⊩ A)
    ⇨ (rhd f) wRv with f wRv
    ... | inj₁ k = ⊮¬ ⇒ k
    ⇦ : (∀ {v} → R w v → M , v ⊩ A) → M , w ⊩ □ A
    ⇦ x = rhd λ wRu → inj₁ (⊮¬ ⇐ (x wRu))
  ⊮□ : ∀ {w A} → M , w ⊮ □ A ⇔ (∃[ u ] (R w u × M , u ⊮ A))
  ⊮□ {w} {A} = equivalence ⇨ ⇦
    where
    ⇨ : M , w ⊮ □ A → (Σ (W) λ u → R w u × M , u ⊮ A)
    ⇨ (rhd (u , wRu , u⊩¬A , snd)) = u , wRu , ⊩¬ ⇒ u⊩¬A
    ⇦ : (Σ (W) λ u → R w u × M , u ⊮ A) → M , w ⊮ □ A
    ⇦ (u , wRu , u⊮A) = rhd (u , (wRu , ⊩¬ ⇐ u⊮A , λ v → inj₂ bot))

  ⊩◊ : ∀ {w A} → M , w ⊩ ◊ A ⇔ (Σ (W) λ u → R w u × M , u ⊩ A)
  ⊩◊ {w} {A} = equivalence ⇨ ⇦
    where
    ⇨ : M , w ⊩ ◊ A → Σ (W) λ u → R w u × M , u ⊩ A
    ⇨ (impl (inj₁ (rhd (u , m , u⊩¬¬A , snd)))) = u , m , ⊩¬¬ ⇒ u⊩¬¬A
    ⇦ : (Σ (W) λ u → R w u × M , u ⊩ A) → M , w ⊩ ◊ A
    ⇦ (u , wRu , snd) = impl (inj₁ (rhd (u , (wRu , ⊩¬¬ ⇐ snd , λ _ →
inj₂ bot))))

  ⊮◊ : ∀ {w A} → M , w ⊮ ◊ A ⇔ (∀ {u} → R w u → M , u ⊮ A)
  ⊮◊ {w} {A} = equivalence ⇨ ⇦
    where
    ⇨ : M , w ⊮ ◊ A → (∀ {u} → R w u → M , u ⊮ A)
    ⇨ (impl (rhd x) y) wRu with x wRu
    ... | inj₁ u⊮¬¬A = ⊮¬¬ ⇒ u⊮¬¬A
    ⇦ : (∀ {u} → R w u → M , u ⊮ A) → M , w ⊮ ◊ A
    ⇦ x = impl (rhd (λ wRu → inj₁ (⊮¬¬ ⇐ (x wRu)))) bot

  ⊩4' : ∀ {w A} → M , w ⊩ □ A → M , w ⊩ □ □ A
  ⊩4' (rhd x) = ⊩□ ⇐ λ {u} wRu → rhd (λ {v} uRv →
    case x {v} (R-trans wRu uRv) of λ { (inj₁ x) → inj₁ x})

  ⊩⤳⇨ : ∀ {w A B} → M , w ⊩ A ⤳ B → M , w ⊩ A → M , w ⊩ B
  ⊩⤳⇨ (impl (inj₁ x)) y = ⊥-elim (⊮→¬⊩ x y)
  ⊩⤳⇨ (impl (inj₂ b)) y = b

  ⊩▷⇨ : ∀ {w A B} → M , w ⊩ A ▷ B → (∀ {u} → R w u → M , u ⊩ A → Σ (W)
λ v → (S) w u v × M , v ⊩ B)
  ⊩▷⇨ (rhd x) {u} wRu uA with x wRu
 ⊩▷⇨ {M} (rhd x) {u} wRu uA | inj₁ x₁ = u , (Sw-refl wRu , ⊥-elim (⊩→¬⊮
uA x₁))
  ⊩▷⇨ (rhd x) {u} wRu uA | inj₂ (fst , fst₁ , snd) = fst , (fst₁ , snd)

R-Irreflexive : Frame W R S → Irreflexive _≡_ R
R-Irreflexive {R = R} F {x} refl Rxx = R-noetherian (infiniteRefl Rxx)
  where open Frame F

-- This module contains the properties that can be proved with the
assumption
-- that we have a procedure to decide M , w ⊩ A
module Extended
  {M : Model {ℓW} {ℓR} {ℓS} W R S V}
  (M,_⊩?_ : DecidableModel M)
  (∈S? : Decidable₃ (S))
  where
  open Frame (Model.F M)

  ⊩⤳ : ∀ {w A B} → M , w ⊩ A ⤳ B ⇔ (M , w ⊩ A → M , w ⊩ B)
  ⊩⤳ {w} {A} {B} = equivalence ⊩⤳⇒ ⇐
    where
    ⇐ : (M , w ⊩ A → M , w ⊩ B) → M , w ⊩ A ⤳ B
    ⇐ x with M, w ⊩? A
    ... | inj₁ z = impl (inj₂ (x z))
    ... | inj₂ y = impl (inj₁ y)

  ⊩▷ : ∀ {w A B} → M , w ⊩ A ▷ B ⇔
    (∀ {u} → R w u → M , u ⊩ A → Σ (W) λ v → (S) w u v × M , v ⊩ B)
  ⊩▷ {w} {A} {B} = equivalence ⊩▷⇒ ⇐
    where
    ⇐ : (∀ {u} → R w u → M , u ⊩ A → Σ W λ v → S w u v × M , v ⊩ B) → M
, w ⊩ A ▷ B
    ⇐ x = rhd (λ {u} wRu → [ (λ x₁ → inj₂ (x wRu x₁)) , inj₁ ] (M, u
⊩? A))

  ⊩4 : ∀ {w A} → M , w ⊩ □ A ⤳ □ □ A
  ⊩4 = ⊩⤳ ⇐ ⊩4'

  ⊩⇔¬⊮ : ∀ {w A} → M , w ⊩ A ⇔ (¬ M , w ⊮ A)
  ⊩⇔¬⊮ {w} {A} = equivalence ⊩→¬⊮ ⇐
    where
    ⇐ : (M , w ⊮ A → ⊥) → M , w ⊩ A
    ⇐ x = [ id , (λ y → ⊥-elim (x y)) ] (M, w ⊩? A)

  ⊮⇔¬⊩ : ∀ {w A} → M , w ⊮ A ⇔ (¬ M , w ⊩ A)
  ⊮⇔¬⊩ {w} {A} = equivalence ⊮→¬⊩ ⇐
    where
    ⇐ : ¬ M , w ⊩ A → M , w ⊮ A
    ⇐ x = [ (λ y → ⊥-elim (x y)) , id ] (M, w ⊩? A)

  ⊮▷ : ∀ {w A B} → M , w ⊮ A ▷ B ⇔ Σ W λ x → R w x × M , x ⊩ A
    × (∀ {z} → S w x z → M , z ⊮ B)
  ⊮▷ {w} {A} {B} = equivalence ⇒ ⇐
    where
    ⇒ : M , w ⊮ A ▷ B → Σ W λ x → R w x × M , x ⊩ A
      × (∀ {z} → S w x z → M , z ⊮ B)
     ⇒ (rhd (y , fst₁ , fst₂ , snd)) = y , fst₁ , fst₂ , λ { {z} Sx →
case snd z of
      λ { (inj₁ x) → ⊥-elim (x Sx) ; (inj₂ y) → y}}
    ⇐ : (Σ W λ x → R w x × M , x ⊩ A × (∀ {z} → S w x z → M , z ⊮ B)) →
M , w ⊮ A ▷ B
    ⇐ (y , Rwy , yA , snd) = rhd (y , Rwy , yA , (λ z → case ∈S? w y z of
      λ { (yes p) → inj₂ (snd p); (no p) → inj₁ p}))
  ⊩◁ : ∀ {w A B} → M , w ⊩ ¬' (A ▷ ¬' B) ⇔ Σ W λ x → R w x × M , x ⊩ A
    × ∀ {z} → S w x z → M , z ⊩ B
  ⊩◁ {w} {A} {B} = equivalence ⇨ ⇦
    where
    ⇨ : M , w ⊩ ¬' (A ▷ ¬' B) → Σ W λ x → R w x × M , x ⊩ A
      × ∀ {z} → S w x z → M , z ⊩ B
    ⇨ x = case ⊮▷ ⇒ (⊩¬ ⇒ x) of λ { (fst , fst₁ , fst₂ , snd) → fst ,
fst₁ , fst₂ ,
      λ {x → ⊮¬ ⇒ (snd x)}}
    ⇦ : (Σ W λ x → R w x × M , x ⊩ A × ∀ {z} → S w x z → M , z ⊩ B) → M
, w ⊩ ¬' (A ▷ ¬' B)
    ⇦ (y , fst₁ , fst₂ , snd) = ⊩¬ ⇐ (⊮▷ ⇐ (y , fst₁ , fst₂ , λ {x → ⊮¬
⇐ snd x}))

  ⊩K : ∀ {w A B} → M , w ⊩ □ (A ↝ B) ↝ □ A ↝ □ B
  ⊩K {w} {A} {B} = ⊩↝ ⇐ λ x → ⊩↝ ⇐ λ y → ⊩□ ⇐
    λ {u} wRu → ⊩MP ((⊩□ ⇒ x) wRu) ((⊩□ ⇒ y) wRu)

  ⊩J1 : ∀ {w A B} → M , w ⊩ □ (A ↝ B) ↝ A ▷ B
  ⊩J1 {w} {A} {B} = ⊩↝ ⇐ λ x → rhd (λ {u} z →
    [ (λ uA → inj₂ (u , Sw-refl z ,
    ⊩MP ((⊩□ ⇒ x) z) uA)) , inj₁ ] (M, u ⊩? A))

  ⊩J2 : ∀ {w A B C} → M , w ⊩ A ▷ B ∧ B ▷ C ↝ A ▷ C
  ⊩J2 {w} {A} {B} {C} = ⊩↝ ⇐ λ x → ⊩▷ ⇐ λ {u} wRu uA → case ⊩∧ ⇒ x of
    λ { (wA▷B , snd) → case (⊩▷ ⇒ wA▷B) wRu uA of
    λ { (v , Swuv , snd') → case (⊩▷ ⇒ snd)
    (proj₂ (Sw⊆R[w]² Swuv)) snd' of
    λ { (e , fst , snd) → e , Sw-trans Swuv fst ,  snd}}}

  ⊩J3 : ∀ {w A B C} → M , w ⊩ A ▷ C ∧ B ▷ C ↝ (A ∨ B) ▷ C
  ⊩J3 {w} {A} {B} {C} = ⊩↝ ⇐ λ x → ⊩▷ ⇐ λ Rwu uA∨B → case ⊩∧ ⇒ x of
    λ {(a , b) → case ⊩∨ ⇒ uA∨B of
    λ { (inj₁ uA) → (⊩▷ ⇒ a) Rwu uA ;
    (inj₂ uB) → (⊩▷ ⇒ b) Rwu uB} }

  ⊩J4 : ∀ {w A B} → M , w ⊩ A ▷ B ↝ ◇ A ↝ ◇ B
  ⊩J4 = ⊩↝ ⇐ λ x → ⊩↝ ⇐ λ y → ⊩◇ ⇐ (case ⊩◇ ⇒ y of λ { (u , wRu , snd)
    → case (⊩▷ ⇒ x) wRu snd of λ { (v , fst₁ , snd)
    → v , (proj₂ (Sw⊆R[w]² fst₁)) , snd}})

  ⊩J5 : ∀ {w A} → M , w ⊩ ◇ A ▷ A
  ⊩J5 = ⊩▷ ⇐ λ {u} wRu u⊩◇A → case ⊩◇ ⇒ u⊩◇A of λ { (v , uRv , snd) →
    v , R-Sw-trans wRu uRv , snd}


module Extended2
  {M : Model {ℓW} {ℓR} {ℓS} W R S V}
  (M,_⊩?_ : DecidableModel M)
  (∈S? : Decidable₃ S)
  where
  open Frame (Model.F M)
  open Extended M,_⊩?_ ∈S?

   L-chain : ∀ {w u A} → R w u → M , u ⊮ A → M , w ⊩ □ (□ A ↝ A) →
InfiniteChain R w
  InfiniteChain.b (L-chain {w} {u} Rwu uA uF) = u
  InfiniteChain.a<b (L-chain {w} {u} Rwu uA uF) = Rwu
  InfiniteChain.tail (L-chain {w} {u} Rwu u⊮A w⊩□⟨□A↝A⟩)
     with (⊩□ ⇒ w⊩□⟨□A↝A⟩) Rwu
  ... | impl (inj₂ u⊩A) = ⊥-elim (⊩→¬⊮ u⊩A u⊮A)
  ... | impl (inj₁ x⊮□A) with ⊮□ ⇒ x⊮□A
  ... | (v , Ruv , v⊩A) = L-chain Ruv v⊩A ((⊩□ ⇒ ⊩4' w⊩□⟨□A↝A⟩) Rwu)

  ⊩L : ∀ {w A} → M , w ⊩ □ (□ A ↝ A) ↝ □ A
  ⊩L {w} {A} = ⊩↝ ⇐ λ w⊩□⟨□A→A⟩ → ⊩□ ⇐ λ {u} Rwu → ⊩⇔¬⊮ ⇐
    λ {u⊮A → R-noetherian (L-chain Rwu u⊮A w⊩□⟨□A→A⟩)}
```

B.31. OrdinaryVeltmanSemantics

```
module _ where

open import Relation.Binary using (REL; Rel; Transitive; Reflexive)
open import Relation.Unary using (Pred; _∈_; _∉_; Decidable)
open import Data.List.Relation.Unary.All using (All; []; _∷_)
open import Relation.Nullary using (¬_)
open import Data.List using (map; List; []; _∷_)
open import Agda.Primitive using (Level; lzero; lsuc; _⊔_)
open import Agda.Builtin.Nat using (Nat; suc; _+_)
open import Data.Product
open import Data.Sum using (_⊎_; inj₁; inj₂)
open import Function using (_∘_)
open import Data.Empty using (⊥; ⊥-elim)

open import Formula using (Fm; Var; _↝_; ⊥'; _▷_; var; ¬'_)
open import Base using (Noetherian; Rel₃)
open import OrdinaryFrame using (Frame) public

private
  variable
    ℓW ℓR ℓS : Level
    W : Set ℓW
    R : Rel W ℓR
    S : Rel₃ W ℓS
    V : REL W Var lzero

Valuation : Frame {ℓW} {ℓR} {ℓS} W R S → Set (lsuc lzero ⊔ ℓW)
Valuation {W = W} F = REL W Var lzero

record Model (W : Set ℓW) (R : Rel W ℓR) (S : Rel₃ W ℓS) (V : REL W
Var lzero)
  : Set (ℓW ⊔ ℓR ⊔ ℓS) where
  constructor model
  field
    F : Frame {ℓW} {ℓR} {ℓS} W R S

-- {-# NO_POSITIVITY_CHECK #-}
-- data _,_⊩'_ (M : Model) (w : MW M) : Fm → Set where
--   var : ∀ {a : Var} → a ∈ (Model.V M w) → M , w ⊩' var a
-- impl : ∀ {A B} → ((Nat → (M , w ⊩' A)) → (M , w ⊩' B)) → M , w ⊩'
(A ↝ B)
--   rhd : ∀ {A B} →
--     (∀ {u} → (MR M) w u → M , u ⊩' A → (Σ (MW M) λ v → (MS M) w u v
× (M , v ⊩' B)))
--     → M , w ⊩' A ▷ B

infix 5 _,_⊩_
data _,_⊩_ (M : Model {ℓW} {ℓR} {ℓS} W R S V) (w : W)
  : Fm → Set (ℓW ⊔ ℓR ⊔ ℓS)

infix 5 _,_⊮_
data _,_⊮_ (M : Model {ℓW} {ℓR} {ℓS} W R S V) (w : W)
  : Fm → Set (ℓW ⊔ ℓR ⊔ ℓS)

data _,_⊩_ {W = W} {R = R} {S = S} {V = V} M w where
  var : {a : Var} → a ∈ V w → M , w ⊩ var a
  impl : ∀ {A B} → M , w ⊮ A ⊎ M , w ⊩ B → M , w ⊩ A ↝ B
  rhd : ∀ {A B} →
    (∀ {u} → R w u → M , u ⊮ A ⊎ (∃[ v ] (S w u v × M , v ⊩ B)))
    → M , w ⊩ A ▷ B

data _,_⊮_ {W = W} {R = R} {S = S} {V = V} M w where
  var : {a : Var} → a ∉ V w → M , w ⊮ var a
  impl : {A B : Fm} → M , w ⊩ A → M , w ⊮ B → M , w ⊮ A ↝ B
  rhd : {A B : Fm} →
    ∃[ u ] (R w u × M , u ⊩ A × ∀ v → (¬ S w u v) ⊎ M , v ⊮ B)
    → M , w ⊮ A ▷ B
  bot : M , w ⊮ ⊥'

DecidableModel : Model {ℓW} {ℓR} {ℓS} W R S V → Set (ℓW ⊔ ℓR ⊔ ℓS)
DecidableModel M = ∀ w A → M , w ⊩ A ⊎ M , w ⊮ A

-- Frame validity
infix 5 _*⊩_
_*⊩_ : Frame {ℓW} {ℓR} {ℓS} W R S → Fm → Set (lsuc lzero ⊔ ℓW ⊔ ℓR ⊔
ℓS)
F *⊩ A = ∀ val w → model {V = val} F , w ⊩ A

-- infix 5 _*⊮_
_*⊮_ : Frame {ℓW} {ℓR} {ℓS} W R S → Fm → Set (lsuc lzero ⊔ ℓW ⊔ ℓR ⊔
ℓS)
_*⊮_ {W = W} F A = Σ (Valuation F × W) λ { (val , w) → model {V = val}
F , w ⊮ A}
```

```
_,_⊩*_ : Model {ℓW} {ℓR} {ℓS} W R S V → W → List Fm → Set (ℓW ⊔ ℓR ⊔
ℓS)
M , w ⊩* Π = All (M , w ⊩_) Π
```

B.32. Principles

```
module _ where

open import Agda.Builtin.Nat using (Nat; zero; suc; _+_)
open import Data.Fin using (Fin; zero; suc; fromℕ<)
open import Data.Nat.Base using (_≤_; _<_; s≤s)
open import Data.Nat.Properties using (≤-step; ≤-pred; ≤-reflexive)
open import Data.Product using (_×_; _,_)
open import Relation.Binary.PropositionalEquality using (_≡_; refl)
open import Relation.Unary using (Pred)

open import Formula

Principle : ∀ ℓ → Set _
Principle ℓ = (P : Pred Fm ℓ) → Set ℓ

M : ∀ {ℓ} → Principle ℓ
M P = ∀ {A B C} → P (A ▷ B ⊸ (A ∧ □ C) ▷ (B ∧ □ C))

P : ∀ {ℓ} → Principle ℓ
P Pr = ∀ {A B} → Pr (A ▷ B ⊸ □ (A ▷ B))

M₀ : ∀ {ℓ} → Principle ℓ
M₀ P = ∀ {A B C} → P (A ▷ B ⊸ (◊ A ∧ □ C) ▷ (B ∧ □ C))

W : ∀ {ℓ} → Principle ℓ
W P = ∀ {A B} → P (A ▷ B ⊸ A ▷ B ∧ □ (¬' A))

W⋆ : ∀ {ℓ} → Principle ℓ
W⋆ P = ∀ A B → P (A ▷ B ⊸ A ▷ B ∧ □ (¬' A))

P₀-Fm : ∀ A B → Fm
P₀-Fm A B = A ▷ (◊ B) ⊸ □ (A ▷ B)

P₀ : ∀ {ℓ} → Principle ℓ
P₀ P = ∀ {A B} → P (P₀-Fm A B)

R : ∀ {ℓ} → Principle ℓ
R P = ∀ {A B C} → P (A ▷ B ⊸ (¬' (A ▷ ¬' C) ▷ (B ∧ □ C)))

R₁ : ∀ {ℓ} → Principle ℓ
R₁ P = ∀ {A B C D E} → P (
  A ▷ B ⊸ ((A ◁ C) ∧ (D ▷ ◊ E)) ▷ (B ∧ □ C ∧ (D ▷ E)))
```

```
R₂-Fm : ∀ {A B C D E F G} → Fm
R₂-Fm {A} {B} {C} {D} {E} {F} {G} =
  A ▷ (B ∧ (C ▷ D)) ↝
  (¬' (A ▷ ¬' E) ∧ (G ▷ ¬' (C ▷ ¬' F)))
  ▷
  (B ∧ (C ▷ D) ∧ (□ E) ∧ (G ▷ C) ∧ (G ▷ D ∧ □ F))

R₂ : ∀ {ℓ} → Principle ℓ
R₂ P = ∀ {A B C D E F G} → (P (R₂-Fm {A} {B} {C} {D} {E} {F} {G}))

R⁰ : ∀ {ℓ} → Principle ℓ
R⁰ P = ∀ {A B C} → P (A ▷ B ↝ (A ◁ C) ▷ (B ∧ □ C))

R¹ : ∀ {ℓ} → Principle ℓ
R¹ P = ∀ {A B C D} → P (A ▷ B ↝ (◇ (D ◁ C) ∧ (D ▷ A)) ▷ (B ∧ □ C))

R² : ∀ {ℓ} → Principle ℓ
R² P = ∀ {A B C D E} → P (A ▷ B ↝ (◇ ((D ▷ E) ∧ ◇ (¬' (D ▷ ¬' C))) ∧
(E ▷ A)) ▷ (B ∧ □ C))

RⁿΩ RⁿU : (m u : Nat) → {u ≤ m} → {C : Fm} → {D : (i : Nat) → {i < suc
m} → Fm} → Fm
RⁿΩ m zero {p} {C} {D} = ¬' (D zero {s≤s p} ▷ ¬' C)
RⁿΩ m (suc k) {p} {C} {D} = (D k {≤-pred (≤-step (s≤s p))} ▷ D (suc k)
{s≤s p}
  ∧ RⁿU m k {≤-pred (≤-step p)} {C} {D})

RⁿU m zero {p} {C} {D} = ◇ (RⁿΩ m zero {p} {C} {D})
RⁿU m (suc k) {p} {C} {D} = ◇ (RⁿΩ m (suc k) {p} {C} {D})

Rⁿ-Fm : (n : Nat) → {A B C : Fm} {D : (i : Nat) → {i < n} → Fm} → Fm
Rⁿ-Fm zero {A} {B} {C} {D} = A ▷ B ↝ ¬' (A ▷ ¬' C) ▷ (B ∧ □ C)
Rⁿ-Fm (suc m) {A} {B} {C} {D} = A ▷ B ↝ (RⁿU m m {≤-reflexive refl}
{C} {D} ∧ (D m {≤-reflexive refl} ▷ A)) ▷ (B ∧ □ C)

Rⁿ : ∀ {ℓ} → Nat → Principle ℓ
Rⁿ r P = ∀ {A B C} → (D : (i : Nat) → {i < r} → Fm) → P (Rⁿ-Fm r {A}
{B} {C} {D})
```

C. Coq library code

Everything that is formalized and proved in Coq is in a single file:

```
From mathcomp Require Import all_ssreflect.
Require Import Coq.Relations.Relation_Definitions.
Require Import Coq.Program.Equality.

Module Base.

Record FromTo A B : Type :=
  fromTo {
      from : (A -> B);
      to : (B -> A)
     }.
Arguments from {A} {B}.
Arguments to {A} {B}.
Notation "a ⇔ b" := (FromTo a b) (at level 99).
Notation "⇒ x" := (from x) (at level 0).
Notation "⇐ x" := (to x) (at level 0).

Definition compose {A} {B} {C} (f : B -> C) (g : A -> B) x := f (g (x)).
Notation "f ∘ g" := (compose f g) (at level 9, right associativity).

Definition relation3 T := T -> relation T.
Notation "a × b" := (prod a b) (at level 96, right associativity).

CoInductive InfiniteChain {W : Type} (R : relation W) (a : W) : Prop
:=
  infinite_chain {
    b : W;
    aRb : R a b;
    tail : InfiniteChain R b;
  }.

Definition Noetherian {W : Type} (R : relation W) : Prop
  := forall {a}, not (InfiniteChain R a).

End Base.

Module Fm.
Import eqtype.
Import Equality.
Definition Var : Type := nat.

Inductive Fm : Type :=
  | var : Var -> Fm
  | rhd : Fm -> Fm -> Fm
  | impl : Fm -> Fm -> Fm
  | bot : Fm.

Notation "⊥" := bot.
Notation "a → b" := (impl a b) (at level 80, right associativity).
Notation "a ▷ b" := (rhd a b) (at level 49, right associativity).
Notation "¬ a" := (a → ⊥) (at level 40).
Notation "a ∨ b" := ((¬ a) → b) (at level 70).
Notation "a ∧ b" := (¬ (a → ¬ b)) (at level 60).
Notation "□ a" := ((¬ a) ▷ ⊥) (at level 30, no associativity).
Notation "◊ a" := (¬ □ (¬ a)) (at level 30).

Fixpoint eqfm (a b : Fm) : bool :=
 match a , b with
   | var v , var v' => v == v'
   | rhd a b , rhd a' b' => eqfm a a' && eqfm b b'
   | impl a b , impl a' b' => eqfm a a' && eqfm b b'
   | bot , bot => true
   | _ , _ => false
 end.
Check Pack.

End Fm.

Module OrdinaryFrame.

Import Base.

Record Frame (W : Type) (R : relation W) (S : relation3 W) : Type :=
  frame {
      witness: W;
      R_trans : transitive _ R;
      R_noetherian : Noetherian R;
      Sw_inc_Rw : forall {w u v}, S w u v -> R w u × R w v;
      Sw_refl: forall {w u}, R w u -> S w u u;
      Sw_trans: forall {w}, transitive _ (S w);
      R_Sw_trans: forall {w u v}, R w u -> R u v -> S w u v
    }.
Arguments R_noetherian {W} {R} {S} f.
Arguments R_trans {W} {R} {S} f [x] [y] [z].
Arguments Sw_refl {W} {R} {S} f [w] [u].
Arguments Sw_inc_Rw {W} {R} {S} f [w] [u] [v].
Arguments Sw_trans {W} {R} {S} f [w] [x].
Arguments R_Sw_trans {W} {R} {S} f [w] [u] [v].

End OrdinaryFrame.

Module OrdinarySemantics.

Export Fm.
```

```
Import Base.
Export OrdinaryFrame.

Definition Valuation (W : Type) : Type := W -> Var -> Prop.

Record Model (W : Type) (R : relation W) (S : relation3 W) (V : Valuation
W) : Type :=
  model {
      frame : Frame W R S
    }.
Arguments frame {W} {R} {S} {V}.

Notation "a ∈ P" := (P a) (at level 0, P at next level, no associativity,
only parsing).

Reserved Notation "M , w ⊩ A"  (at level 0).
Reserved Notation "M , w ⊮ A"  (at level 0).
Inductive forces {W R S V} (M : Model W R S V) (w : W) : Fm -> Prop :=
  | fvar : forall {a : Var}, a ∈ (V w) -> M , w ⊩ (var a)
  | fimp : forall {A B}, M , w ⊮ A \/ M , w ⊩ B -> M , w ⊩ (A → B)
  | frhd : forall {A B}, (forall {u}, R w u -> M , u ⊮ A \/ (ex (fun v
=> (S w u v) × (M , v ⊩ B))))
                  -> M , w ⊩ (A ▷ B)
where "M , w ⊩ A" := (forces M w A)
with nforces {W R S V} (M : Model W R S V) (w : W) : Fm -> Prop :=
  | nvar : forall {a : Var}, not (V w a) -> M , w ⊮ (var a)
  | nimp : forall {A B}, M , w ⊩ A /\ M , w ⊮ B -> M , w ⊮ (A → B)
  | nbot : M , w ⊮ ⊥
  | nrhd : forall {A B},
      ex (fun u => (R w u) × (M , u ⊩ A) ×
          (forall v, (not (S w u v)) \/ M , v ⊮ B )) -> M , w ⊮ (A ▷ B)
where "M , w ⊮ A" := (nforces M w A).

Definition DecidableModel {W R S V} (M : Model W R S V) : Type :=
  forall (w : W) (A : Fm), (M , w ⊩ A) \/ (M , w ⊮ A).
End OrdinarySemantics.

Module OrdinarySemanticsProperties.

Import OrdinarySemantics.
Import Base.
Import Fm.

Parameter W : Type.
Parameter R : relation W.
Parameter S : relation3 W.
Parameter V : Valuation W.
Parameter M : Model W R S V.
Parameter dec : DecidableModel M.

Local Notation "w ⊩? A" := (dec w A) (at level 0).
```

```
Definition frame : Frame W R S := frame M.

Definition forces_bot0 : not (is_true false) :=
  fun a => match a with end.

Lemma forces_bot1 {w A} : (M , w ⊩ A) -> not (A = ⊥).
  by case.
Qed.

Lemma forces_bot {w} : not (M , w ⊩ ⊥).
Proof using Type.
  inversion 1. Qed.
Notation "⊩⊥" := forces_bot.

Lemma forces_bot2 {w} : not (M , w ⊩ ⊥).
  by move => h; apply forces_bot1 in h.
Qed.

Lemma forces_var {v w} : M , w ⊩ (var v) -> V w v.
Proof using Type. inversion_clear 1 => //. Qed.

Lemma fimp_inv {w A B} : M , w ⊩ (A → B) -> M , w ⊮ A \/ M , w ⊩ B.
Proof using Type. inversion_clear 1 => //. Qed.

Lemma forces_rhd {w A B} :
  M , w ⊩ (A ▷ B) ->
      forall {u}, R w u -> nforces M u A \/ (ex (fun v => (S w u v) ×
(forces M v B))).
Proof using Type. inversion_clear 1 => //. Qed.

Lemma fimp_inv2 {w A B} : M , w ⊩ (A → B) -> M , w ⊮ A \/ M , w ⊩ B.
Proof using Type.
  by move => h; inversion h.
Qed.

Lemma nforces_var {w p} : M , w ⊮ (var p) -> not (V w p).
Proof using Type.
  by inversion_clear 1.
Qed.

Lemma nforces_rhd {w A B} :
  M , w ⊮ (A ▷ B) ->
     ex (fun u => (R w u) × (forces M u A) ×
           (forall v, (not (S w u v)) \/ nforces M v B )).
Proof using Type. by inversion_clear 1 => //. Qed.

Lemma nimp_inv {w A B} :
  M , w ⊮ (A → B) ->
      forces M w A /\ nforces M w B.
Proof using Type. inversion_clear 1 => //. Qed.
```

```
Lemma nforces_to_not_forces {w A} : M , w ⊮ A -> not (M , w ⊩ A).
Proof using Type.
  move: A w.
  elim => [ v w a na | A ihA B ihB w | A ihA B ihB w | w b ].
  +  dependent destruction a.
  + dependent destruction na => //.
  + move/nforces_rhd=> [u [Rwu [uA p]] /forces_rhd-wArhdB].
    case: (wArhdB u Rwu) => [/ihA-h1 //| [z [Swuz zB]]].
  + case: (p z) => [//|/ihB //].
    move/nimp_inv=> [wA wB /fimp_inv[/ihA //|/ihB //]].
  inversion_clear 1.
Qed.

Lemma forces_to_not_nforces {w A} : M , w ⊩ A -> not (M , w ⊮ A).
Proof using Type.
  move: A w.
  elim => [ v w /forces_var-a /nforces_var-na //
         | A ihA B ihB w /forces_rhd-ArhdB /nforces_rhd[u [Rwu [uA pu]]
]
        | A ihA B ihB w /fimp_inv-[nwA | wB] /nimp_inv[/ihA-wA /ihB-nwB
//]
          | w ].
  case: (ArhdB _ Rwu) => [/ihA //|[v [Swuv /ihB]]].
  case: (pu v) => [//|//].
  inversion_clear 1.
Qed.

Notation "⊮→¬⊩" := nforces_to_not_forces.
Notation "⊩→¬⊮" := forces_to_not_nforces.

Lemma not_forces_to_nforces {w A} : not (M , w ⊩ A) -> M , w ⊮ A.
Proof using Type.
  case (w ⊩? A) => [//|//].
Qed.

Lemma not_nforces_to_forces {w A} : not (M , w ⊮ A) -> M , w ⊩ A.
Proof using Type.
  case (w ⊩? A) => [//|//].
Qed.

Lemma forces_fromto_not_nforces {w A} : (M , w ⊩ A) ⇔ (not (M , w ⊮
A)).
Proof using Type.
  apply: fromTo.
  apply: forces_to_not_nforces.
  apply: not_nforces_to_forces.
Qed.
Notation "⊩⇔¬⊮" := forces_fromto_not_nforces.

Lemma not_forces_fromto_nforces {w A} : (not (M , w ⊩ A)) ⇔ (M , w ⊮
A).
Proof using Type.
  apply: fromTo.
  apply: not_forces_to_nforces.
  apply: nforces_to_not_forces.
Qed.
Notation "¬⊩⇔⊮" := not_forces_fromto_nforces.

Lemma fimp_inv' {w A B} : (M , w ⊩ (A → B)) ⇔ (M , w ⊩ A -> M , w ⊩ B).
Proof using Type.
  apply: fromTo => [/fimp_inv-[/(⇐ ¬⊩⇔⊮)  //|//]
                  | AtoB].
  apply: fimp.
  case: (w ⊩? A) => [/AtoB-wB|].
  apply: or_intror => //.
  apply: or_introl => //.
Qed.
Notation "⊩→" := (fimp_inv').

Lemma forces_rhd' {w A B} :
  (M , w ⊩ (A ▷ B)) ⇔
    forall {u}, R w u -> forces M u A -> (ex (fun v => (S w u v) ×
(forces M v B))).
Proof using Type.
  apply: fromTo => [/forces_rhd-a u Rwu uA|p].
  case: (a _ Rwu) => [/(⇐ ¬⊩⇔⊮) //|//].
  apply: frhd => [u Rwu].
  case: (u ⊩? A) => [uA |nuA].
  apply: or_intror. apply: p => //.
  apply: or_introl => //.
Qed.
Notation "⊩▷" := (forces_rhd').

Lemma forces_neg {w A} : (M , w ⊩ ¬ A) ⇔ (M , w ⊮ A).
Proof using Type.
  apply: fromTo => [/(⇒ ⊩→)-a|/(⇐ ¬⊩⇔⊮)-nwA].
  case: (w ⊩? A) => [/a/⊩⊥//|//].
  apply: (⇐ ⊩→) => //.
Qed.
Notation "⊩¬" := forces_neg.

Lemma nforces_neg {w A} : (M , w ⊮ ¬ A) ⇔ (M , w ⊩ A).
Proof using Type.
  apply: fromTo => [/nimp_inv-[a //]|a].
  apply: nimp. apply: conj => //. apply: nbot.
Qed.
Notation "⊮¬" := nforces_neg.

Lemma forces_box {w A} :
  (M , w ⊩ (□ A)) ⇔ (forall {u}, R w u -> M , u ⊩ A).
Proof using Type.
```

```
  apply: fromTo => [/forces_rhd-p u Rwu|a].
  case: (p _ Rwu) => [/(⇒ ⊩¬)//|[v [_ /⊩⊥ //]]].
  apply: frhd => [u Rwu]. apply: or_introl.
  apply: (⇐ ⊩¬). apply: a => //.
Qed.
Notation "⊩□" := forces_box.

Lemma nforces_box {w A} :
   (M , w ⊮ (□ A)) ⇔ (ex (fun v => (R w v) × (M , v ⊮ A))).
Proof using Type.
  apply: fromTo => [/nforces_rhd-[u [Rwu [/(⇒ ⊩¬)uA pA]]]|[u [Rwu /(⇐
⊩¬)nuA]]].
  exists u => //.
  apply: nrhd. exists u. apply: conj => //. apply: conj => //.
  move => v. apply: or_intror. apply: nbot.
Qed.
Notation "⊮□" := nforces_box.

Lemma forces_diamond {w A} :
   (M , w ⊩ (◊ A)) ⇔ (ex (fun v => (R w v) × (M , v ⊩ A))).
Proof using Type.
   apply: fromTo => [/(⇒ ⊩¬)/(⇒ ⊮□) [u [Rwu /(⇒ ⊮¬)-uA]]|[u [Rwu /(⇐
⊮¬)uA]]].
  exists u => //. apply: (⇐ ⊩¬). apply: (⇐ ⊮□). exists u => //.
Qed.
Notation "⊩◊" := forces_diamond.

Lemma nforces_diamond {w A} :
   (M , w ⊮ (◊ A)) ⇔ (forall {u}, R w u -> M , u ⊮ A).
Proof using Type.
  apply: fromTo => [/(⇒ ⊮¬)/(⇒ ⊩□) p u Rwu|p].
  apply: (⇒ ⊩¬). apply: p => //.
  apply: (⇐ ⊮¬). apply: (⇐ ⊩□) => [u Rwu].
  apply: (⇐ ⊩¬). apply: p => //.
Qed.
Notation "⊮◊" := nforces_diamond.

Lemma forces_and {w A B} :
   (M , w ⊩ (A ∧ B)) ⇔ (M , w ⊩ A × M , w ⊩ B).
Proof using Type.
  apply: fromTo => [/(fimp_inv) [/nimp_inv [a /(⇒ ⊮¬) b]|/⊩⊥ //]
                   |[a /(⇐ ⊮¬)b]].
  apply: conj => //.
  apply: fimp. apply: or_introl. apply: nimp. apply: conj => //.
Qed.
Notation "⊩∧" := forces_and.

Lemma nforces_and {w A B} : (M , w ⊮ (A ∧ B)) ⇔ (M , w ⊮ A \/ M , w ⊮
B).
Proof using Type.
 apply: fromTo => [/nimp_inv-[/fimp_inv-[a|/(⇒ ⊩¬)b] _]|[a|/(⇐ ⊩¬)b]].
```

```
  apply: or_introl => //. apply: or_intror => //.
  apply: nimp. apply: conj. apply: fimp. apply: or_introl => //. apply:
nbot.
  apply: nimp. apply: conj. apply: fimp. apply: or_intror => //. apply:
nbot.
Qed.
Notation "⊮∧" := nforces_and.

Lemma forces_or {w A B} :
   (M , w ⊩ (A ∨ B)) ⇔ (M , w ⊩ A \/ M , w ⊩ B).
Proof using Type.
  apply: fromTo => [/fimp_inv-[/(⇒ ⊮¬)a|b]|[/(⇐ ⊮¬)a|b]].
  apply: or_introl => //.
  apply: or_intror => //.
  apply: fimp. apply: or_introl => //.
  apply: fimp. apply: or_intror => //.
Qed.
Notation "⊩∨" := forces_or.

Lemma nforces_or {w A B} :
   (M , w ⊮ (A ∨ B)) ⇔ (M , w ⊮ A × M , w ⊮ B).
Proof using Type.
  apply: fromTo => [/nimp_inv-[/(⇒⊩¬)a b] |[/(⇐ ⊩¬)a b]].
  apply: conj => //.
  apply: nimp. apply: conj => //.
Qed.
Notation "⊮∨" := nforces_or.

Inductive subset {A} (P Q : A -> Prop) : Type :=
  | ss : (forall {x}, (P x -> Q x)) -> subset P Q.
Arguments ss {A} {P} {Q}.

Lemma forces_J1 {w A B} : M , w ⊩ (□ (A → B) → A ▷ B).
Proof using Type.
  apply: (⇐ ⊩→) => /(⇒ ⊩□)p.
  apply: (⇐ ⊩▷) => [u Rwu uA].
  exists u. apply: conj => //.
  apply: (Sw_refl frame). apply: Rwu.
  apply: ⇒⊩→(p _ Rwu)uA.
Qed.
Notation "⊩J1" := forces_J1.

Lemma forces_J2 {w A B C} : M , w ⊩ (A ▷ B ∧ B ▷ C → A ▷ C).
Proof using Type.
  apply: (⇐ ⊩→) => /(⇒ ⊩∧) [/(⇒⊩▷)ab /(⇒⊩▷)bc].
  apply: (⇐ ⊩▷) => [u Rwu uA].
  pose proof (ab _ Rwu uA) as [v [Swuv vB]].
  pose proof (Sw_inc_Rw frame Swuv) as [_ Rwv].
  pose proof (bc v Rwv vB) as [z [Swvz zC]].
  exists z. apply: conj => //.
  apply: (Sw_trans frame). apply: Swuv. apply: Swvz.
```

```
Qed.
Notation "⊩J2" := forces_J2.

Lemma forces_J3 {w A B C} : M , w ⊩ (A ▷ C ∧ B ▷ C → (A ∨ B) ▷ C).
Proof using Type.
  apply: (⇐ ⊩→) => /(⇒ ⊩∧)[/(⇒ ⊩▷)ac /(⇒ ⊩▷)bc].
  apply: (⇐ ⊩▷) => [u Rwu /(⇒ ⊩∨)[uA|uB]].
  apply: ac => //.
  apply: bc => //.
Qed.

Lemma forces_J4 {w A B} : M , w ⊩ ((A ▷ B) → ◊ A → ◊ B).
Proof using Type.
  apply: (⇐ ⊩→) => /(⇒ ⊩▷)p. apply: (⇐ ⊩→) => /(⇒ ⊩◊)[v [Rwv vA]].
  apply: (⇐ ⊩◊).
  pose proof (p _ Rwv vA) as [z [Swvz zB]].
  pose proof (Sw_inc_Rw frame Swvz) as [_ Rwz].
  exists z. apply: conj => //.
Qed.

Lemma forces_J5 {w A} : M , w ⊩ ((◊ A) ▷ A).
Proof using Type.
  apply: (⇐ ⊩▷) => [u Rwu /(⇒ ⊩◊)[v [Ruv vA]]].
  exists v. pose proof (R_Sw_trans frame Rwu Ruv) as Swuv.
  apply: conj => //.
Qed.

Lemma forces_K {w A B} : M , w ⊩ ((□ (A → B)) → (□ A → □ B)).
Proof using Type.
  apply: (⇐ ⊩→) => /(⇒⊩□)pab.
  apply: (⇐ ⊩→) => /(⇒⊩□)pa.
  apply: (⇐ ⊩□) => [u Rwu].
  apply: (⇒⊩→)(pab u Rwu)(pa _ Rwu).
Qed.

Lemma forces_4 {w A} : M , w ⊩ (□ A) -> M , w ⊩ (□ (□ A)).
Proof using Type.
  move=> /(⇒⊩□)boxA.
  apply: ⇐ ⊩□ => u Rwu.
  apply: ⇐ ⊩□ => v Ruv.
  apply: boxA (R_trans frame Rwu Ruv).
Qed.

Lemma L_chain {w u A} : R w u -> M , u ⊮ A -> M , w ⊩ (□ (□ A → A)) ->
InfiniteChain R w.
Proof using Type.
  move: w u A.
  cofix h => w u A Rwu uA p.
  pose proof fimp_inv((⇒ ⊩□ p) _ Rwu) as [nboxA|a].
  pose proof (⇒ ⊮□ nboxA) as [v [Ruv nvA]].
  apply: infinite_chain. apply: Rwu.
  apply: h. apply: Ruv. apply: nvA.
  apply: ⇒ ⊩□ (forces_4 p) _ Rwu.
  exfalso. apply: (⇒ ⊩⇔¬⊮) a uA.
Qed.


Lemma forces_L {w A} : M , w ⊩ (□ (□ A → A) → □ A).
Proof using Type.
  apply: (⇐⊩→) => p. apply: (⇐ ⊩□) => u Rwu.
  case: (dec u A) => [//|nuA].
  exfalso. apply: (R_noetherian frame).
  apply: (L_chain Rwu nuA p).
Qed.

Lemma forces_C1 {w A B} : M , w ⊩ (A → B → A).
Proof using Type.
  apply: (⇐ ⊩→) => wA. apply: (⇐ ⊩→) => //.
Qed.

Lemma forces_C2 {w A B C} : M , w ⊩ ((A → (B → C)) → ((A → B) → (A →
C))).
Proof using Type.
  apply: (⇐ ⊩→) => wABC. apply: (⇐ ⊩→) => /(⇒ ⊩→)wAB. apply: (⇐ ⊩→) =>
wA.
  apply: (⇒⊩→(⇒⊩→ wABC wA)(wAB wA)).
Qed.

Lemma forces_C3 {w A B} : M , w ⊩ ((¬ A → ¬ B) → B → A).
Proof using Type.
  apply: (⇐⊩→) => /(⇒⊩∨)[a|/(⇐¬⊩⇔⊮ ∘ ⇒⊩¬)b]; apply: (⇐ ⊩→) => //.
Qed.

Inductive ILProof (Π : list Fm) : Fm -> Prop :=
  (* | Ax : forall {A}, In A Π -> ILProof Π A *)
  | C1 : forall {A B}, ILProof Π (A → (B → A))
  | C2 : forall {A B C}, ILProof Π ((A → (B → C)) → ((A → B) → (A → C)))
  | C3 : forall {A B}, ILProof Π ((¬ A → ¬ B) → B → A)
  | K : forall {A B}, ILProof Π ((□ (A → B)) → (□ A → □ B))
  | L : forall {A}, ILProof Π (□ (□ A → A) → □ A)
  | J1 : forall {A B}, ILProof Π (□ (A → B) → (A ▷ B))
  | J2 : forall {A B C}, ILProof Π (A ▷ B ∧ B ▷ C → (A ▷ C))
  | J3 : forall {A B C}, ILProof Π (((A ▷ C) ∧ (B ▷ C)) → ((A ∨ B) ▷ C))
  | J4 : forall {A B},  ILProof Π ((A ▷ B) → ◊ A → ◊ B)
  | J5 : forall {A},  ILProof Π (◊ A ▷ A)
  | MP : forall {A B}, ILProof Π (A → B) -> ILProof Π A -> ILProof Π B
  | nec : forall {A}, ILProof nil A -> ILProof Π (□ A).
Notation "Π ⊢ A" := (ILProof Π A) (at level 50).

Lemma soundness {D} (w : W) (p : nil ⊢ D) : M , w ⊩ D.
Proof using Type.
  elim: p w => [Π A B w
```

```
          |Π A B C w
          |Π A B w
          |Π A B w
          |Π A w
          |Π A B w
          |Π A B C w
          |Π A B C w
          |Π A B w
          |Π A w
          |Π A B _ hAB _ hA w
          |Π A ilA h w].
  apply: forces_C1.
  apply: forces_C2.
  apply: forces_C3.
  apply: forces_K.
  apply: forces_L.
  apply: forces_J1.
  apply: forces_J2.
  apply: forces_J3.
  apply: forces_J4.
  apply: forces_J5.
  apply: (⇒⊩→(hAB _) (hA _)).
  apply: (⇐⊩□) => u Rwu. apply: h.
Qed.

End OrdinarySemanticsProperties.
```

D. Manuscript by Verbrugge: Set Veltman frames and models

In the following pages we copy a scan of a manuscript by Verbrugge (see notes below). It has not been processed or edited by us in any way. The original title, in Dutch, is *Verzamelingen-Veltman frames en modellen.*

Annotations are probably from various authors, like Dick de Jongh and possibly others. Verbrugge commented in private correspondence that in particular the "English translation on Page 1a" on the first typed page is from her own hand and likely the "$\setminus\emptyset$" and the two x's on that same page too while the accolades on handwritten Page 1a are not her handwriting. These can most likely be attributed to Dick de Jongh together with the phrase $= KM_1$ on Page 2 of the handwritten part. The double exclamations marks have an uncertain author and they are possibly from somebody else altogether. Possibly the "Rineke Verbrugge, manuscript, 1992" at the top of the first typed page is from Troelstra's hand.

Rineke Verbrugge, manuscript, 1992

Verzamelingen-Veltman frames en modellen

Definitie 1.

Een *IL_{Set}frame* is een L-frame $\langle W,R\rangle$ met een extra relatie S_w voor iedere $w\in W$, met de volgende eigenschappen:

(I) $S_w \subseteq W[w] \times P(W[w]) \setminus \emptyset$

(II) S_w is "reflexief": Als wRx, dan $xS_w\{x\}$

S_w is "transitief": Als xS_wY dan geldt voor alle $y\in Y$ en alle $V\in P(W[w])$: $yS_wV \Rightarrow xS_wV$.

(III) Als $wRw'Rw''$, dan $w'S_w\{w''\}$

English translation on page 1a

Een *IL_{Set} model* bestaat uit een IL_{Set} frame $\langle W,R,S\rangle$ met een forcing relatie $\Vdash$ die voldoet aan:

$$u\Vdash\Box\varphi \iff \forall v(uRv \Rightarrow v\Vdash\varphi)$$

$$u\Vdash\varphi\rhd\psi \iff \forall v(uRv \text{ en } v\Vdash\varphi \Rightarrow \exists V(vS_uV \text{ en } \forall \overset{x}{\cancel{y}}\in V\ \overset{x}{\cancel{y}}\Vdash\psi))$$

Het is niet moeilijk in te zien dat voor ieder IL_{Set} model K, $K\models IL$.

From IL_{Set} models to IL models

Theorem 2.

Let $\langle W,R,S,\Vdash\rangle$ be an IL_{Set} model. Then there is an IL model $\langle W',R',S',\Vdash'\rangle$ and a bijection $f:W\rightarrow P(W')$ such that for all $w\in W$ and all $w'\in f(w)$: $w\Vdash\varphi \iff w'\Vdash'\varphi$.

Proof.

Let W' consist of all points $\langle x,A\rangle$, where $x\in W$ and A is a set of ordered pairs such that:

For all w, V with xS_wV, there is a $v\in V$ with $\langle w,v\rangle\in A$;

and, conversely, if $\langle w,v\rangle\in A$, then there is a V such that $v\in V$ and xS_wV.

Define R', S' as follows:

$\langle x,A\rangle R'\langle y,B\rangle \iff xRy$ and for all w such that wRx and all z: if $\langle w,z\rangle\in B$, then $\langle w,z\rangle\in A$.

$\langle x,A\rangle S'_{\langle w,C\rangle}\langle y,B\rangle \iff \langle w,C\rangle R'\langle x,A\rangle$, $\langle w,C\rangle R'\langle y,B\rangle$ and for all v:

if $\langle w,v\rangle \in B$, then $\langle w,v\rangle \in A$
(thus in particular, because S_w's being "reflexive" implies $\langle w,y\rangle \in B$, we have $\langle w,y\rangle \in A$).

Finally, define $\Vdash'$ as follows:
$\langle x,A\rangle \Vdash' p \iff x \Vdash p$.

We will prove that

(a) $\langle W',R',S'\rangle$ is an IL frame

(b) For all φ, $\langle x,A\rangle \Vdash' \varphi \iff x \Vdash \varphi$

Proof of (a):
First of all, it is not difficult to see that $\langle W',R'\rangle$ is an L-frame. Checking the clauses for $S'_{\langle w,C\rangle}$ requires a bit more work.

(I') If $\langle w,C\rangle \in W'$, then $S'_{\langle w,C\rangle}$ is a relation on $W'[\langle w,C\rangle]$; this follows immediately from the definition of $S'_{\langle w,C\rangle}$.

(II') $S'_{\langle w,C\rangle}$ is reflexive; for suppose $\langle w,C\rangle R' \langle x,A\rangle$, then by the definition of W': $\langle x,A\rangle S'_{\langle w,C\rangle} \langle x,A\rangle$.
$S'_{\langle w,C\rangle}$ is transitive; for suppose $\langle x,A\rangle S'_{\langle w,C\rangle} \langle y,B\rangle S'_{\langle w,C\rangle} \langle z,D\rangle$, then $\langle w,C\rangle R' \langle x,A\rangle$, $\langle w,C\rangle R' \langle z,D\rangle$ and for all v : if $\langle w,v\rangle \in D$, then $\langle w,v\rangle \in B$, and thus $\langle w,v\rangle \in A$; therefore $\langle x,A\rangle S'_{\langle w,C\rangle} \langle z,D\rangle$.

(III') If $\langle w,C\rangle R' \langle x,A\rangle R' \langle y,B\rangle$, then wRx and by definition of R': for all z, if $\langle w,z\rangle \in B$, then $\langle w,z\rangle \in A$; therefore $\langle x,A\rangle S'_{\langle w,C\rangle} \langle y,B\rangle$.

Proof of (b):
As usual, the only interesting case is the induction step for $\rhd$.

Suppose $w \Vdash \varphi \rhd \psi$ and $\langle w,C\rangle \in W'$. We want to prove $\langle w,C\rangle \Vdash' \varphi \rhd \psi$.
So suppose $\langle w,C\rangle R' \langle x,A\rangle$ and $\langle x,A\rangle \Vdash' \varphi$.
Then wRx and, by the induction hypothesis, $x \Vdash \varphi$. Therefore, there is a V with xS_wV and $\forall y \in V\ y \Vdash \psi$. We want to find a B such that $\langle x,A\rangle S'_{\langle w,C\rangle} \langle y,B\rangle$. This is possible because of the following two facts:

(1) By "transitivity" of S_w, we have $\forall V\ (yS_wV \Rightarrow xS_wV)$

(2) For any b with $bRwRy$ we have, by (III), $wS_b\{y\}$, and thus by transitivity: if yS_bV, then wS_bV.

Therefore, we can take a B such that $\langle y,B\rangle \in W'$ and, by (1), for all v, if $\langle w,v\rangle \in B$, then $\langle w,v\rangle \in A$; moreover wRy and, by (2), if bRw, then for all z: if $\langle b,z\rangle \in B$, then $\langle b,z\rangle \in C$, so $\langle w,C\rangle R'\langle y,B\rangle$. In conclusion, $\langle x,A\rangle S'_{\langle w,C\rangle}\langle y,B\rangle$ and by induction hypothesis, $\langle y,B\rangle \Vdash' \psi$.

Suppose on the other hand that $w \Vdash \neg(\varphi \rhd \psi)$. We will prove $\langle w,C\rangle \Vdash' \neg(\varphi \rhd \psi)$.

First, $w \Vdash \neg(\varphi \rhd \psi)$ implies that there is an x with wRx and for all V with xS_wV there is a $y \in V$ such that $y \Vdash \neg\psi$.

Therefore, it is possible to take an A such that $\langle x,A\rangle \in W'$ and for all y: if $\langle w,y\rangle \in A$, then $y \Vdash \neg\psi$, and moreover $\langle w,C\rangle R'\langle x,A\rangle$ (the extra clause for R' doesn't interfere with our desiderations for A).

For this $\langle x,A\rangle$, we have by induction hypothesis $\langle x,A\rangle \Vdash \varphi$.

Moreover, if $\langle x,A\rangle S'_{\langle w,C\rangle}\langle y,B\rangle$, then $\langle w,y\rangle \in B$, and thus $\langle w,y\rangle \in A$ [see earlier remark], so by induction hypothesis $\langle y,B\rangle \Vdash \neg\psi$

Def $W[w] := \{w' \in W \mid wRw'\}$

Def

An IL_{Set} frame is an L-frame $\langle W, R\rangle$ + for each $w \in W$, a relation S_w with the following properties:

(I) $S_w \subseteq W[w] \times \mathcal{P}(W[w]) \setminus \{\emptyset\}$

(II) S_w is "quasi-reflexive":
for all w, x, $wRx \longrightarrow x S_w \{x\}$

S_w is "quasi-transitive":
$x S_w Y \longrightarrow \forall y \in Y, \forall V \in \mathcal{P}(W[w]) (y S_w V \Rightarrow x S_w V)$

(III) $wRw'Rw'' \longrightarrow w' S_w \{w''\}$.

An IL_{Set} model consists of an IL_{Set}-frame $\langle W, R, \{S_w : w \in W\}\rangle$ + a forcing relation $\Vdash$ satisfying the following:

$u \Vdash \Box\varphi \iff \forall v (uRv \Rightarrow v \Vdash \varphi)$

$u \Vdash \varphi \rhd \psi \iff \forall v (uRv \wedge v \Vdash \varphi \Rightarrow \exists V (v S_u V \wedge \forall x \in V\ x \Vdash \psi$

It is not difficult to check that for any IL_{Set}-model K, $K \models IL$.

It seems that for some applications, transitivity is more nicely defined as:

$x S_w Y \wedge \forall y \in Y (y S_w Z_y) \rightarrow x S_w \bigcup_{y \in Y} Z_y$

+ extra condition:

$x S_w Y \wedge Y \subseteq Z \longrightarrow x S_w Z$.

We didn't check what happens in this case, and in the rest of these pages we always use the first definition

$M: \underline{A \rhd B \longrightarrow A \wedge \Box C \rhd B \wedge \Box C}$

What property does M characterize on set models?

$M*)\ u S_w V \longrightarrow \exists V' \subseteq V, (u S_w V' \wedge \forall v \in V' (vRz \to uRz))$

Proof

1) Suppose our frame satisfies $M*$ and $w \Vdash A \rhd B$.

Suppose wRu, $u \Vdash A \wedge \Box C$. Because $w \Vdash A \rhd B$,

$\exists V\ \ u S_w V\ \&\ \forall v \in V\ v \Vdash B$.

Now by $*$,

$\exists V' \subseteq V\ (u S_w V' \wedge \forall v \in V' (vRz \to uRz))$

so $\forall v \in V',\ v \Vdash B \wedge \Box C$.

Therefore $w \Vdash A \wedge \Box C \rhd B \wedge \Box C$.

2) Suppose our frame does not satisfy $M*$, i.e.

Suppose $u S_w V$ but $\forall V' \subseteq V (u S_w V' \to \exists v \in V' (vRz \wedge \neg uRz$

now take a valuation which:

- forces p only in u
- forces r in V but nowhere else
- !! Does not force q in those $v \in V$ with $vRz \wedge \neg uRz$, but does force q everywhere else.

Then $w \nVdash p \wedge \Box q \rhd r \wedge \Box q$,

because wRu, $u \Vdash p \wedge \Box q$.

But for any V' with $u S_w V'$ and $V' \Vdash r$, we have $V' \subseteq V$, so $V' \nVdash \Box q$.

Π: $A \rhd \Diamond B \rightarrow \Box(A \rightarrow \Diamond B)$

What property is characterized by Π on set frames
(On normal frames it is the same as the M-property)

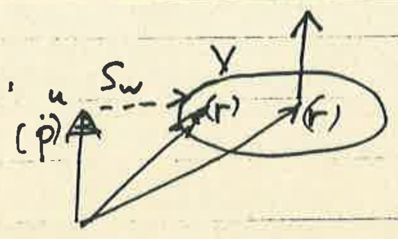
Π* : If uS_wV, then $\exists v \in V\ \forall z(vRz \rightarrow uRz)$

Proof 1) Suppose our frame satisfies Π*,
and suppose $w \Vdash A \rhd \Diamond B$. Suppose wRu and $u \Vdash A$
Then there is a V such that uS_wV, and
for all $v \in V\ \exists z_v\ vRz_v\ \&\ z_v \Vdash B$.
By *), $\exists v \in V$ such that $\forall z(vRz \rightarrow uRz)$,
So, especially uRz_v, thus $u \Vdash \Diamond B$.
Therefore $w \Vdash \Box(A \rightarrow \Diamond B)$.

2) Suppose our frame does not satisfy Π*, i.e. we suppose
$uS_wV\ \&\ \forall v \in V\ \exists z_v\ (vRz_v \wedge \neg uRz_v)$.
We take a valuation such that
- p is forced only in u
- q is forced only in the "z_v's".

Then $w \Vdash p \rhd \Diamond q$, because
uS_wV and $\forall v \in V$, $v \Vdash \Diamond q$ (because $z_v \Vdash q$).
But not $w \Vdash \Box(p \rightarrow \Diamond q)$, because wRu and
$u \nVdash \Diamond q$ (u has no R-arrow to any "z_v").

$IL\Pi \nvDash M$: Countermodel



This frame satisfies property Π*, but
$w \nVdash p \rhd r \rightarrow p \wedge \Box\bot \rhd r \wedge \Box\bot$

$P: \; A \rhd B \to \Box(A \rhd B)$

What property does P characterize on set frames?

$*P: \; u S_w V \wedge w R w' R u \to \exists V' \subseteq V \; u S_{w'} V'$

Proof

1) Suppose our frame satisfies $*P$,
and suppose $w \Vdash A \rhd B$.
Moreover suppose wRw', $w'Ru$ and $u \Vdash A$.
We have by transitivity wRu, so there is a V
with $u S_w V$ and $\forall v \in V \; v \Vdash B$.
By $*P$, $\exists V' \subseteq V \; u S_{w'} V'$, and $\forall v \in V' \; v \Vdash B$.
So $w' \Vdash A \rhd B$, thus $w \Vdash \Box(A \rhd B)$.

2) Suppose our frame does not satisfy $*P$, e.g.
suppose
$u S_w V$, $wRw'Ru$, but $\forall V' \subseteq V$ not $u S_{w'} V'$.
Take a valuation such that
- p is forced only in u
- q is forced everywhere in V, but nowhere else.

Then $w \Vdash p \rhd q$, but
$w' \nVdash p \rhd q$, so $w \nVdash (p \rhd q) \to \Box(p \rhd q)$.

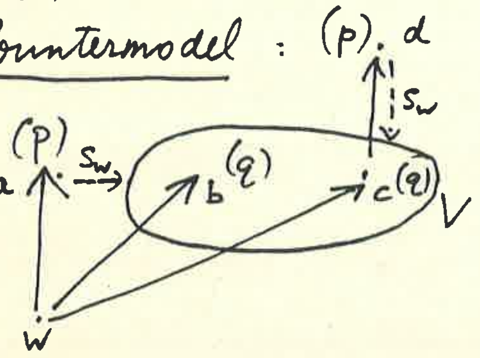
F: $\varphi \rhd \Diamond\varphi \longrightarrow \Box\neg\varphi$

W: $\varphi \rhd \psi \longrightarrow \varphi \rhd \psi \wedge \Box\neg\varphi$

KW1: $\varphi \rhd \Diamond\top \longrightarrow \top \rhd \neg\varphi$

<u>Th</u> $ILF \nvdash W$

<u>Countermodel</u> :



1) $w \nVdash p \rhd q \longrightarrow p \rhd q \wedge \Box\neg p$.

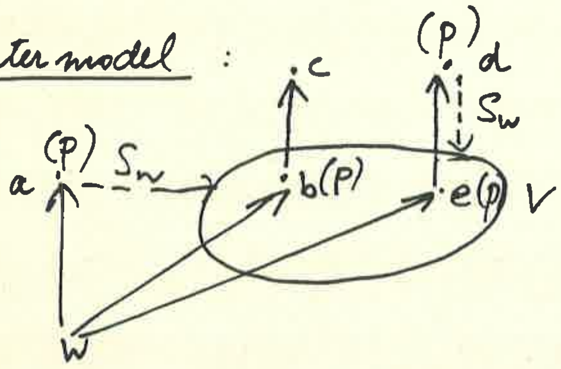
<u>Pf</u>. For, $w \Vdash p \rhd q$: from both points in which p holds (a and d), one can reach V by S_w, and $V \Vdash q$.
Also $w \nVdash p \rhd q \wedge \Box\neg p$: from a, we cannot reach any V' with $V' \Vdash q \wedge \Box\neg p$, because q holds only in V, and $V \nVdash \Box\neg p$ because cRd, $d \Vdash p$.

2) for all φ, $w \Vdash \varphi \rhd \Diamond\varphi \longrightarrow \Box\neg\varphi$. So, $w \Vdash ILF$

<u>Pf</u>. Notice first that a and d are indistinguishable: They force exactly the same sentences (proof by induction).
Now suppose that $w \Vdash \varphi \rhd \Diamond\varphi$.
Suppose $a \Vdash \varphi$, Then we must have $a \Vdash \Diamond\varphi$ or $V \Vdash \Diamond\varphi$ $\lightning$
So $a \nVdash \varphi$ and $d \nVdash \varphi$.
Suppose $c \Vdash \varphi$, Then $c \Vdash \Diamond\varphi$ and $d \Vdash \varphi$ $\lightning$, so $c \nVdash \varphi$.
Suppose ~~b~~ $b \Vdash \varphi$, then $b \Vdash \Diamond\varphi$ $\lightning$, so $b \nVdash \varphi$.
Thus, $w \Vdash \Box\neg\varphi$.

<u>Th</u> $\mathrm{ILF} \nvdash \mathrm{KW1}$

<u>Countermodel</u> :



1) $w \nVdash p \rhd \Diamond\top \longrightarrow \top \rhd \neg p$

<u>Pf</u> $w \Vdash p \rhd \Diamond\top$: from a, we can reach by S_w V, and $V \Vdash \Diamond\top$
from b, we reach $\{b\}$, and $b \Vdash \Diamond\top$
from e, we reach $\{e\}$ and $e \Vdash \Diamond\top$
from d, we reach V , and $V \Vdash \Diamond\top$.
$w \nVdash \top \rhd \neg p$: from a, we can not reach by S_w any set V' with $V' \Vdash \neg p$.

2) For all φ, $w \Vdash \varphi \rhd \Diamond\varphi \to \Box\neg\varphi$.

<u>Pf</u>. Again, a and d are indistinguishable.
Suppose that $w \Vdash \varphi \rhd \Diamond\varphi$. Suppose $c \Vdash \varphi$. Then $c \Vdash \Diamond\varphi$ ↯ ∴ $c \nVdash \varphi$
Suppose $a \Vdash \varphi$, then either $a \Vdash \Diamond\varphi$ or $V \Vdash \Diamond\varphi$.
so especially $c \Vdash \varphi$ ↯. So $a \nVdash \varphi$ and $d \nVdash \varphi$.
Suppose $b \Vdash \varphi$, then either $b \Vdash \Diamond\varphi$ or $c \Vdash \varphi$, in both cases $c \Vdash \varphi$ ↯, so $b \nVdash \varphi$
Suppose $e \Vdash \varphi$, then $e \Vdash \Diamond\varphi$ or $d \Vdash \varphi$, in either case $d \Vdash \varphi$ ↯, so $e \nVdash \varphi$.
We conclude $w \nVdash \Box\varphi$