# Dynamic Epistemic Logic for Guessing Games and Cryptographic Protocols

**MSc Thesis** *(Afstudeerscriptie)*

written by

**Malvin Gattinger**
(born October 10th, 1988 in Offenbach am Main, Germany)

under the supervision of **Prof Dr Jan van Eijck**, and submitted to the Board of Examiners in partial fulfillment of the requirements for the degree of

**MSc in Logic**

at the *Universiteit van Amsterdam.*

| **Date of the public defense:** | **Members of the Thesis Committee:** |
|---|---|
| *June 20th, 2014* | Dr Maria Aloni (chair) |
| | Dr Alexandru Baltag |
| | Dr Christian Schaffner |
| | Prof Dr Jan van Eijck (supervisor) |
| | Prof Dr Johan van Benthem (in absentia) |
| | Dr Joshua Sack (in absentia) |

Institute for Logic, Language and Computation

## Abstract

We present two variants of Dynamic Epistemic Logic based on a new formal representation of what it means to know a number.
The *Logic of Guessing Games* GG formalizes number guessing games and information updates happening in such games. We discuss many examples and provide a sound and complete axiomatization of GG.
With *Epistemic Crypto Logic* ECL we apply the idea of register models to the analysis of cryptographic protocols. Feasible computation and the communication between multiple agents can be expressed in the language itself, allowing for a thorough analysis in one single framework. Our main example is the famous Diffie-Hellman protocol for secret key distribution over an insecure network.
For both systems we implement model checking in Haskell. For ECL we also provide a Monte Carlo algorithm that gives probabilistic results but runs much faster than the ordinary implementation.
All source code is included as part of the text. The main features and design choices are highlighted with annotations.

**Acknowledgements**

First of all, I want to thank Jan van Eijck for sharing the idea of register languages with me and supervising this thesis. Without our many meetings and his continuous motivation it would not exist.
Alexandru Baltag helped me at two different times: I enjoyed his enthusiastic Introduction to Modal Logic during my first term and received very helpful feedback after a presentation of this work.
My gratitude also goes to Maria Aloni for being the chair of my committee, Joshua Sack for pointing out a crucial mistake and Johan van Benthem for challenging questions.
I also want to thank Christian Schaffner for his great Introduction to Modern Cryptography and joining my defense committe last-minute.
I am indebted to the authors of [BDV01] for writing my favorite logic textbook and Yde Venema in particular for enlightening courses and supervision of an individual research project.
Once again, Pauline Fleischmann deserves my gratitude for reading and commenting on a draft of this thesis. I also want to thank Michael Schmidt, Jan Schejbal and Marten Seemann who provided helpful comments.
Furthermore, I am grateful to all the people who enriched my two years both in Science Park and the rest of Amsterdam: Swantje, Evante, Simon, Nadine, Laura, Adriana and last not least Emma.
Finally, I want to thank Anne, Hans and Pia Gattinger for all kinds of support someone in my situation could ever ask for.

# Contents

# Introduction

What does it mean to know a number?
And why is this an important question?
In this thesis we will relate the question to something we use every day, often without knowing about it, namely cryptography: When we send and receive emails, text messages or money, when we make phone calls, when we use credit cards, when we get tickets for public transport, and so on.
Cryptography consists of secret communication and correct computation. While the correctness of the latter is usually easy to check within a mathematical framework, proving that it also leads to the secrecy of the former often poses a challenge. This creates a need for reliable frameworks of reasoning about cryptography – in the best case yielding formal proofs that certain assumptions suffice to guarantee secrecy and authenticity.
Our main focus here is what different agents know and what they communicate. We first formalize an answer to the basic question what it means to know a number. The *Guessing Game Logic* should already make the idea both apprehensible and precise. Then we extend this formalism to a more expressive formal language that describes multi-agent knowledge, communication and computation in detail. This *Epistemic Crypto Logic* then allows us to analyze cryptographic protocols and attacks on them.
Our aim is to contribute to different fields and research communities at the same time:

1. Model checking is a big and established research area but only recently has started to specifically look at epistemic logics. Our work combines several ideas for domain-specific optimization, for example representing relations as partitions and encoding large models as small models with registers.

2. Logical and philosophical debates about knowledge, belief and epistemic reasoning often revolve around rather artificial or extremely simplified examples. There is nothing wrong with card games and muddy children, but we think that cryptographic schemes can also provide very interesting and relevant examples to evaluate different theories and logics of knowledge.

3. Finally, as mentioned above, the verification of cryptography is both in the interest of researchers and the general public.

The thesis is structured as follows: In the first chapter we introduce the necessary basics of modal logic, dynamic epistemic logic and product updates. Readers who are already familiar with these concepts might want to skip this and jump directly to our original work in the following chapters.
The second chapter introduces the idea of register models to capture the knowledge of a number and presents the logic of guessing games $\mathsf{GG}$. We give definitions of syntax and

semantics, a sound and complete axiomatization and an annotated implementation in Haskell.
In the third chapter we first discuss which expressive powers a logic for the analysis of cryptographic protocols should have. We then show how to extend GG to Epistemic Crypto Logic (ECL) by enriching both the syntax and the semantics in various ways. Again we also provide all syntactic and semantic definitions and an implementation in Haskell.
In Chapter four it is time to harvest: We discuss several small examples and then show how the Diffie-Hellman key exchange can be represented and checked in our framework. We also sketch how our language can be used to analyze attacks on cryptographic protocols.
Finally, in chapter five we conclude what was achieved and provide a multitude of ideas for further research in different directions.

We use literate programming in the spirit of [Knu84]. Source code of all implementations is presented as part of the text and available at `w4eg.de/malvin/illc/thesis`.

ALICE SENDS A MESSAGE TO BOB SAYING TO MEET HER SOMEWHERE.
UH HUH.
BUT EVE SEES IT, TOO, AND GOES TO THE PLACE.
WITH YOU SO FAR.
BOB IS DELAYED, AND ALICE AND EVE MEET.
YEAH?
I'VE DISCOVERED A WAY TO GET COMPUTER SCIENTISTS TO LISTEN TO ANY BORING STORY.
CC-BY-NC `http://xkcd.com/1323`

# Chapter 1

# Foundations

In this chapter we introduce the foundations of our work, namely the basic definitions for dynamic epistemic logic. Since the exact definitions vary from paper to paper and by now there is a plethora of epistemic logics, we will make sure to provide all the definitions needed in the later chapters. Still, we assume that the reader is familiar with propositional and first-order logic, formal definitions of truth/satisfaction and set-theoretic notation. Symbols and abbreviations are also listed on page 83.
Our introduction is systematic rather than historical. For a history of the development of dynamic epistemic logic we refer to the introductory chapter in [VVK07]. Readers who are already familiar with the main concepts might want to skip this chapter and jump directly to our original work starting in the next chapter.

## 1.1 From Modal Logic to Knowledge

Building on top of propositional logic, modal logics introduce new connectives to the language in order to formalize notions of modality. The most-studied modalities are *possibility* and *necessity*, but modal logic in the broader sense deals with various concepts like *provability*, temporal relations like *before* or epistemic modalities like *belief* and *knowledge*.[1]

**Definition 1.** *We write* $\mathbf{P}$ *for a countable infinite set of propositions and denote the elements with* $p$, $q$, $r$, $p_1$, $p_2$, $p_3$ *etc. The basic modal language* ($\mathcal{L}_\Diamond$) *over* $\mathbf{P}$ *is given by the following Backus-Naur Form (BNF):*

$$\phi \ ::= \ \top \mid p \mid \neg\phi \mid \phi \wedge \phi \mid \Diamond\phi$$

*This definition says that a formula can be constructed in one of five ways: The true constant* $\top$*, an atomic proposition, a negation of a formula, a conjunction of two formulas or a single formula prefixed with the symbol* $\Diamond$ *which we call diamond and read as "it is possible that". Moreover we define the following abbreviations:*

$$\begin{array}{rcl@{\qquad\qquad}rcl} \bot & := & \neg\top & \phi \vee \psi & := & \neg(\neg\phi \wedge \neg\psi) \\ \Box\phi & := & \neg\Diamond\neg\phi & \phi \to \psi & := & \neg(\phi \wedge \neg\psi) \end{array}$$

[1]See [Gar13] for an overview of different modal logics from a philosophical perspective.

Throughout this text we take the notion of a proposition as given. Our syntactical definitions of languages are independent of what propositions actually are and the same goes for the interpretation in models where our valuations at each possible world contain a set of propositions. As long as the same $\mathbf{P}$ is used in both semantic and syntactic definitions we can leave concerns about what propositions are aside.

### 1.1.1 Kripke Frames and Models

Kripke frames and models are the most common structures used to define semantics for modal logics. They consist of i) a set of possible worlds which are also called states and despite their stately name have no further structure themselves and ii) a relation on this set that says which other worlds are reachable (also: accessible). Graphically this can be represented with simple dots and arrows.

**Definition 2.** *A* Kripke frame *for basic modal logic is a tuple $\mathcal{F} = (W, R)$ where $W$ is a set of* worlds *or* states *and $R$ is a relation on $W$ (i.e. $R \subseteq W \times W$) which is also called* accessibility relation. *Elements of $W$ are usually called $w$, $v$, $s$, $t$ etc. We write $wRv$ to say that the relation $R$ holds between $w$ and $v$. In a set-theoretical framework this just means $(w, v) \in R$.*

**Definition 3.** *A* Kripke model *for basic modal logic is a Kripke frame for basic modal logic $\mathcal{F}$ together with a valuation function $V : W \to \mathcal{P}(\mathbf{P})$. We write it as $\mathcal{M} = (W, R, V)$ and say that $\mathcal{M}$ is* based *on $\mathcal{F}$. Furthermore, we also refer to the elements of $W$ as the* worlds or states of the model $\mathcal{M}$. *A* pointed model *is a model $\mathcal{M}$ together with one of its worlds that is marked as the actual world $w$. We write pointed models as $\mathcal{M}, w$.*

We define the meaning of a modal formula like $\Diamond p$ which could for example stand for "It is *possible* that $p$" by referring to the relational structure of the model. The usual definition stipulates that $\Diamond\phi$ is true at a world $w$ if there is a so-called reachable world $v$ such that $wRv$ and $\phi$ is true at $v$. Besides this condition, we also include the usual semantics for boolean connectives in the next definition.

**Definition 4.** *The semantics for $\mathcal{L}_\Diamond$ are given by:*

$$\begin{array}{lcllll}
\mathcal{M}, w & \vDash & \top & :\iff & \textit{always} \\
\mathcal{M}, w & \vDash & p & :\iff & p \in V(w) \\
\mathcal{M}, w & \vDash & \neg\phi & :\iff & \textit{not } \mathcal{M}, w \vDash \phi \\
\mathcal{M}, w & \vDash & \phi \wedge \psi & :\iff & \mathcal{M}, w \vDash \phi \textit{ and } \mathcal{M}, w \vDash \psi \\
\mathcal{M}, w & \vDash & \Diamond\phi & :\iff & \exists v : wRv \textit{ and } \mathcal{M}, v \vDash \phi
\end{array}$$

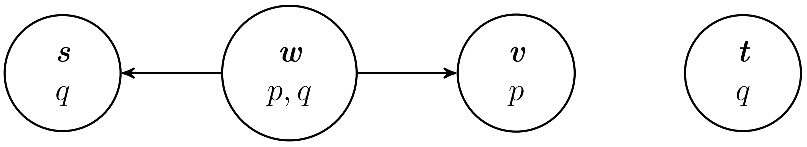
We usually assume that "necessary" is the boolean dual of "possible", i.e. that something is necessarily true if it is not possible that it is false. This is reflected by the fact that according to Definition 1 $\Box\phi$ is just an abbreviation for $\neg\Diamond\neg\phi$. Alternatively, we could give semantics for $\Box$ directly with the following definition:

$$\mathcal{M}, w \vDash \Box\phi :\iff \forall v : \text{If } wRv \text{ then } \mathcal{M}, v \vDash \phi$$

In fact we would have to do so if we dealt with non-classical (e.g. intuitionistic or minimal) logics, but throughout this text we will always take our underlying propositional logic to be classical.
Given the semantics we can now define truth and validity of $\mathcal{L}_\Diamond$ formulas in general.

**Definition 5** (Truth and Validity)**.** *We say that $\phi$ is* true at $w$ in $\mathcal{M}$ *iff $\mathcal{M}, w \vDash \phi$. We say that $\phi$ is* true in $\mathcal{M}$ *iff it is true at all worlds in $\mathcal{M}$. We say that $\phi$ is* valid on a frame $\mathcal{F}$ *iff it is true in all models based on $\mathcal{F}$. We say that $\phi$ is* valid on a class of frames $\mathfrak{F}$ *iff it is valid on all $\mathcal{F} \in \mathfrak{F}$. We say that $\phi$ is* valid *and write $\vDash \phi$ iff it is valid on the class of all frames.*

**Example 6.** *Consider a Kripke model $\mathcal{M} = (W, R)$ given by the set $W = \{w, v, s, t\}$ and the relation $R = \{(w, s), (w, v)\}$. To include the valuation into the figure we use circles instead of dots and write a proposition $p$ below the name of a world iff this proposition is true there. The relation $R$ is represented by the arrows between the circles. In this case from $w$ the worlds $s$ and $v$ are reachable but no other connection is given. In particular $R$ is irreflexive and no world can reach itself.*



*We can see that $\Diamond p$ is true at $w$ because from there we can reach the world $v$ where $p$ is true. Formally: $\mathcal{M}, w \vDash \Diamond p$. However we do* not *have $\mathcal{M}, w \vDash \Box p$ because we can also reach the world $s$ where $p$ is false.*

An important property of modal logic which we can already observe here is *locality*: While the truth of a formula at a certain world *can* depend on the facts in other worlds, it suffices to look only at reachable worlds. In the example above we do not have to look at the world $t$ to evaluate a statement at $w$. We will make the idea that only a certain part of a model "matters" precise when we discuss generated submodels in section 1.4.2.
The inconspicuous definitions for box $\Box$ and diamond $\Diamond$ are surprisingly powerful and sparked a lot of research from mathematical, logical and philosophical perspectives. One of the most interesting research areas is *correspondence theory* which tries to find which properties of frames can be described with a modal formula and conversely which modal formulas describe a property of frames.

**Definition 7** (Correspondence)**.** *We say that a formula $\phi$* corresponds *to a frame property $P$ (which is expressible in first-order logic) iff $\phi$ is valid on every frame that has the property $P$ and vice versa any frame on which $\phi$ is valid also has the property $P$.*

Before giving some examples of correspondence we define important properties and the reflexive transitive closure of a relation. The latter is crucial for the notion of common knowledge which we introduce in Section 1.1.3.

**Definition 8.** *A relation $R$ on a set $W$ is* reflexive *iff $\forall s \in W : sRs$,* symmetric *iff $\forall s, t : sRt \to tRs$,* transitive *iff $\forall s, t, u : sRt \wedge tRu \to sRu$ and* euclidean *iff $\forall s, t, u : sRt \wedge sRu \to tRu$. A relation is called an equivalence relation iff it is transitive, symmetric and reflexive. The* reflexive transitive closure *$R^*$ of a relation $R$ is the least set such that $R \subseteq R^*$ and that $R^*$ is reflexive and transitive.*

**Theorem 9.** *If a relation is reflexive and euclidean, then it is an equivalence relation.*

*Proof.* Suppose $R$ is reflexive and euclidean. We have to show that $R$ is symmetric and transitive. Symmetry: Suppose $xRy$. By reflexivity we also have $xRx$. Hence, by euclideanness we also have $yRx$. Transitivity: Suppose $xRy$ and $yRz$. By symmetry we also have $yRx$. Hence, by euclideanness we have $xRz$. $\Box$

**Theorem 10.** *The following are examples for correspondences between modal formulas and frame properties. The duality between $\Box$ and $\Diamond$ yields two versions for each of the formulas. Still, note that those are in general not equivalent in the strong sense of having the same truth conditions but only regarding their* validity *on frames, i.e. one of them is valid on a frame iff the other one is.*

- $p \to \Diamond p$ *and* $\Box p \to p$ *correspond to reflexivity.*
- $p \to \Box\Diamond p$ *and* $\Diamond\Box p \to p$ *correspond to symmetry.*
- $\Diamond\Diamond p \to \Diamond p$ *and* $\Box p \to \Box\Box p$ *correspond to transitivity.*
- $\Diamond p \to \Box\Diamond p$ *and* $\Diamond\Box p \to \Box p$ *correspond to euclideanness.*

**Definition 11** (The logic $\mathsf{K}$)**.** *The logic* $\mathsf{K}$ *is given by the following axiomatization.*

- *If* $\phi$ *is a propositional tautology, then* $\vdash \phi$*.*
- *Modus Ponens: If* $\vdash \phi$ *and* $\vdash \phi \to \psi$*, then* $\vdash \psi$*.*
- *Distributivity: If* $\vdash \Box(\phi \to \psi)$*, then* $\vdash \Box\phi \to \Box\psi$*.*
- *Necessitation: If* $\vdash \phi$*, then* $\vdash \Box\phi$*.*

Correspondence results allow us to find axiomatizations that are sound and complete for various classes of frames by adding different axioms to $\mathsf{K}$.

**Definition 12** (The logic $\mathsf{S5}$)**.** *The system* $\mathsf{S5}$ *(named after the two additional axioms) is obtained by adding the following two axioms to* $\mathsf{K}$*.*

- $\vdash \Box\phi \to \phi$
- $\vdash \Diamond\phi \to \Box\Diamond\phi$

**Theorem 13.** *The axiomatization of* $\mathsf{S5}$ *is sound and complete for the class of Kripke frames based on reflexive and euclidean (and therefore: equivalence) relations.*

It should be noted however, that a modal logic given by some list of axioms does not always have to be sound and complete with regard to a class of Kripke frames. The first examples for such logics in the basic modal language were presented in [Fin74] and [Tho74]. In fact it its not easy to decide which formulas characterize a class of frames and which do not. The most famous advances in this area are the Sahlqvist correspondence results in [Sah75] and generalizations thereof.

### 1.1.2 Knowledge

Instead of possibility and necessity we can also use Kripke structures to model knowledge about propositions. To do so we take the box-modality, now written as $K$, as the fundamental one and define what it means to know something as follows.

**Definition 14.** *The language of the modal logic of knowledge $\mathcal{L}_K$ is given by this BNF:*

$$\phi \quad ::= \quad \top \mid p \mid \neg\phi \mid \phi \wedge \phi \mid K\phi$$

*and the usual abbreviations for $\rightarrow$ and $\vee$. The semantics of $\mathcal{L}_K$ are the same as those of $\mathcal{L}_\Diamond$ in Definition 4 but with the clause for $\Diamond$ replaced by:*

$$\mathcal{M}, w \vDash K\phi : \Longleftrightarrow \forall v \in W : wRv \rightarrow v \vDash \phi$$

*Namely, $\phi$ is known at world $w$ iff $\phi$ is true in all worlds that an be reached from $w$.*

Intuitively we can think of reachable worlds as those which are deemed possible or probable while non-reachable (or even non-existing) worlds are those which an agent does not even consider. Again note the locality of modal logic: An agent might know something at a certain world but not at another in the same model.
The results of correspondence theory which we sketched above now become relevant for the philosophical and logical debate which properties knowledge has. Knowledge is truthful. Hence, we would always want the formula $K\phi \rightarrow \phi$ to be valid which corresponds to reflexivity. Two other important properties which are discussed by many authors (see [Sch14] for an overview) are about meta-knowledge, namely positive and negative introspection. Positive introspection means that whenever something is known, then it also is known that it is known. In a formula: $K\phi \rightarrow KK\phi$. As we have seen above this formula corresponds to transitivity of $R$. Negative introspection is concerned with what is not known and usually a bit more controversial: It means that whenever something is not known, then it is known that this is not known. Again we can write this as a formula $\neg K\phi \rightarrow K\neg K\phi$ and observe that this formula corresponds to is euclideanness.
We have shown above (Theorem 9) that reflexivity and euclideanness together imply that $R$ is an equivalence relation, i.e. it is also symmetric and reflexive. Thus any Kripke semantics approach to formalize knowledge with truthfulness and negative introspection will use equivalence relations and any complete proof system will incorporate the validities of $\mathsf{S5}$, including positive introspection.
The widespread usage of $\mathsf{S5}$ led to the slogan "Knowledge is an equivalence relation." which might intuitively make sense but should also be used carefully. Our model surely contains an equivalence relation, but we should remind ourselves that it is a model and its properties are not necessarily also properties of knowledge as it is given by the situation or observation we are modeling. In fact, something like being-an-equivalence-relation can make the distinction of modeled and model very clear: Already asking the question "Is knowledge an equivalence relation?" only makes sense within our modeling because the concept of an equivalence relation is only defined in a formal framework. In contrast, an informal description of positive introspection like "I know what I know" is a statement about knowledge itself, not our models. It is by far not trivial but rather a substantial *claim* that $K\phi \rightarrow KK\phi$ captures positive introspection.
In everything that follows we leave aside this philosophical debate and always represent knowledge with equivalence relations. Our main motivation is that a strong notion of knowledge fits to the phenomena that we intend to cover here. Furthermore, equivalence relations are also easier to handle in an actual implementation as we discuss in Section 1.6. However, this choice is not necessary for model checking per se. We leave it for further research to find suitable representations and efficient methods for other classes of models.

### 1.1.3 Multiple Agents and Common Knowledge

The next generalization of our models is to represent not just the knowledge of a single agent but a whole set of agents $I$.

**Definition 15.** *A multi-agent Kripke frame for a set of agents $I$ is a pair $(W, \mathcal{R})$ where $W$ is a set of worlds and $\mathcal{R}$ is a family of relations $\mathcal{R} = (R_i)_{i \in I}$ on $W$.*

**Definition 16.** *A* multi-agent Kripke model *is a multi-agent Kripke frame together with a valuation $V : W \to \mathcal{P}(\mathbf{P})$. We write $\mathcal{M} = (W, \mathcal{R}, V)$.*
*We then include the following clause into our semantics for every agent $i \in I$:*

$$\mathcal{M}, w \vDash K_i \phi :\iff \forall v \in W : wR_iv \to v \vDash \phi$$

That is, an agent knows something iff it is true in all worlds which this agent considers possible according to her very own accessibility relation.
A formula like $K_bp$ should now be read as "Bob knows that $p$". The indexed modalities also allows us to express sentences like "Alice knows that Bob knows that $p$" by nesting the connectives of different agents: $K_a(K_jp)$. Furthermore, if we are only concerned with the three agents, say Alice, Bob and Charlie, then we can formalize "everybody knows that $p$" as $K_ap \wedge K_bp \wedge K_cp$.
One of the strongest notions in epistemic logics is *common knowledge* for which we introduce the modality $C$. This modality is supposed to be even stronger than "everybody knows". If $\phi$ is common knowledge then everyone knows that $\phi$ and everyone knows that everyone knows that $\phi$. And everyone knows that everyone knows that everyone knows $\phi$. And so on.
Formally we can define $C\phi$, the common knowledge of $\phi$, in two ways. On the semantic level we can interpret $C$ just like the $K$ modalities but with respect to the relation $(\cup_{i \in I} R_i)^*$, i.e. the transitive reflexive closure of the union of all agents' relations. A syntactic approach on the other hand would define $C$ as an abbreviation for the conjunction of any number of nested knowledge modalities. Note that this conjunction would be an infinite sentence, thus $C$ still should be added to the language as an additional connective and not as an abbreviation in order to keep the language finite. Fortunately, we can easily show that the two ways to define common knowledge are equivalent.

**Definition 17** (Semantic Common Knowledge)**.**

$$\mathcal{M}, w \vDash C\phi :\iff (\cup_{i \in I} R_i)^*$$

**Theorem 18** (Syntactic Common Knowledge)**.**

$$\mathcal{M}, w \vDash C\phi \iff \textit{for any } n \in \mathbb{N} \textit{ and any } (i_1, \ldots, i_n) \in I^n\textit{:}\ \mathcal{M}, w \vDash K_{i_1} \ldots K_{i_n}\phi$$

*Proof.* Left to right: Suppose $\mathcal{M}, w \vDash C\phi$. Then by definition $\forall v : w(\cup_{i \in I} R_i)^* v \to v \vDash \phi$. Now fix any $n \in \mathbb{N}$ and $(i_1, \ldots, i_n) \in I^n$. By $(R_{i_1} \circ \cdots \circ R_{i_n}) \subseteq (\cup_{i \in I} R_i)^*$ we now have in particular $\forall v : w(R_{i_1} \circ \cdots \circ R_{i_n})v \to v \vDash \phi$ which implies $\mathcal{M}, w \vDash K_{i_1} \ldots K_{i_n}\phi$.
Right to left, by contraposition: Suppose $\mathcal{M}, w \nvDash C\phi$. Then by definition there is a $v$ such that $w(\cup_{i \in I} R_i)^* v$ and $v \nvDash \phi$. Therefore we have an $n \in \mathbb{N}$ and $i_1, \ldots, i_n \in I$ such that $w(R_{i_1} \circ \cdots \circ R_{i_n})v$. Hence, by the semantics from Definition 16 we have $\mathcal{M}, w \nvDash K_{i_1} \ldots K_{i_n}\phi$. □

There are many more notions of group knowledge which can be formalized in Kripke semantics. A detailed discussion can be found in [VVK07, pp. 30-38].

## 1.2 Public Announcements

So far our epistemic logic is static: Given a certain model, our agents either know something or not. We also want to capture the truth conditions of statements about the change of knowledge due to new information becoming available. Examples are "If it was announced that it rains every Tuesday, Bob would know that it will rain tomorrow." and again also stacked modalities like "If it was revealed that Bob does not know that it rains, Alice would know that the curtain is closed." A well-studied framework dealing with such sentences is the logic of public announcements, revolving around the next definition. Intuitively, a public announcement of $\phi$ removes all possible worlds where $\phi$ is false. The following Definition 19 gives meaning to a new unary connective $[!\phi]$ which says that a statement is true after a truthful public announcement of $\phi$.

**Definition 19** (Public Announcement)**.** *The result of the public announcement $!\phi$ in the model $\mathcal{M} = (W, \mathcal{R}, V)$ is the model $\mathcal{M}^{!\phi} = (W', \mathcal{R}', V')$ where $W' := \{w \in W \mid \mathcal{M}, w \vDash \phi\}$, $R'_i := R_i \cap (W' \times W')$ and $V' := V \cap (W' \times W')$. Given this definition we add the dynamic modality $!\phi$ to our language and give it the following semantics:*

$$\mathcal{M}, w \vDash [!\phi]\psi :\iff \mathcal{M}, w \vDash \phi \textit{ implies } \mathcal{M}^{!\phi}, w \vDash \psi$$

Here "implies" is a material implication. Hence the operator $[!\phi]$ should be thought of as a box rather than a diamond: $[!\phi]\psi$ means that after every truthful public announcement that $\phi$ is true, $\psi$ would be true. If $\phi$ is false, then it can never be truthfully announced and this holds trivially. The restriction to truthful announcements is needed, because after the announcement we want to evaluate $\psi$ at the same world. This is only possible if it survives the announcement. The fact that $\psi$ is evaluated in a different model nicely reflects the use of conditional sentences in our two examples above. We can now formalize them as $[!p]K_b q$ and $[!\neg K_b p]K_a q$ respectively.
All these announcements are *public* because removing worlds is an update that does not differentiate between agents. Hence, Definition 19 can only model situations where everyone can hear the announcement and should be distinguished from private or group announcements like "If Bob (but not anyone else) was told ...".
Another important and somewhat surprising property of public announcements is that they do not always generate common knowledge. That is, only for some formulas $\phi$ we have that $[!\phi]C\phi$ is valid. For counterexamples and a detailed discussion of the so-called successful fragment, see [VVK07, p. 84].

## 1.3 Product Update

### 1.3.1 Epistemic Change

The public announcements discussed in the previous section are not the only events which we want to analyze in our multi-agent models. Consider for example that Alice gets to know a secret but Bob and Charlie do not. And suppose Alice then tells this secret to Bob while Charlie is not listening. Both of these events are obviously not public announcements. In this and the next section we will define *actions* as which we can represent such events correctly.
The framework of action structures in [BMS98] provides a canonical way to model almost arbitrary actions and events, partly inspired by the update semantics developed in [Vel96].

What matters about an event in the context of epistemic logic is first *what* happens and second *who knows about it*. This motivates the essential idea of action structures: We describe actions themselves as Kripke structures, very similar to the static situations before and afterwards.

**Definition 20** (Actions)**.** *An* action structure *is a tuple* $(A, \mathcal{R})$ *where* $A$ *is a set of so-called* action tokens*,* $\mathcal{R} = (R_i)_{i \in I}$ *is a family of equivalence relations on* $A$ *and furthermore for any* $\alpha \in A$ *we have a formula* $\mathsf{pre}(\alpha)$ *which is called the* precondition *of* $a$*.*
*An* action *is a triple* $(A, \mathcal{R}, \alpha)$ *where* $(A, \mathcal{R})$ *is an action structure and* $\alpha \in A$*. We say that* $\alpha$ *is* actually happening *and often write just* $\alpha$ *to refer to the action as a whole..*

Note that we demand action structures to be based on equivalence relations. This is also done in [BMS98] but not necessary for product updates in general.
Now we can define how to update a model with an action. We implicitly assume that model and action are defined for the same set of agents.

**Definition 21** (Product Update)**.** *The* product update *given by an action* $(A, \mathcal{R}, \alpha)$ *is a partial function that maps pointed models to pointed models and is defined as follows:*

$$\begin{array}{rcl} (W, \mathcal{R}, V), w & \mapsto & (W', \mathcal{R}', V'), (w, \alpha) \textit{ where} \\ & & W' := \{(w, \alpha) \in W \times A \mid w \vDash \mathsf{pre}(\alpha)\} \\ & & (w, \alpha) R'_i (v, \beta) :\iff wR_iv \textit{ and } \alpha R_i \beta \\ & & V'(w, \alpha) := V(w) \end{array}$$

*We write* $\mathcal{M}^\alpha$ *for the result of updating* $\mathcal{M}$ *with the action* $\alpha$*.*

That is, an agent will confuse two worlds in the resulting model iff she could not distinguish the two worlds before and could not distinguish the two different actions.
It is important to remark that the model before and after the product update are not of the same type, namely our worlds are now pairs of worlds and actions. While this is not a problem for our formal definitions (where $W$ can be any set), in our implementation we rewrite a model after the product update into one where the states are of the same type.

**Example 22.** *The public announcement of* $\phi$ *is given by the action* $(\{\alpha\}, \mathcal{R}, \alpha)$ *where* $\mathsf{pre}(\alpha) = \phi$ *and* $\forall i \in A : R_i := \{(\alpha, \alpha)\}$*. It is easy to check that this action and definition 19 are equivalent. Note in particular that only* $\phi$*-worlds survive the action and the relation is the same for all agents.*

### 1.3.2 Factual Change

Action structures so far allow us to model quite a few different events, but they are restricted to *epistemic change*. This means they can change what agents know but not what is actually the case. Yet, an announcement like "I am now thinking of a secret number..." does not merely tell other agents that they do not know this number. Instead it introduces the variable itself, bringing something completely new into the conversation. We want to model this as the *creation* of a variable.
Therefore, we also need a way to model *factual change*. This is done by adding valuation changes to the action structures and can be done in various ways. In [BEK06] factual change is represented by substitutions of formulas for atomic sentences. In this framework

the substitution $\sigma := \{p \mapsto q, q \mapsto p\}$ for example swaps the truth values of $p$ and $q$. We get that $(p \vee q) \to [\sigma](p \vee q)$ is a validity but $(p \to q) \to [\sigma](p \to q)$ is not.
The models we will discuss here have more complex valuations, i.e. for each world the valuation is the usual set of true propositions plus additional information, for example which values some variable can take or who is listening. Actions should also be able to change these parts of the valuation. We will thus not represent factual change as substitutions but as functions that map valuations to valuations. This allows us to define the new valuation after an update as the sequential execution of the previous valuation function and the change function. For any given kind of valuation we define actions with factual change as follows.

**Definition 23** (Actions with factual change)**.** *An* action structure with factual change *is a tuple $(A, \mathcal{R})$ where $A$ is a set of so-called* action tokens, *$\mathcal{R} = (R_i)_{i \in I}$ is a family of equivalence relations on $A$ and furthermore for any $\alpha \in A$ we have a formula $\mathsf{pre}(\alpha)$, called the* precondition *of $\alpha$, and a function $\mathsf{change}_\alpha$ which maps valuations to valuations.*
*An* action with factual change *is a triple $(A, \mathcal{R}, \alpha)$ where $(A, \mathcal{R})$ is an action structure with factual change and $\alpha$ is an element $\alpha \in A$. Again we also write just $\alpha$ to refer to the action as a whole and say that $\alpha$ is* actually happening.

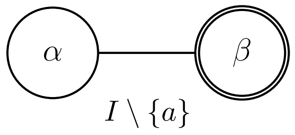
**Definition 24** (Product Update with factual change)**.** *The* product update with factual change *given by an action with factual change $(A, \mathcal{R}, \alpha)$ is a function that maps models to models and is defined as follows.*

$$\begin{array}{rcl} (W, \mathcal{R}, V), w & \mapsto & (W', \mathcal{R}', V'), (w, \alpha) \textit{ where} \\ & & W' := \{(w, \alpha) \in W \times A \mid w \vDash \mathsf{pre}(\alpha)\} \\ & & (w, \alpha) R'_i (v, \beta) :\iff wR_iv \textit{ and } \alpha R_i \beta \\ & & V'(w, \alpha) := (\mathsf{change}_\alpha \circ V)(w) \end{array}$$

*Again we write $\mathcal{M}^\alpha$ for the result of updating $\mathcal{M}$ with the action $\alpha$.*

**Example 25.** *Consider models with valuation functions that map worlds to sets of propositions $X \subseteq \mathbf{P}$. Let $(\{\alpha, \beta\}, \mathcal{R}, \beta)$ be an action defined as follows. Note that we leave out the reflexive arrows in the drawing and see Definition 33 for the notation of the relations as partitions.*

$$\begin{array}{ll} \mathsf{pre}(\alpha) := \top, & \mathsf{change}_\alpha := \mathsf{id}, \\ \mathsf{pre}(\beta) := \top, & \mathsf{change}_\beta(X) := \begin{cases} X \setminus \{p\} & \textit{if } p \in X \\ X \cup \{p\} & \textit{otherwise} \end{cases} \\ \textit{and} & R_i := \begin{cases} \{\alpha \mid \beta\} & \textit{if } i = a \\ \{\alpha\beta\} & \textit{otherwise} \end{cases} \end{array}$$



*Then $\beta$ changes the truth value of $p$ and tells agent $a$ about the change. All other agents will not know the truth value of $p$ any longer, i.e. the formula $[\beta]\neg K_i p$ is valid for all $i \neq a$. But note that $a$ does not necessarily know the truth value of $p$ after the update. In fact the formula $(K_a p \vee K_a \neg p)$ is invariant under $\beta$.*

Given this definition and example, one might wonder if our approach is more or less general than the original in [BEK06] and the answer is indeed nontrivial. While mapping valuations to valuations on first sight seems more powerful than a substitution of formulas, it is in fact less expressive. Example 26 shows that substitutions are more expressive because via epistemic formulas they can also refer to the valuation somewhere else.

**Example 26** (Expressive power of substitution actions)**.** *Consider the action "Hey Bob, if you know whether p is the case, please wave!" and let q be a proposition that represents if Bob is waving. Assuming that Bob follows the order, we can formalize this action as the substitution* $\{q \mapsto (K_b p \vee K_b \neg p)\}$*. But it can not be represented as a map between valuations in general: The new valuation at a world does not only depend on the old valuation at this world but – because it is an epistemic statement – also depends on the truth values of p at other worlds in the old model. When defining the action as a function on valuations, these would not be available.*

For the other direction, namely to see that all our actions could also be represented as substitutions, first note that for valuations of the type $X \subseteq \mathbf{P}$ our maps are just substitutions of boolean formulas for propositions. For complex valuations, we can argue that they are just encodings for "normal" boolean valuations in bigger models: The local listener set $L_w = \{a\}$ that we introduce in Definition 66 could also be viewed as just another way to write down a valuation that makes the atomic sentence $L_a$ true and for all $i \neq a$ the atomic sentence $L_i$ false at $w$. Therefore, also a change function working on valuations with such listener sets can be represented as a substitution of boolean formulas for some of the $L_i$s. Similar arguments can be given for the registers and constraint sets we introduce in Definitions 44 and 66. Hence, our representation of factual changes as valuation-maps just restricts the approach in [BMS98, BEK06] to boolean combinations of non-epistemic statements and we can still rely on the results presented there.
There is a lot more to say about actions, in particular about their status as syntactic or semantic objects and the general language with operators for every possible action model. As the definitions above suffice for our purposes, we just refer to [VVK07, Chapter 6].

## 1.4 Smaller Equivalent Models

Many actions increase the size of our models because they create multiple copies of worlds. Especially after multiple updates we can end up in huge models that are hard to analyze. Do we have to deal with these huge models? The answer is: no. Two well-established notions from modal logic will be helpful here, namely bisimulation and generated submodels. These ideas provide a way to find equivalent models in the sense that they satisfy exactly the same formulas as the original models. Moreover, if models are redundant, these methods can give us smaller models that are easier to handle. We will use them in Section 3.6.2 to increase the efficiency of our implementation.

### 1.4.1 Bisimulation

**Definition 27** (Bisimulation)**.** *A non-empty relation* $Z \subseteq W \times W'$ *is called a* bisimulation *between* $\mathcal{M}_1 = (W, \mathcal{R}, V)$ *and* $\mathcal{M}_2 = (W', \mathcal{R}', V')$ *iff*

*(i) If* $wZw'$ *then* $V(w) = V'(w')$*.*

*(ii) If* $wZw'$ *and* $wR_iv$*, then there is a* $v' \in W'$ *such that* $vZv'$ *and* $w'R'_iv'$*.*

*(iii) If* $wZw'$ *and* $w'R'_iv'$*, then there is a* $v \in W$ *such that* $vZv'$ *and* $wR_iv$*.*

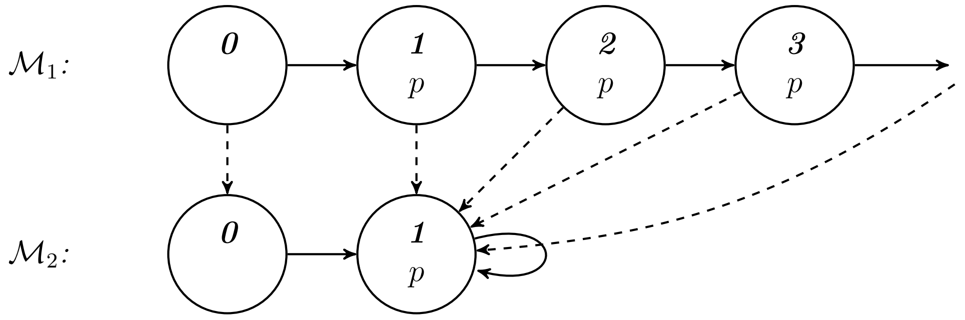
*If such a Z exists with* $w_1Zw_2$ *we say that* $\mathcal{M}_1, w_1$ *is bisimilar to* $\mathcal{M}_2, w_2$ *and use the binary relation symbol* $\underline{\leftrightarrow}$ *as in* $\mathcal{M}_1, w_1 \underline{\leftrightarrow} \mathcal{M}_2, w_2$*.*

The following example of two models for normal modal logic shows that sometimes very large or even infinite models can be bisimilar to small ones.

**Example 28.** *Let $\mathcal{M}_1 = (\mathbb{N}, <, f)$ and $\mathcal{M}_2 := (2, \{(0,1),(1,1)\}, f_{\restriction 2})$ where*

$$f(n) := \begin{cases} \varnothing & \text{if } n = 0 \\ \{p\} & \text{otherwise} \end{cases}$$

*Then we can easily check that $Z := \{(0,0)\} \cup \{(k,1) \mid 0 \neq k \in \mathbb{N}\}$, drawn below with dashed arrows, is a bisimulation and for example $\mathcal{M}_1, 0 \underline{\leftrightarrow} \mathcal{M}_2, 0$.*



Bisimulation is a most interesting notion for modal logics in general because of the following theorem which also holds for many modal logics without global modalities.

**Theorem 29.** *If $\mathcal{M}_1, w \underline{\leftrightarrow} \mathcal{M}_2, w_2$, then for all $\phi$ we have $\mathcal{M}_1, w_1 \vDash \phi$ iff $\mathcal{M}_2, w_2 \vDash \phi$.*

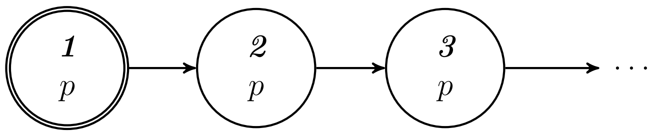
*Proof.* By induction on the complexity of $\phi$. See [BDV01, Theorem 2.20, p. 67]. □

The concept of bisimulation comes with a useful complexity result: Minimal bisimilar models can be obtained efficiently and [Eij14] provides a generic implementation which we will employ in Section 3.6.2

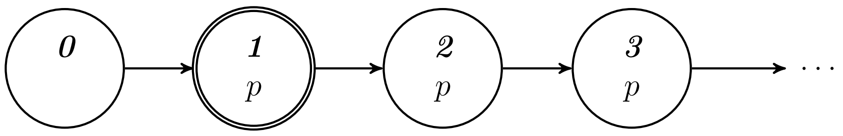
## 1.4.2 Generated Submodels

Again we consider two models which are similar in some sense.

**Example 30.** *Let $\mathcal{M} = (\mathbb{N} \setminus \{0\}, <, f)$ where $f(n) := \begin{cases} \varnothing & \text{if } n = 0 \\ \{p\} & \text{otherwise} \end{cases}$.*



*Let $\mathcal{M}' = (\mathbb{N}, <, f)$.*



*Note that all worlds in $\mathcal{M}$ can be identified with worlds in $\mathcal{M}'$. Moreover the $\mathcal{M}'$-pendants of all worlds of $\mathcal{M}$ can only reach worlds which are also in $\mathcal{M}$. In this situation we call $\mathcal{M}$ a generated submodel according to the following definition.*

**Definition 31.** *Given two models $\mathcal{M} = (W, \mathcal{R}, V)$ and $\mathcal{M}' = (W', \mathcal{R}', V')$ We say that $\mathcal{M}$ is a* generated submodel *of $\mathcal{M}'$ iff $W \subseteq W'$, $\mathcal{R}' = \mathcal{R} \cap (W \times W)$ and $V' = V_{\restriction W}$.*

The expected and useful result about generated submodels is that truth of modal formulas is preserved to generated submodels. Again, a warning should be made that this does not hold for modal languages with the global modality.

**Theorem 32.** *If $\mathcal{M}$ is a generated submodel of $\mathcal{M}'$ and $w$ a world in $\mathcal{M}$ then for all $\phi$ we have $\mathcal{M}, w \vDash \phi$ iff $\mathcal{M}', w \vDash \phi$.*

*Proof.* By induction on $\phi$. Alternatively and as noted in [BDV01, Proposition 2.6, p. 56], it suffices to show that generated submodels induce a bisimulation to the original model. □

## 1.5 Functional Programming

For our implementations we use the functional programming language *Haskell* which is particularly suited for logical and mathematical programming for several reasons.

- Functional style fits our purpose much better than imperative. Examples of the latter are *C++* and *Python*. Where in those languages we "tell" the computer what to *do*, in Haskell we rather *define* the intended result.
- The Syntax of Haskell is often close to mathematical notation. Two examples should illustrate this point. First, consider the set of even natural numbers. Here is how a definition in a text book might look like and how to define it in Haskell:

  $$E := \{x * 2 \mid x \in \mathbb{N}\}$$

  ```
  e = [ x*2 | x <- [0..]  ]
  ```

  Second, the characteristic function of $\{1,5\}$ as we know it and in Haskell:

  $$f(x) := \begin{cases} 1 & \text{if } x \in \{1,5\} \\ 0 & \text{otherwise} \end{cases}$$

  ```
  f x = if (elem x [1,5]) then 1 else 0
  ```

- Haskell is statically typed which means that everything our programs are dealing with has a type. Examples of types are integers, strings and lists. Also our concepts of propositions, agents, models and formulas will be represented as types.
- The `ghc` compiler we are using offers "lazy" evaluation. This means that functions are only called and computations only made if they are actually needed. This fits nicely to the locality of modal logic: Whatever might occur in a structure but does not matter for the evaluation of a formula can be ignored.

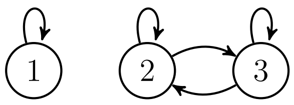
We will not provide an introduction to Haskell here but refer to existing literature. A book which nicely teaches both foundations of logic and their implementation in Haskell at the same time is [Dv04].
There also is a plethora of good online resources which is summarized at

`www.haskell.org/haskellwiki/Learning_Haskell`.

One can also try out simple examples in a browser on `www.tryhaskell.org`.

## 1.6 Relations as Partitions

Equivalence relations are isomorphic to partitions and very often mathematicians like to identify these isomorphic structures or just call them two different ways to refer to the same object. But from a computational perspective it matters a lot of which type our objects are because a concise data structure can be read and modified faster than a more complicated structure with unnecessary redundancy.
For example, consider a simple S5 frame and two different representations in Haskell:



```
relation  = [ (1,1), (2,2), (2,3), (3,2), (3,3) ]
partition = [ [1], [2,3] ]
```

We can see that the representation as a partition is much more concise. As we are only concerned with equivalence relations throughout this thesis, we can use it in our implementations. For a detailed discussion of relations as partitions, see [Eij14].
To keep our formal definitions succinct as well we also define the following notation for equivalence relations as partitions.

**Definition 33** (Equivalence Relations as Partitions)**.** *To simplify the definition of equivalence relations, we write partitions like*

$$\alpha_{1,1} \dots \alpha_{1,n_1} \mid \dots \mid \alpha_{m,1} \dots \alpha_{m,n_m}$$

*to denote the corresponding equivalence relation given in standard set-theoretic notation:*

$$\{(\alpha_{k,i}, \alpha_{k,j}) \mid k \leq m \textit{ and } i, j \leq n_m\}$$

**Example 34.** *A partition of three elements into two classes of size one and two expands to an equivalence relation with five elements:*

$$\alpha \mid \beta\gamma \textit{ corresponds to } \{(\alpha,\alpha),(\beta,\beta),(\beta,\gamma),(\gamma,\beta),(\gamma,\gamma)\}$$

# Chapter 2

# Guessing Games

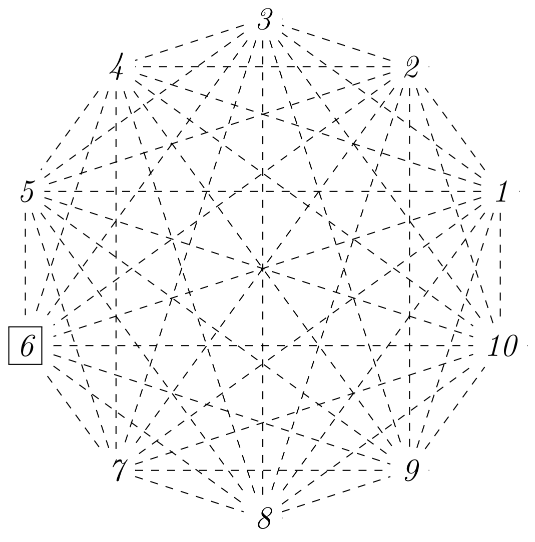
## 2.1 The Number Guessing Game

### 2.1.1 What does it mean to know a number?

To explore this question we consider the following number guessing game, played between Jan and his twin daughters Gaia and Rosa.

JAN: "I have a number in mind, in the range from one to ten. You may take turns guessing. Whoever guesses the number first wins."
GAIA: "I love this game!"
ROSA: "Me too, can I guess first?"
JAN: "Okay, go ahead."

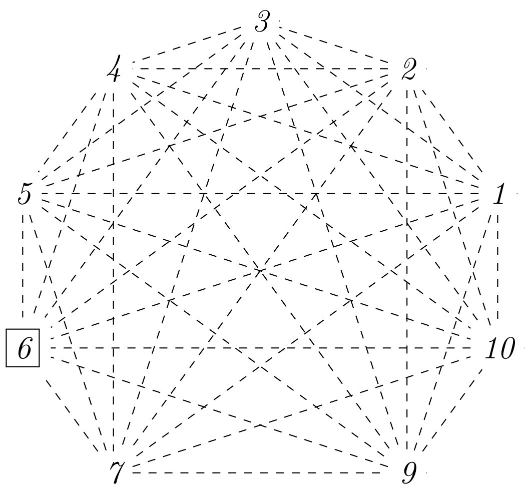
**Example 35.** *A naive representation of this game can be given as a multi-agent Kripke model with ten worlds. The actual world – where the number happens to be* 6 *– is indicated by a box. At the start we have a total graph which represents the ignorance of the twins.*



**Example 36.** *Now suppose the following exchange takes place:*

ROSA: *"Eight?"*
JAN: *"No."*

*This results in an update of the model: The possibility* 8 *drops out, and this is common knowledge among the twins because they both hear Jans reply:*



And so on. But as the twins get older they refuse to play the game like this and a slight change of rules is necessary to regain their trust:

JAN: "I have a number in mind, in the range from one to ten. You may take turns guessing. Whoever guesses the number first gets the toy. It is Gaia's turn to start, for last time we played this game Rosa started the guessing."

GAIA: "But how can we know you are not cheating on us? Please write down the number before we start guessing, so you can show it to us afterwards as a proof."

JAN: "Okay." [*Jan writes a number on a piece of paper, hidden from Gaia and Rosa.*]

GAIA: "Is it five?"

JAN: "No."

ROSA: "Is it six?"

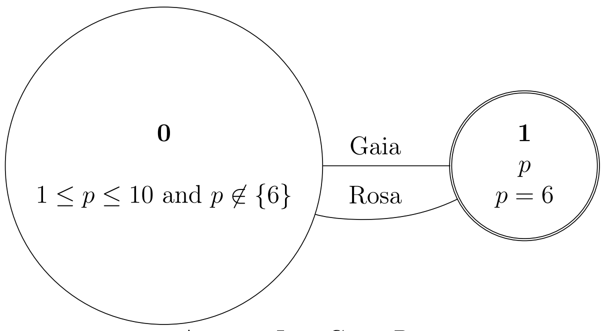
JAN: "Yes, Rosa. You have won." [*Jan shows the piece of paper with a* 6 *on it as a proof.*]

At the point where the twins demand that the secret number gets *written down*, and that Jan shows it as a proof that he was honest, the important notion of a *register* arises. The register allows Jan to prove that he really knew the number and that he had fixed it before the guessing started. In particular, he did not just accept Rosa's guess because he wanted her to win.

So what is it that Jan knew when we say that he knew the number? Let us say: Jan can see the difference between a register with *the correct number* written in it, and the same register with *some different number* written on it. If someone else would change the number on the piece of paper during the game, Jan would realize it, but Gaia and Rosa would not.

We can now see that to represent the game in a Kripke model it is in fact enough to have two possible worlds and such a register. In the actual world the register $n$ has the value 6 and in the other world it can be anything else in the range which was agreed upon earlier. Jan knows $n$, the two children do not know $n$. This leads to the following *register model* in which we indicate the actual world by a double circle.

**Example 37** (Register Model)**.** *In this model the real value of $p$ is $6$ which we can see in the actual world **1** where $p$ is true. In **0** however, $p$ is false and the value can be anything else, but not $6$. Furthermore, the connections for Gaia and Rosa tell us that they do not have access to the register and do not know the value of $p$.*



Agents: Jan, Gaia, Rosa

The way we will talk about such register models yields a clash of notation and speech when it comes to "the value of $p$". We have to distinguish the *truth value* of $p$ and the *numeric value* assigned to $p$. However, it will usually be clear from the context what we mean and we will not introduce separate notation.

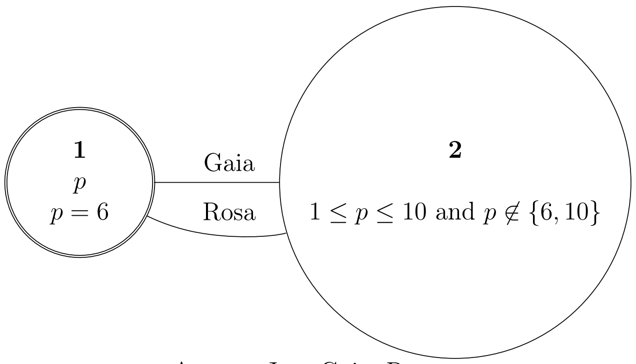
### 2.1.2 Announcements

Suppose now Gaia makes a guess.

GAIA: "Ten?"
JAN: "No."

Because Rosa is also present, this is a *public announcement* and the following example makes precise what happens.

**Example 38** (Announcing Negative Information)**.** *If the register model from example 37 is updated with the announcement $!p \neq 10$ everyone will know that the value of $p$ is not $10$. The resulting register model reflects this:*



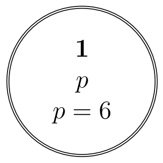
Agents: Jan, Gaia, Rosa

*The only change that has taken place is that in the world where $p$ is false we added the restriction that $p$ may not be $10$.*

**Example 39.** *When the true value is guessed we only have one world left.*

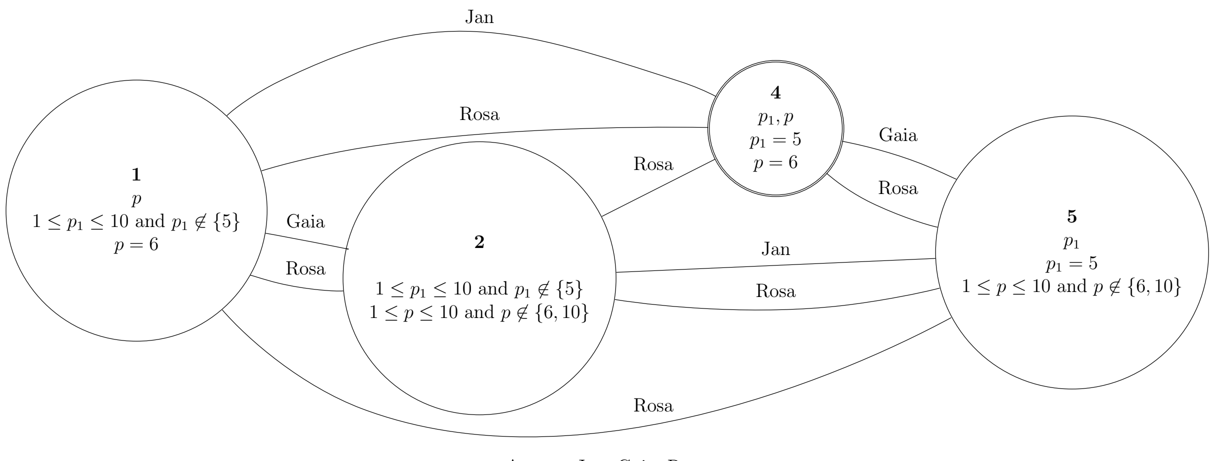
ROSA: *"Is it six?"*

JAN: *"Yes, you guessed it!"*



Agents: Jan, Gaia, Rosa

**Example 40.** *We can also represent the moment when Gaia prepares to announce a guess but has not yet revealed it to the others. We write $p_1$ for the guess of Gaia. She knows her guess, but Jan and Rosa do not know it yet:*



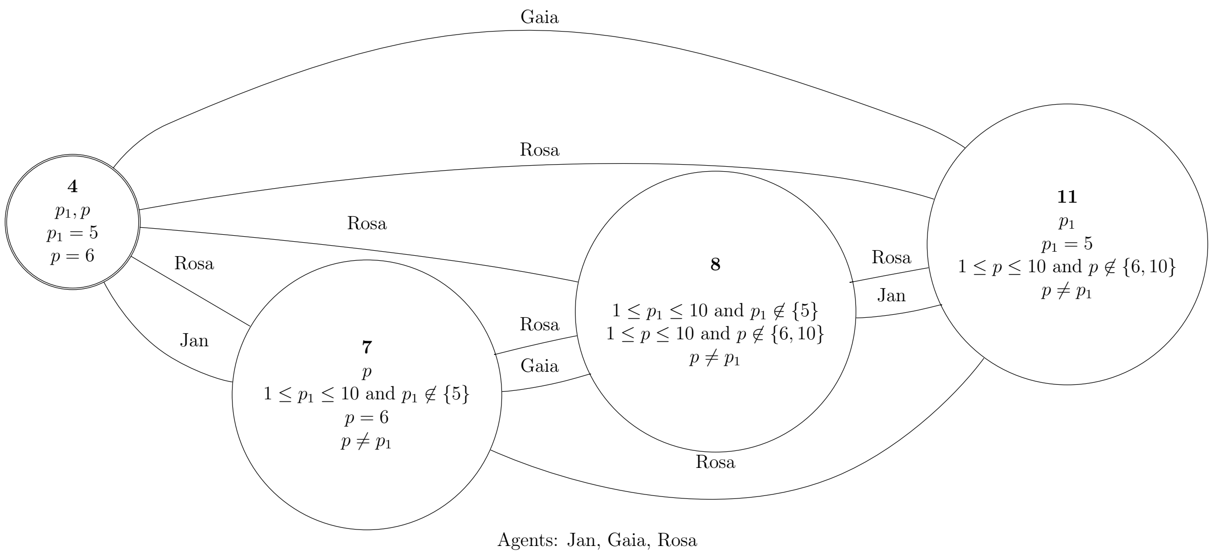
Jan
Rosa
1
$p$
$1 \leq p_1 \leq 10$ and $p_1 \notin \{5\}$
$p = 6$
Gaia
Rosa
2
$1 \leq p_1 \leq 10$ and $p_1 \notin \{5\}$
$1 \leq p \leq 10$ and $p \notin \{6, 10\}$
Rosa
4
$p_1, p$
$p_1 = 5$
$p = 6$
Gaia
Rosa
Jan
Rosa
5
$p_1$
$p_1 = 5$
$1 \leq p \leq 10$ and $p \notin \{6, 10\}$
Rosa

Agents: Jan, Gaia, Rosa

*In particular we can see that Rosa knows the least because she is confusing all worlds with each other while Jan and Gaia both know something, namely $p$ and $p_1$ respectively.*

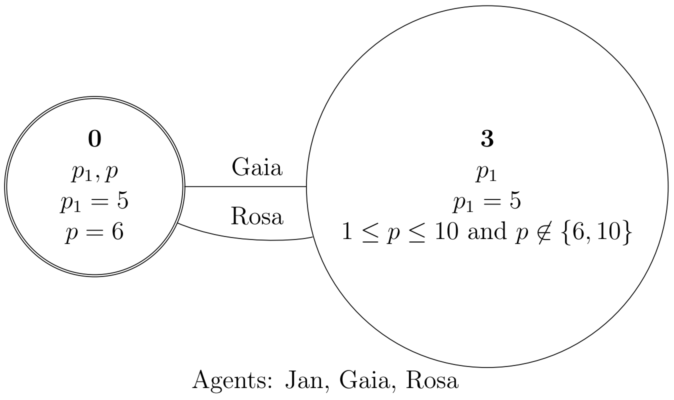
This model might already seem a bit complicated but note that without registers, it would blow up to a much bigger model in which we would have 100 worlds. More generally, if we want to model a situation with $k$ many variables each of which can have a value from 1 to $m$, then we would get a "naive" Kripke frame with $m^k$ possible worlds while our new representation only needs $2^k$ many worlds.

**Example 41.** *Suppose now that Gaia tells someone else about her guess who also knows Jan's secret Number. And suppose then this person announces (truthfully) to everyone that Gaia's next guess will be wrong. This means that $p \neq p_1$ is announced in the situation* before *Gaia has revealed her guess. The model of Example 40 changes into the following:*


Gaia
Rosa
Rosa
Rosa
Jan
4
$p_1, p$
$p_1 = 5$
$p = 6$
11
$p_1$
$p_1 = 5$
$1 \leq p \leq 10$ and $p \notin \{6, 10\}$
$p \neq p_1$
8
$1 \leq p_1 \leq 10$ and $p_1 \notin \{5\}$
$1 \leq p \leq 10$ and $p \notin \{6, 10\}$
$p \neq p_1$
Rosa
Jan
Rosa
Gaia
7
$p$
$1 \leq p_1 \leq 10$ and $p_1 \notin \{5\}$
$p = 6$
$p \neq p_1$
Rosa
Agents: Jan, Gaia, Rosa

*Note that Rosa still neither knows the value of $p$ nor $p_1$ but she does know that $p \neq p_1$.*

**Example 42.** *If Gaia reveals the contents of her register by means of the announcement $p_1 = 5$, the model of Example 40 changes into the following.*



Jan can now state that the guess is wrong, by means of the announcement $n \neq g$. The result is a model that is just like above, except for the fact at world **5** that $p \notin \{6, 5, 10\}$ (the possibility $p = 5$ has dropped out).

## 2.2 Syntax

Registers are like names for numbers, and the referential puzzles that are well known from the philosophy of language [Fre92, Qui60, Kri72] reappear in the present context. Just compare $p = q$ with "Hesperus is Phosphorus", and see that creating a register and writing a number in it can be viewed as a *baptism*, just like "Let's call this bright object in the sky that is visible just after sundown Hesperus." Register equality statements are like name equality statements. The number equality statement $n = n$ can be compared to "Hesperus is Hesperus.", and the truth of such trivial equality statements should be common knowledge in any register model.

**Definition 43** (Register Language for Guessing Games)**.** *Let $p$ range over the set of propositions $\mathbf{P}$, let $N$ range over $\mathbb{N}$ and let $i$ range over the finite set of agents $I$. The register language for guessing games $\mathcal{L}_{\mathsf{GG}}$ consists of the following formulas, commands and expressions.*

$$\begin{aligned} \phi &::= \top \mid p \mid p = E \mid \neg\phi \mid \phi \wedge \phi \mid K_i\phi \mid G\phi \mid \langle C \rangle \phi \\ C &::= \,!p = E \mid !p \neq E \mid p \overset{i}{\leftarrow} N \mid A; A \\ E &::= p \mid N \end{aligned}$$

*Again we also define some abbreviations. Let $\bot := \neg\top$. For any $\phi$ and any command $C$ we define $[C]\phi := \neg\langle C\rangle\neg\phi$ and for any $\phi$ and $\psi$ we let $\phi \vee \psi := \neg(\neg\phi \wedge \neg\psi)$, $\phi \to \psi := \neg(\neg\phi \wedge \psi)$ and $\phi \leftrightarrow \psi := (\phi \to \psi) \wedge (\psi \to \phi)$. For any $p$ and $E$, let $p \neq E := \neg(p = E)$. Furthermore, we define the epistemic diamond by $\widehat{K_i}\phi := \neg K_i \neg\phi$.*

A new element in this language is the unary connective $G$ which we will interpret as the global modality. It states that $\phi$ is true at all worlds in our model, i.e. its interpretation is always the total relation. The global modality is particularly useful to find reduction axioms for announcements and register creation; see Theorem 56 for details.
We purposely do not introduce common knowledge (based on the the reflexive-transitive closure of the union of all epistemic relations) here, because it complicates the axiomatization. Concretely, the reduction schemes P6, N6 and R8 in Theorem 56 are no longer valid if $G$ is the common knowledge operator. Still, note that we do interpret $G$ as common knowledge in the extended framework of $\mathsf{ECL}$ in Chapter 3.

## 2.3 Semantics

**Definition 44** (Guessing Game Models)**.** *A* guessing game model *for $I$ and $\mathbf{P}$ is a tuple $\mathcal{M} = (W, \mathcal{R}, V)$ where*

- *$(W, \mathcal{R})$ is a multi-agent $\mathsf{S5}$ frame for $I$ according to Definition 15,*
- *$V$ is a valuation function for some $Q \subseteq \mathbf{P}$ (the global set of used variables): It assigns to each world $w \in W$ a valuation which is a tuple $(P_w, f_w, C_w^+, C_w^-)$ where*
  - *$P_w \subseteq \mathbf{P}$ (the basic propositions true at $w$),*
  - *$f_w$ is a function on $Q$ that assigns to each $q \in Q$ a triple $(n, m, X)$ with $n, m \in \mathbb{Z}$, $n \leq m$, $X \subseteq \mathbb{Z}$, satisfying two constraints:*
    - *(i) whenever $q \in P_w$ then for $f_w(q) = (n, m, X)$ we have $n = m$ and $X = \varnothing$*
    - *(ii) whenever $p \in P_v \cap P_w$ for $v, w \in W$ then $f_v(p) = f_w(p)$,*
  - *$C_w^+$ is a subset of $Q^2$ (the equality constraints of $w$)*
  - *$C_w^-$ is another subset of $Q^2$ (the inequality constraints of $w$) which is consistent with $C_w^+$ according to definition 45 below.*

*We refer to parts of guessing game models in various ways:*

- *In cases where it is clear or does not matter which world we are talking about we leave out $w$ and just write valuations as $(P, f, C^+, C^-)$.*

- *The triple $f_w(q)$ to which we also refer by $(f_w^0(q), f_w^1(q), f_w^2(q))$ is called the* range of $q$ in $w$*. Its elements are the lower bound $n$, upper bound $m$, and a list of excluded values $X$. Translated to standard notation, $f_w(p) = (n, m, X)$ means that the set of possible values for $p$ at the world $p$ is $\{x \in \mathbb{N} \mid x \notin X \wedge n \leq x \leq m\}$.*

- *The sets $C_w^+$ and $C_w^-$ are the positive and negative equality constraints at $w$, in the following sense: $(p, q) \in C_w^+$ expresses that $p$ and $q$ have the same values at $w$ and $(p, q) \in C_w^-$ expresses that $p$ and $q$ have different values at $w$.*

**Definition 45** (Consistency of Constraints)**.** *An equality constraint set $C^+$ is* consistent with *an inequality constraint set $C^-$ iff there is no $(p, q) \in C^-$ which is also in the transitive symmetric reflexive closure of $C^+$ on $\mathbf{P}$.*

**Example 46.** *The constraint sets $C^+ = \{(p, q), (r, s)\}$ and $C^- = \{(p, r)\}$ are consistent. In contrast, $C^+ = \{(p, q), (q, r)\}$ and $C^- = \{(p, r)\}$ are not.*

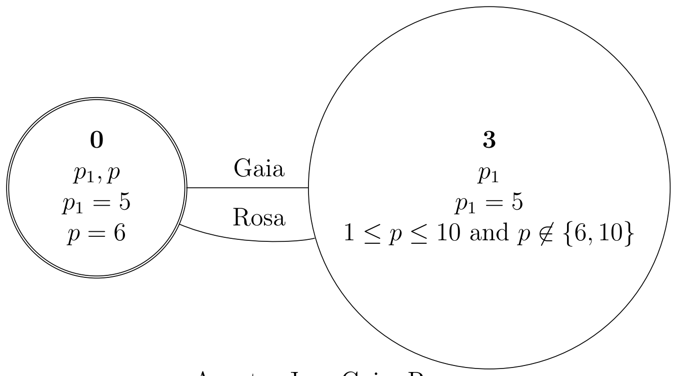
**Definition 47.** *An* assignment *is a function that maps propositions to integers. An assignment $h$* agrees *with a world $w$ (notation $w \multimap h$ or $h \multimapinv w$) iff*

- *for all $q \in Q$: $f_w^0(q) \leq h(q) \leq f_w^1(q)$ and $h(q) \notin f_w^2(q)$,*

- *$h$ satisfies the positive constraints $C_w^+$: if $(p, q) \in C_w^+$ then $h(p) = h(q)$ and*

- *$h$ satisfies the negative constraints $C_w^-$: if $(p, q) \in C_w^-$ then $h(p) \neq h(q)$.*

**Example 48** (Agreement and disagreement)**.** *Again consider the following two-world model where everyone knows $p_1$ but only Jan knows $p$.*



Agents: Jan, Gaia, Rosa

*According to Definition 47 we have that:*

- *The function $h = \{p \mapsto 6, p_1 \mapsto 5\}$ agrees with **0**, but not with **3**.*

- *The function $h' = \{p \mapsto 3, p_1 \mapsto 5\}$ agrees with **3** but not with **0**.*

**Definition 49** (Interpretation of $\mathcal{L}_{\mathsf{GG}}$ in Register Models)**.** *We define the satisfaction relation $\mathcal{M}, w, h \models \phi$. Let $\mathcal{M} = (W, \mathcal{R}, V, A)$ be a register model, $w \in W$ and $h$ an assignment that agrees with $w$. Inductively we define for all $\phi \in \mathcal{L}_{\mathsf{GG}}$:*

$$\begin{array}{rll}
\mathcal{M}, w, h \models \top & & \textit{always} \\
\mathcal{M}, w, h \models p & \textit{iff} & p \in P_w \\
\mathcal{M}, w, h \models p_1 = p_2 & \textit{iff} & h(p_1) = h(p_2) \\
\mathcal{M}, w, h \models p = N & \textit{iff} & h(p) = N \\
\mathcal{M}, w, h \models \neg\phi & \textit{iff} & \textit{not } \mathcal{M}, w, h \models \phi \\
\mathcal{M}, w, h \models \phi_1 \wedge \phi_2 & \textit{iff} & \mathcal{M}, w, h \models \phi_1 \textit{ and } \mathcal{M}, w, h \models \phi_2 \\
\mathcal{M}, w, h \models K_i\phi & \textit{iff} & wR_iw' \textit{ implies that for all } h' \multimap w' : \mathcal{M}, w', h' \models \phi \\
\mathcal{M}, w, h \models G\phi & \textit{iff} & \textit{for all } w' \in W \textit{ and all } h' \multimap w' : \mathcal{M}, w', h' \models \phi \\
\mathcal{M}, w, h \models \langle !\, p = E\rangle\phi & \textit{iff} & \mathcal{M}, w, h \models p = E \textit{ and } \mathcal{M}^{!p=E}, (w, \alpha), h \models \phi \\
\mathcal{M}, w, h \models \langle !\, p \neq E\rangle\phi & \textit{iff} & \mathcal{M}, w, h \models p \neq E \textit{ and } \mathcal{M}^{!p\neq E}, (w, \alpha), h \models \phi \\
\mathcal{M}, w, h \models \langle p \overset{i}{\leftarrow} N\rangle\phi & \textit{iff} & \mathcal{M}, w, h \models G\neg p \textit{ and } \mathcal{M}^{p\overset{i}{\leftarrow}N}, (w, \alpha), h \cup \{(p, N)\} \models \phi \\
\mathcal{M}, w, h \models \langle A_1; A_2\rangle\phi & \textit{iff} & \mathcal{M}, w, h \models \langle A_1\rangle\langle A_2\rangle\phi
\end{array}$$

*where the models $\mathcal{M}^{!p=E}$, $\mathcal{M}^{!p\neq E}$ and $\mathcal{M}^{p\overset{i}{\leftarrow}N}$ are given by actions with factual change as introduced in Definition 23 and specified in the following three definitions.*

**Definition 50** ($\mathcal{M}^{!p=E}$)**.** *This update represents "You guessed it" and reveals positive information. The update is only defined for truthful announcements, i.e. if $p = E$ is true at the actual world. It is given by the following two action structures, depending on whether $E$ is a number or another variable.*
*If $E$ is a number, $\mathcal{M}^{!p=N}$ is the result of restricting $\mathcal{M}$ to the worlds where $p = N$ is true:*

$$\begin{array}{lll}
(\{\alpha\}, \mathcal{R}, \alpha) & \textit{where} & \mathsf{pre}(\alpha) := (p = N) \\
& & \mathsf{change}_\alpha := \mathsf{id} \\
& & R_i := \alpha \textit{ for all } i \in I
\end{array}$$

*If $E$ is another variable, $\mathcal{M}^{!p=q}$ is obtained by restricting $\mathcal{M}$ to the worlds where $p$ and $q$ have the same truth value. Furthermore, in worlds where $p$ and $q$ are both false the constraint $(p, q)$ is added to $C^+(w)$.*

$$\begin{array}{lll}
(\{\alpha, \beta\}, \mathcal{R}, \alpha) & \textit{where} & \mathsf{pre}(\alpha) := p \wedge q \wedge (p = q) \\
& & \mathsf{pre}(\beta) := \neg p \wedge \neg q \\
& & \mathsf{change}_\alpha := \mathsf{id} \\
& & \mathsf{change}_\beta(P, f, C^+, C^-) := (P, f, C^+ \cup \{(p, q)\}, C^-) \\
& & R_i := \alpha\beta \textit{ for all } i \in I
\end{array}$$

**Definition 51** ($\mathcal{M}^{!p\neq E}$)**.** *This represents "Your guess was wrong" and reveals negative information. It is only defined for truthful announcements, i.e. $p \neq E$ has to be true at the actual world. Again we give two action structures for the cases whether $E$ is a number or variable.*

*If $E$ is a number, $\mathcal{M}^{!p \neq N}$ is the result of adding $N$ to the list of excluded values of $p$ at all worlds where $p$ is false:*

$$(\{\alpha,\beta\},\mathcal{R},\alpha) \quad \textit{where} \quad \begin{array}{l} \mathsf{pre}(\alpha) := p \wedge (p \neq N) \\ \mathsf{pre}(\beta) := \neg p \\ \mathsf{change}_\alpha := \mathsf{id} \\ \mathsf{change}_\beta(P,f,C^+,C^-) := (P,\mathsf{new}(f),C^+,C^-) \\ \mathsf{new}(f)(q) := \begin{cases} (f^0(p), f^1(p), f^2(p) \cup \{N\}) & \textit{if } q = p \\ f(q) & \textit{otherwise} \end{cases} \\ R_i := \alpha\beta \textit{ for all } i \in I \end{array}$$

*If $E$ is a variable $q$, then $\mathcal{M}^{!p \neq q}$ is the result of adding the constraint $(p,q)$ to $C^-(w)$ for every world where $p \wedge q$ is false:*

$$(\{\alpha,\beta\},\mathcal{R},\alpha) \quad \textit{where} \quad \begin{array}{l} \mathsf{pre}(\alpha) := p \wedge q \wedge (p \neq q) \\ \mathsf{pre}(\beta) := \neg(p \wedge q) \\ \mathsf{change}_\alpha := \mathsf{id} \\ \mathsf{change}_\beta(P,f,C^+,C^-) := (P,f,C^+,C^- \cup \{(p,q)\}) \\ R_i := \alpha\beta \textit{ for all } i \in I \end{array}$$

The creation of new registers ("I am thinking of a number ...") is represented by the command $p \overset{a}{\leftarrow} N$. This links $p$ to the number $N$ with the link known only to agent $a$. Formally, also this is given by an action structure. Besides the proposition it also depends on $\mathsf{regsize}$, the globally fixed maximum value any variable can take.
The precondition for register creation is $G\neg p$ and makes sure that $p$ has not been introduced as a register already because then it would have to be true somewhere.

**Definition 52** (Register Creation)**.** *For any model $\mathcal{M}$ the model $\mathcal{M}^{p \overset{i}{\leftarrow} N}$ is given by the action structure $(\{\alpha,\beta\},\mathcal{R},\alpha)$ to $\mathcal{M}$ where*

$$\begin{array}{rcl} \mathsf{pre}(\alpha) & := & G\neg p \\ \mathsf{pre}(\beta) & := & G\neg p \\ \mathsf{change}_\alpha(P,f,C^+,C^-) & := & (P \cup \{p\}, f \cup \{(p,(N,N,\varnothing))\}, C^+, C^-) \\ \mathsf{change}_\beta(P,f,C^+,C^-) & := & (P, f \cup \{(p,(0,\mathsf{regsize},\{N\}))\}, C^+, C^-) \\ R_i & := & \begin{cases} \alpha \mid \beta & \textit{if } i = a \\ \alpha\beta & \textit{otherwise} \end{cases} \end{array}$$

We can now lift the notion of truth with regard to assignments to truth and falsity at a world by saying that a formula is true at a world iff it is true with regard to all agreeing assignments. It is false iff it is false with regard to all agreeing assignments.

**Definition 53** (Truth at a world, Validity)**.**

$$\mathcal{M}, w \models \phi \quad \textit{iff} \quad \forall h \textit{ with } w \multimap h : \mathcal{M}, w, h \models \phi.$$

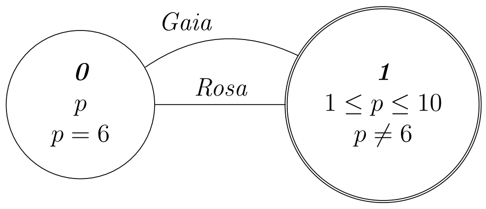
*A formula $\phi$ is* valid *iff for all $\mathcal{M}$ and all $w$ we have $\mathcal{M}, w \vDash \phi$. We then write $\vDash \phi$.*

Note that we could additionally introduce a notion of falsification:

$$\mathcal{M}, w \dashv \phi \quad \text{iff} \quad \forall h \text{ with } w \multimap h : \mathcal{M}, w, h \not\models \phi.$$

Then formulas could be undecided in the sense that $\mathcal{M}, w \models \phi$ and $\mathcal{M}, w \dashv \phi$ can both be false – see Example 54. This will be reflected in our implementation where we use the data type `Maybe Bool`. However, for the notion of *validity* this does not matter and our logic is still classical, e.g. the law of excluded middle is valid.

**Example 54** (An undecided formula)**.** *In the following model, $p = 7$ is false in $0$, but neither true nor false in $1$. Similarly, $p \neq 7$ is true in $0$, but neither true nor false in $1$.*



This illustrates that the definitions of $\mathcal{M}, w \models \phi$ and $\mathcal{M}, w \mathrel{=\!\!|} \phi$ create truth value gaps. However, $K_i$ closes these truth value gaps again. This is because $K_i\phi$ does not depend on the given assignment and becomes false in case that $\phi$ is undecided in a reachable world.

**Theorem 55.** *For all $\mathcal{M}$, $w$, $i$ and $\phi$ we have either $\mathcal{M}, w \models K_i\phi$ or $\mathcal{M}, w \mathrel{=\!\!|} K_i\phi$.*

*Proof.* It suffices to observe the following equivalences.

$$\begin{array}{ll} & \mathcal{M}, w \mathrel{=\!\!|} K_i\phi \\ \text{iff} & \forall h \text{ with } w \multimap h : \mathcal{M}, w, h \models \neg K_i\phi \\ \text{iff} & \forall h \text{ with } w \multimap h \exists w', h' \text{ with } wR_iw', w' \multimap h' \text{ and } \mathcal{M}, w', h' \models \neg\phi \\ \text{iff} & \exists h \text{ with } w \multimap h \exists w', h' \text{ with } wR_iw', w' \multimap h' \text{ and } \mathcal{M}, w', h' \models \neg\phi \\ \text{iff} & \exists h \text{ with } w \multimap h \text{ and } \mathcal{M}, w, h \not\models K_i\phi \\ \text{iff} & \mathcal{M}, w \not\models K_i\phi \end{array}$$

□

## 2.4 Axiomatization

We will now present a proof system for the register language for guessing games. We first provide reduction axioms for all three commands. These are inspired by [BMS98] and [BEK06]. We then note that all reduction axioms are valid and thereby enable us to find equivalent command-free formulas for any given formula. Finally, we add our reduction axioms to a standard axiomatization of S5 for multiple agents with a global modality and thus obtain a sound and complete system for the logic of guessing games.

**Theorem 56.** *The following reduction schemes are valid.*
*Positive announcements:*

*P0)* $\langle !p = E\rangle\top \leftrightarrow (p = E)$

*P1)* $\langle !p = E\rangle q \leftrightarrow (p = E \wedge q)$

*P2)* $\langle !p = E\rangle(q = E') \leftrightarrow (q = E')$

*P3)* $\langle !p = E\rangle\neg\phi \leftrightarrow (p = E \wedge \neg\langle !p = E\rangle\phi)$

*P4)* $\langle !p = E\rangle(\phi \wedge \psi) \leftrightarrow (\langle !p = E\rangle\phi \wedge \langle !p = E\rangle\psi)$

*P5)* $\langle !p = E\rangle\widehat{K_i}\phi \leftrightarrow (p = E \wedge \widehat{K_i}(\langle !p = E\rangle\phi))$

*P6)* $\langle !p = E\rangle G\phi \leftrightarrow (p = E \wedge G(p = E \rightarrow \langle !p = E\rangle\phi))$

*Negative announcements:*

*N0)* $\langle !p \neq E\rangle\top \leftrightarrow \neg(p = E)$

*N1)* $\langle !p \neq E\rangle q \leftrightarrow (\neg(p = E) \wedge q)$

*N2)* $\langle !p \neq E\rangle(q = E') \leftrightarrow (\neg(p = E) \wedge (q = E'))$

*N3)* $\langle !p \neq E\rangle\neg\phi \leftrightarrow (\neg(p = E) \wedge \neg\langle !p \neq E\rangle\phi)$

*N4)* $\langle !p \neq E\rangle(\phi \wedge \psi) \leftrightarrow (\langle !p \neq E\rangle\phi \wedge \langle !p \neq E\rangle\psi)$

*N5)* $\langle !p \neq E\rangle\widehat{K_i}\phi \leftrightarrow (\neg(p = E) \wedge \widehat{K_i}(\langle !p \neq E\rangle\phi))$

*N6)* $\langle !p \neq E\rangle G\phi \leftrightarrow (\neg(p = E) \wedge G(\neg(p = E) \rightarrow \langle !p \neq E\rangle\phi))$

*Register creation:*

*R0)* $\langle p \overset{i}{\leftarrow} N\rangle\top \leftrightarrow (G\neg p)$

*R1)* $\langle p \overset{i}{\leftarrow} N\rangle p \leftrightarrow (G\neg p)$

*R2)* $\langle p \overset{i}{\leftarrow} N\rangle q \leftrightarrow (G\neg p \wedge q)$ *where* $p \neq q$

*R3) For equality statements we consider a few subcases:*

*R3a1)* $\langle p \overset{i}{\leftarrow} N\rangle(p = N) \leftrightarrow (G\neg p)$

*R3a1')* $\langle p \overset{i}{\leftarrow} N\rangle(p = M) \leftrightarrow \bot$ *where* $M \neq N$

*R3a2)* $\langle p \overset{i}{\leftarrow} N\rangle(q = M) \leftrightarrow (G\neg p \wedge (q = M))$ *where* $p \neq q$

*R3b1)* $\langle p \overset{i}{\leftarrow} N\rangle(p = p) \leftrightarrow (G\neg p)$

*R3b1')* $\langle p \overset{i}{\leftarrow} N\rangle(p = q) \leftrightarrow (G\neg p \wedge (q = N))$ *where* $p \neq q$

*R3b2)* $\langle p \overset{i}{\leftarrow} N\rangle(q = p) \leftrightarrow (G\neg p \wedge (q = N))$ *where* $p \neq q$

*R3b2')* $\langle p \overset{i}{\leftarrow} N\rangle(q = r) \leftrightarrow (G\neg p \wedge (q = r))$ *where* $p \neq q$ *and* $p \neq r$

*R4)* $\langle p \overset{i}{\leftarrow} N\rangle\neg\phi \leftrightarrow (G\neg p \wedge \neg\langle p \overset{i}{\leftarrow} N\rangle\phi)$

*R5)* $\langle p \overset{i}{\leftarrow} N\rangle(\phi \wedge \psi) \leftrightarrow (\langle p \overset{i}{\leftarrow} N\rangle\phi \wedge \langle p \overset{i}{\leftarrow} N\rangle\psi)$

*R6)* $\langle p \overset{i}{\leftarrow} N\rangle(K_i\phi) \leftrightarrow (G\neg p \wedge K_i(G\neg p \rightarrow \langle p \overset{i}{\leftarrow} N\rangle\phi))$

*R7)* $\langle p \overset{i}{\leftarrow} N\rangle(K_j\phi) \leftrightarrow (G\neg p \wedge K_j\phi)$ *where* $j \neq i$

*R8)* $\langle p \overset{i}{\leftarrow} N\rangle(G\phi) \leftrightarrow G(\langle p \overset{i}{\leftarrow} N\rangle\phi)$

*Proof.* We show that P6 $\langle !p = E\rangle G\phi \leftrightarrow (p = E \wedge G(p = E \rightarrow \langle !p = E\rangle\phi))$ is valid for the case where $E$ is some $N \in \mathbb{N}$. For left-to-right, suppose $\mathcal{M}, w, h \vDash \langle !p = N\rangle G\phi$. Then $\mathcal{M}, w, h \vDash p = N$ and $\mathcal{M}^{!p=N}, w, h \vDash G\phi$ (call this $\heartsuit$). To show $\mathcal{M}, w, h \vDash G(p = N \rightarrow \langle !p = N\rangle\phi)$, suppose it is not the case. Then there is a world $w'$ in $\mathcal{M}$ and an agreeing assignment $h'$ such that $\mathcal{M}, w', h' \vDash p = N$ but $\mathcal{M}, w', h' \nvDash \langle !p = N\rangle\phi$. By the first, $w'$ survives the announcement $!p = N$ and $h'$ also agrees with it afterwards. Now by the second we have $\mathcal{M}^{!p=N}, w', h' \nvDash \phi$. But this contradicts $\heartsuit$. Hence $\mathcal{M}, w, h \vDash G(p = N \rightarrow \langle !p = N\rangle\phi)$ must be the case and the right hand side holds.
For right-to-left, suppose $\mathcal{M}, w, h \vDash (p = N \wedge G(p = N \rightarrow \langle !p = N\rangle\phi))$. To show $\mathcal{M}, w, h \vDash \langle !p = N\rangle G\phi$, suppose it is not the case. By assumption $\mathcal{M}, w, h \vDash p = N$, so the announcement does not fail and we have $\mathcal{M}^{!p=N}, w, h \nvDash G\phi$. This means there is a $w'$ in $\mathcal{M}^{!p=N}$ with an agreeing $h'$ such that $\mathcal{M}^{!p=N}, w', h' \nvDash \phi$ (call this $\heartsuit$). Only pairs of worlds and assignments where $p = N$ is true survive the announcement, therefore $\mathcal{M}, w', h' \vDash p = N$. By assumption $p = N \rightarrow \langle !p = N\rangle\phi$ is globally true in $\mathcal{M}$. Hence $\mathcal{M}, w', h' \vDash \langle !p = N\rangle\phi$ and therefore $\mathcal{M}^{!p=N}, w', h' \vDash \phi$. But this contradicts $\heartsuit$. Hence $\mathcal{M}, w, h \vDash \langle !p = N\rangle G\phi$ must hold.
Together we have shown that P6 is valid. Note that we really need the global modality here and the proof would not work for common knowledge. □

**Theorem 57.** *For every formula $\phi$ in our language there is a formula $\psi$ such that $\phi \leftrightarrow \psi$ is valid and $\psi$ does not contain any commands.*

*Proof sketch.* It suffices to note that given any formula, the reduction schemes from Theorem 56 allow us to "push" the commands inwards until they disappear at the level of atomic propositions. Then, an appropriate notion of complexity of a formula can be used for a proof by induction. □

**Example 58.** *After any creation of a private register p for the agent Jan with the actual value 5, Jan knows that $p = 5$. This statement can be expressed in our register language and we can see that the reduction schemes allow us to find an equivalent formula which does not contain any commands.*

$$\begin{aligned}
& [p \overset{Jan}{\leftarrow} 5]K_{Jan}(p = 5) \\
\overset{abbrev.}{\equiv}\; & \neg\langle p \overset{Jan}{\leftarrow} 5\rangle\neg K_{Jan}(p = a) \\
\overset{R4}{\equiv}\; & \neg(G\neg p \wedge \neg\langle p \overset{Jan}{\leftarrow} 5\rangle K_{Jan}(p = 5)) \\
\overset{R6}{\equiv}\; & \neg(G\neg p \wedge \neg(G\neg p \wedge K_{Jan}(G\neg p \rightarrow \langle p \overset{Jan}{\leftarrow} 5\rangle(p = 5)))) \\
\overset{R3a1}{\equiv}\; & \neg(G\neg p \wedge \neg(G\neg p \wedge K_{Jan}(G\neg p \rightarrow G\neg p)))
\end{aligned}$$

*Note that this formula does not contain 5 and we can easily see that it is valid:*

$$\begin{aligned}
&\equiv \neg(G\neg p \wedge \neg(G\neg p \wedge K_{Jan}(\top))) && \equiv \neg(G\neg p \wedge \neg(G\neg p \wedge \top)) \\
&\equiv \neg(G\neg p \wedge \neg G\neg p) && \equiv \neg(\bot) && \equiv \top
\end{aligned}$$

**Definition 59** (The Logic of Guessing Games)**.** *The system* GG *is given by the following rules and axiom schemes:*

- *All instances of propositional tautologies.*
- *All reduction axioms from Theorem 56.*

- *Modus Ponens:* $\dfrac{\vdash\ \phi \qquad\quad \vdash\ \phi \to \psi}{\vdash\ \psi}$

- *For all agents $i$:*

  - *Necessitation:* $\dfrac{\vdash\ \phi}{\vdash\ K_i\phi}$
  - *Distribution:* $\vdash K_i(\phi \to \psi) \to (K_i\phi \to K_i\psi)$
  - *Reflexivity:* $\vdash K_i\phi \to \phi$
  - *Euclideanness:* $\vdash \neg K_i\phi \to K_i\neg K_i\phi$

  *Note that this is an axiomatization of* S5 *for all agents. By Theorem 9 and correspondence results also symmetry:* $\vdash \phi \to K_i\neg K_i\neg\phi$ *and transitivity:* $\vdash K_i\phi \to K_iK_i\phi$ *are admissible.*

- *For the global modality $G$:*

  - *Necessitation:* $\dfrac{\vdash\ \phi}{\vdash\ G\phi}$
  - *Distribution:* $\vdash G(\phi \to \psi) \to (G\phi \to G\psi)$
  - *Reflexivity:* $\vdash G\phi \to \phi$
  - *Euclideanness:* $\vdash \neg G\phi \to G\neg G\phi$
  - *Inclusion for all agents $i$:* $\vdash G\phi \to K_i\phi$

- *For all commands $C$:*

  - *Necessitation:* $\dfrac{\vdash\ \phi}{\vdash\ [C]\phi}$
  - *Distribution:* $\vdash [C](\phi \to \psi) \to ([C]\phi \to [C]\psi)$

- *For all expressions $E$ and $E'$:*

  - *Identity:* $\vdash E = E$
  - *Substitution of (locally) equal expressions:* $\vdash E = E' \to (\phi(E) \to \phi(E'))$

- *For all nonequal natural numbers $N \neq M$ :* $\vdash N \neq M$

**Theorem 60** (Completeness)**.** *The system* GG *proves all validities in the register language for guessing games given by Definition 53. Formally: For all* $\phi \in \mathcal{L}_{\mathsf{GG}}$*, if* $\vDash \phi$*, then* $\vdash \phi$*.*

*Proof sketch.* By contraposition. We extend the well-known method of Lindenbaum Lemma, Canonical Models and Truth Lemma to our register models. For a detailed explanation of the method itself we refer to [VVK07, Chapter 7]. Let $\mathcal{L}^*_{\mathsf{GG}}$ denote the set of all command-free formulas of $\mathcal{L}_{\mathsf{GG}}$.

**Properties of maximally consistent sets**: A set of formulas $\Gamma$ is called *consistent* iff $\Gamma \not\vdash \bot$. It is called *maximally consistent* iff it has no consistent proper superset. If $\Gamma$ is a maximally consistent set of $\mathcal{L}^*_{\mathsf{GG}}$-formulas, then (i) $\Gamma$ is deductively closed, (ii) $\phi \in \Gamma$ iff $\neg\phi \notin \Gamma$ and (iii) for every $p$ there is a unique $N_p$ such that $p = N_p \in \Gamma$ ($\mathbb{N}$-property).

**Lindenbaum Lemma**: Every consistent set is a subset of a maxmally consistent set. *Proof sketch.* Note that $\mathcal{L}_{\mathsf{GG}}$ is of countable size and can thus be enumerated. Thus, given any consistent $\Gamma$ we can inductively go through all formulas, adding them to our set whenever the result is consistent and skipping it otherwise. The limit of this process is a maximally consistent set.

**Canonical Model**: For every maximally consistent set $\Theta \subseteq \mathcal{L}^*_{\mathsf{GG}}$, we define a canonical model $\mathcal{M}^\Theta := (W, \mathcal{R}, V)$ where

- $W := \{\Gamma \subset \mathcal{L}^*_{\mathsf{GG}} \mid \Gamma$ is maximally consistent and $G\phi \in \Gamma$ iff $G\phi \in \Theta$ for all $\phi\}$.
- For each agent $i$, let $R_i := \{(\Gamma, \Delta) \mid K_i\phi \in \Gamma$ iff $K_i\phi \in \Delta$ for all $\phi\}$
- The valuation function $V$ is defined at state $\Gamma$ as follows:
  - $P_\Gamma := \{p \in \mathbf{P} \mid p \in \Gamma\}$
  - $f_\Gamma(p) := (N_p, N_p, \varnothing)$ using the unique $N_p$ from the $\mathbb{N}$-property
  - $C^+_\Gamma := \{(p, q) \mid p = q \in \Gamma\}$
  - $C^-_\Gamma := \{(p, q) \mid p \neq q \in \Gamma\}$

Note that canonical models are guessing game models according to Definition 44. In particular, the defined relations are equivalence relations and the positive and negative constraint sets are consistent with each other.

**Truth Lemma**: For every state $\Gamma$ in a canonical model $\mathcal{M}^\Theta$ and every command-free formula $\phi$ we have $\phi \in \Gamma$ iff $\mathcal{M}^\Theta, \Gamma \vDash \phi$.

*Proof sketch.* By induction on complexity of $\phi$. Easy cases are $\top$, $p$, $\neg\phi$ and $\phi \wedge \psi$. For the cases of $p = N$ and $p = q$, note that by the $\mathbb{N}$-property above for each state $\Gamma$ in a canonical model there is exactly one agreeing assignment which will satisfy all the equality statements in $\Gamma$. Finally, $K_i\phi$ is taken care of by the definition of $R_i$ and $G\phi$ by the second condition in the definition of $W$ which ensures that all states agree on global truth.

Now, to show completeness by contraposition, take any $\phi \in \mathcal{L}_{\mathsf{GG}}$ such that $\nvdash \phi$. As $\mathsf{GG}$ includes all reduction axioms, there is a $\phi' \in \mathcal{L}^*_{\mathsf{GG}}$ such that $\vdash \phi \leftrightarrow \phi'$ and thus $\nvdash \phi'$. Therefore $\{\neg\phi'\}$ is consistent and by the Lindenbaum Lemma there is a maximally consistent set $\Gamma$ such that $\neg\phi' \in \Gamma$. Consider the canonical model $\mathcal{M}^\Theta$. Then by the Truth Lemma we have that $\mathcal{M}^\Theta, \Gamma \vDash \neg\phi'$, hence $\mathcal{M}^\Theta, \Gamma \nvDash \phi'$. By Theorem 56 all reduction axioms are valid, hence we also have $\mathcal{M}^\Theta, \Gamma \nvDash \phi$. Therefore, $\nvDash \phi$. $\square$

## 2.5 Implementation

In this section we will implement a Haskell model checker for the presented logic of guessing games. We first define models and formulas as data types and then translate the semantics into Haskell functions. Our program can update models with the commands described above and evaluate formulas on them. Furthermore, we implement a formula rewriting algorithm based on the reduction schemes given in Theorem 56.
At the end of the section we also provide a visualization for models and formulas – already the figures of Kripke frames above were generated automatically with this implementation.

```haskell
module GG where
-- from ghc:
import Data.List
-- local files:
import REL
import KRIPKEVIS
```

### 2.5.1 Agents, Propositions and Models

Agents are represented as integers, marked with `Ag`.

```haskell
data Agent = Ag Integer deriving (Eq,Ord)
jan,gaia,rosa :: Agent
jan  = Ag 0
gaia = Ag 1
rosa = Ag 2

instance Enum Agent where
  fromEnum = (\(Ag n) -> fromIntegral n)
  toEnum = (\n -> Ag (fromIntegral n))

instance Show Agent where
  show (Ag 0) = "Jan"
  show (Ag 1) = "Gaia"
  show (Ag 2) = "Rosa"
  show (Ag n) = "Ag "++(show n)
```

Also propositions are integers but prefixed with `P`.

```haskell
data Prp = P Integer deriving (Eq,Ord)

instance Show Prp where
  show (P 0) = "p";
  show (P n) = "p "++(show n);

prpIndex :: Prp -> Integer
prpIndex (P k) = k
```

The following code lines define states as integers, partitions of them, registers, constraints, valuations and finally our guessing game models.

```haskell
type State = Integer

type Partition =  [[State]]

type Register = (Integer,Integer,[Integer])

fullregister :: Register
fullregister = (1,10,[])

without :: Register -> Integer -> Register
without (low,high,excl) n = (low,high,nub (n:excl))

type Constraint = (Prp,Prp)

type Valuation = ([Prp],[(Prp,Register)],[Constraint],[Constraint])

data GuessM = Mo
  [State]
  [(Agent,Partition)]
  [(State,Valuation)]
  State
  deriving (Eq)

instance Show GuessM where
  show (Mo sts rel val cur) = "(Mo \n  "
    ++ show sts ++ "\n  "
    ++ show rel ++ "\n  "
    ++ show val ++ "\n  "
    ++ show cur ++ "\n  )"
```

The function m0for generates the blissful ignorance model for a given set of agents.

```haskell
m0for :: [Agent] -> GuessM
m0for ags = (Mo
  [0]
  [(a,[[0]]) | a <- ags]
  [ (0,([],[],[],[] )) ]
  0
  )
```

The following are helper functions which provide easy access to certain properties of the model at the current or another given state.

```haskell
agents :: GuessM -> [Agent]
agents (Mo _ rel _ _) = map fst rel
states :: GuessM -> [State]
states (Mo s _ _ _) = s
reachable :: GuessM -> [State]
reachable model = nub $ concat $ map (reachableBy model) (agents model)
reachableBy :: GuessM -> Agent -> [State]
reachableBy (Mo _ rel _ cur) agent
  = head $ filter (\set -> elem cur set) (apply rel agent)
reachableByFrom :: GuessM -> Agent -> State -> [State]
reachableByFrom (Mo _ rel _ _) agent state
  = head $ filter (\set -> elem state set) (apply rel agent)
reachableFrom :: GuessM -> State -> [State]
reachableFrom model state
  = nub $ concat $ map (\a -> reachableByFrom model a state) (agents model)
size :: GuessM -> Int
size (Mo sts _ _ _) = length sts
facts :: GuessM -> [Prp]
facts (Mo _ _ val cur) = fst4 (apply val cur)
factsAt :: GuessM -> State -> [Prp]
factsAt (Mo _ _ val _) state = fst4 (apply val state)
registers :: GuessM -> [(Prp,Register)]
registers (Mo _ _ val cur) = snd4 (apply val cur)
registersAt :: GuessM -> State -> [(Prp,Register)]
registersAt (Mo _ _ val _) state = snd4 (apply val state)
posConstraints :: GuessM -> [Constraint]
posConstraints (Mo _ _ val cur) = trd4 (apply val cur)
posConstraintsAt :: GuessM -> State -> [Constraint]
posConstraintsAt (Mo _ _ val _) state = trd4 (apply val state)
negConstraints :: GuessM -> [Constraint]
negConstraints (Mo _ _ val cur) = fth4 (apply val cur)
negConstraintsAt :: GuessM -> State -> [Constraint]
negConstraintsAt (Mo _ _ val _) state = fth4 (apply val state)
```

### 2.5.2 Formulas, Expressions and Commands

We now implement the three different layers of the language $\mathcal{L}_{\mathsf{GG}}$ according to definition 43: Formulas, commands and expressions.

```haskell
data Form = Top | PrpF Prp | Equal Prp Exp
  | Neg Form | Conj [Form]
  | K Agent Form | G Form | Com Com Form
  deriving (Eq,Ord,Show)

data Com = AnnounceEqual Prp Exp
  | AnnounceNotEqual Prp Exp
  | Create Prp Agent Integer
  | Com :- Com
  deriving (Eq,Ord,Show)

data Exp = PrpE Prp | Nmbr Integer
  deriving (Eq,Ord,Show)
```

The following functions define disjunctions, implications and boxes as abbreviations. Implementing these connectives as abbreviations and not as primitives is preferable because it also means that we do not have to implement separate semantics for them. As mentioned earlier (see p. 4) this only works because our basis is classical logic.

```haskell
bot :: Form
bot = Neg Top

disj :: [Form] -> Form
disj list = Neg $ Conj [ Neg d | d <- list ]

implies :: Form -> Form -> Form
implies a b = disj [Neg a, b]

box :: Com -> Form -> Form
box com form = Neg (Com com (Neg form))
```

The following helper functions return the set of propositions occurring in a formula, expression or command, respectively.

```haskell
propsInForm :: Form -> [Prp]
propsInForm Top              = []
propsInForm (PrpF aprop)     = [aprop]
propsInForm (Equal p e)      = nub $ [p] ++ propsInExp e
propsInForm (Neg formula)    = propsInForm formula
propsInForm (Conj formset)   = nub $ concat (map propsInForm formset)
propsInForm (K _ formula)    = propsInForm formula
propsInForm (G formula)      = propsInForm formula
propsInForm (Com c formula) = nub $ (propsInForm formula) ++ (propsInCom c)

propsInExp :: Exp -> [Prp]
propsInExp (PrpE aprop)      = [aprop]
propsInExp (Nmbr _)          = []

propsInCom :: Com -> [Prp]
propsInCom (Create p _ _)          = [p]
propsInCom (AnnounceEqual p e)     = nub $ [p] ++ propsInExp e
propsInCom (AnnounceNotEqual p e) = nub $ [p] ++ propsInExp e
propsInCom (com1 :- com2)          = nub $ propsInCom com1 ++ propsInCom com2
```

### 2.5.3 Assignments, Evaluating expressions

This code defines what assignments are, how we evaluate expressions and when an assignment is consistent with given constraint sets.

```haskell
type Assignment = [(Prp,Integer)]

evalEAss :: Assignment -> Exp -> Integer
evalEAss _   (Nmbr n) = n
evalEAss ass (PrpE p) = apply ass p

consistent :: [Constraint] -> [Constraint] -> Assignment -> Bool
consistent pcs ncs ass = and [all equal pcs, all (not.equal) ncs]
  where
    equal (p1,p2) = ( (apply ass p1) == (apply ass p2) )
```

Furthermore, we need a way to create assignments. We generate all assignments agreeing with the actual world in a given model in the loop function called `aALoop`.

```haskell
allAss :: GuessM -> [Assignment]
allAss model = filter (consistent pcs ncs) (aALoop [] (registers model))
  where
    pcs = posConstraints model
    ncs = negConstraints model

aALoop :: [ Assignment ] -> [ (Prp,Register) ] -> [ Assignment ]
aALoop []   []     = [ [] ]
aALoop done []     = done
aALoop []   (x:xs) = aALoop [ [ ((fst x),v) ] | v <- reg2lst (snd x) ] xs
aALoop done (x:xs) = aALoop [ (((fst x),v):o) | v <- reg2lst (snd x), o <- done ] xs

reg2lst :: Register -> [Integer]
reg2lst (low,high,excl) = foldr delete [low..high] excl
```

At most times we will not need all different complete assignments but only care about which values they assign to certain variables. The following function takes a set of propositions as an extra argument and generates partial assignments.

```haskell
allRelevantAss :: GuessM -> [Prp] -> [Assignment]
allRelevantAss model props =
  filter (consistent pcs ncs) (aALoop [] (restrict (registers model) relprops))
    where
      relprops = nub $ props ++ (\l -> (map fst l)++(map snd l)) (pcs++ncs)
      pcs = posConstraints model
      ncs = negConstraints model

restrict :: Eq a => [(a,b)] -> [a] -> [(a,b)]
restrict rel domain = filter (\pair -> elem (fst pair) domain) rel
```

The function `allRelevantAss` could be further optimized by first computing a set of relevant constraints, namely those which are directly or transitively related to the given set of propositions. The additional propositions that are relevant could then be obtained from this possibly smaller set of constraints.
However, one should keep in mind that this computation will also take its resources and thus overall there might be no gain or even a loss of efficiency. We therefore do not implement this alternative for now.

### 2.5.4 Evaluating Formulas

#### Evaluating formulas with regard to assignments

```haskell
evalAss :: GuessM -> Assignment -> Form -> Bool
evalAss model _ (PrpF p)    = (elem p (facts model))

evalAss _ ass (Equal p e)   = (a == b)
  where a = evalEAss ass (PrpE p)
        b = evalEAss ass e

evalAss _     _   Top       = True
evalAss model ass (Neg f)   = not (evalAss model ass f)
evalAss model ass (Conj fs) = and (map (evalAss model ass) fs)

evalAss model _ (G f) = all (\x -> x == Just True) set
  where set = map (\s -> evalAt model s f) (states model)

evalAss model _ (K agent f) = all (\x -> x == Just True) set
  where set = map (\s -> evalAt model s f) (reachableBy model agent)
```

The next lines implement formulas with commands. It is important to see that our implementation reflects diamonds and not boxes for the dynamic modalities. Even stronger, we let the program fail and throw a Haskell exception if the action cannot be performed or leads to a contradictory actual world (e.g. with no consistent assignments). This means that the model checker will not make a formula beginning with a failing command false but instead refuse to continue.
Our motivation for this design choice is that we mainly want to check that protocols lead to certain results and not whether they can run at all. In all our applications we will run the protocols only on models where the commands succeed.

```
evalAss model ass (Com com form) =
  if (assSet /= [])
    then and results
    else error ("No compatible assignments!")
  where
    newmodel = update model com
    assSet   = filter (subs ass) (allRelevantAss newmodel props)
    props    = nub $ propsInCom com ++ propsInForm form
    chkFct   = (\newass -> evalAss (newmodel) newass form)
    results  = map chkFct assSet
    subs a b = all (\x -> (apply a x == apply b x)) (map fst a)
```

### Evaluating formulas at the world level

In principle, on the level of a world we could evaluate all formulas which do not include statements about expressions without referring to assignment functions. But this would lead to strange effects, for example the law of excluded middle would not hold any longer as the following example shows: Suppose we have a sentence $\phi$ which is true for some assignments but false for others. Then $\phi$ and its negation would be undefined on the world level. If we now evaluated a disjunction only on this level, also $\phi \vee \neg\phi$ would be undefined which we clearly do not want. Therefore, to implement Definition 53 also the connectives which are seemingly assignment-independent have to be evaluated with respect to a certain assignment function.
The evaluation of formulas on the world level first generates all assignments and then evaluates the formula with respect to these. Note that we cannot simply use `and` on the set of results because this would return `False` for the case that we have both `True` and `False` in the set `results`.

Furthermore, to speed up the evaluation we only consider partial assignments for the propositions which occur in the formula that is being checked.

```haskell
eval :: GuessM -> Form -> Maybe Bool
eval model formula =
  if (and results)
    then
      Just True
    else
      if (and $ map not results)
        then Just False
        else Nothing
  where
    results = [ evalAss model ass formula | ass <- assSet ]
    assSet  = allRelevantAss model (propsInForm formula)

evalAt :: GuessM -> State -> Form -> Maybe Bool
evalAt (Mo sts rel val _) newcur form = eval (Mo sts rel val newcur) form
```

While our implementation yields a three-valued logic, reflected by the data type `Maybe Bool`, it still preserves the law of excluded middle for formulas with undetermined variables, as the example in Section 4.1.2 shows.

### 2.5.5 Product Update

The following three type definitions `ValChange`, `ActionS` and `Action` together with the function `productUpdate` implement action structures with factual change and product update as in Definitions 23 and 24 respectively.

```haskell
type ValChange = Valuation -> Valuation

type ActionS   = ( [State], [(State,Form)], [(State,ValChange)], [(Agent,Partition)] )

type Action    = ( ActionS, State )

productUpdate :: GuessM -> Action -> GuessM
productUpdate model@(Mo oldstates oldrel oldval oldcur) (actionStructure,faction) =
  let
    (actions, tests, changes, actrel) = actionStructure
    startcount          = (maximum oldstates) + 1
    newstatesTriples    = concat [ copiesOf (s,a) | s <- oldstates, a <- actions ]
    copiesOf (s,a)      = if (evalAt model s (apply tests a) == Just True)
                            then [ (s,a,(a*startcount + s)) ]
                            else [ ]
    newstates           = map trd3 newstatesTriples
    newValFor (s,a,t)   = (t, (apply changes a) (apply oldval s))
    newval              = map newValFor newstatesTriples
    listFor a           = cartProd (apply oldrel a) (apply actrel a)
    newPartsFor a       = [ cartProd as bs | (as,bs) <- listFor a ]
    translSingle pair   = map trd3 $ take 1 (copiesOf (pair))
    transEqClass list   = concat $ map translSingle list
    nTransPartsFor a    = map transEqClass (newPartsFor a)
    newrel              = [ (a, nTransPartsFor a)   | a <- (agents model) ]
    newcur              = trd3 $ head $ copiesOf (oldcur,faction)
    factTest            = apply tests faction
  in
    if (sort $ nub (agents model)) == (sort $ nub (map fst actrel))
      then if (eval model factTest == Just True)
        then (Mo newstates newrel newval newcur)
        else error ("The actual precondition '" ++ (show factTest) ++ "' is false!")
    else error "Agent sets of model and actionStructure are not the same!"
```

Note that we do not run any optimization on the result. We include minimizing under bisimulation and generated submodels in our later implementation of ECL.

### 2.5.6 Commands

Every command is evaluated on a model and the result is again a model. Using the implementation of `productUpdate` we can easily give definitions for our commands. The following implements $!p = E$ as given in Definition 50 and $!p \neq E$ as in Definition 51.

```haskell
update :: GuessM -> Com -> GuessM

update model (AnnounceEqual p (Nmbr n)) = productUpdate model action
  where
    action = ( ( [0],
      [(0,Equal p (Nmbr n))],
      [(0,id              )],
      actrel ), 0 )
    actrel = [ (i,[[0]]) | i <- (agents model) ]

update model (AnnounceEqual p (PrpE q)) = productUpdate model action
  where
    action = ( ( [0,1],
      [(0,Conj[PrpF p,PrpF q,Equal p (PrpE q)]),(1,Conj[Neg (PrpF p),Neg (PrpF q)])],
      [(0,id                                  ),(1,addPC                          )],
      actrel ), 0 )
    addPC  = \(fcts,regs,pc,nc) -> (fcts,regs,(p,q):pc,nc)
    actrel = [ (i,[[0,1]]) | i <- (agents model) ]

update model (AnnounceNotEqual p (Nmbr n)) = productUpdate model action
  where
    action = ( ( [0,1],
      [(0,Conj [PrpF p, Neg $ Equal p (Nmbr n)]), (1,Neg (PrpF p))],
      [(0,id                                   ), (1,exclN       )],
      actrel ), 0 )
    exclN  = \(fcts, regs, nc, pc) -> (fcts, map change regs, nc, pc)
    change (prp,reg) = if (prp == p) then (prp,without reg n) else (prp,reg)
    actrel = [ (i,[[0,1]]) | i <- (agents model) ]

update model (AnnounceNotEqual p (PrpE q)) = productUpdate model action
  where
    action = ( ( [0,1],
      [(0,Conj[PrpF p,PrpF q,Neg$Equal p (PrpE q)]),(1,Neg(Conj[PrpF p,PrpF q]))],
      [(0,id                                      ),(1,addNC                   )],
      actrel ), 0 )
    addNC  = \(fcts, regs, pc, nc) -> (fcts, regs, pc, ((p,q):nc))
    actrel = [ (i,[[0,1]]) | i <- (agents model) ]
```

It remains to define register creation. Note that both actions in the action model have the same precondition, namely that the used proposition is almost-globally false.

```haskell
update model (Create prp agent n) = productUpdate model action
  where
    pre    = G (Neg (PrpF prp))
    action = ( ( [0,1], [ (0, pre   ), (1, pre   ) ],
                        [ (0, addFct), (1, addReg) ], actrel ), 0 )
    addFct = \(fcts,reg,pc,nc) -> (prp:fcts,(prp,(n, n,[ ])):reg,pc,nc)
    addReg = \(fcts,reg,pc,nc) -> (    fcts,(prp,(1,10,[n])):reg,pc,nc)
    others = delete agent (agents model)
    actrel = [ (agent,[[0],[1]]) ] ++ [ (i,[[0,1]]) | i <- others ]
```

Finally, we implement the command ; which allows us to write longer chains of commands as one instead of repeating `update` over and over again.

```haskell
update model (comA :- comB) = update (update model comA) comB
```

### 2.5.7 Rewriting to Command-free Formulas

To automatically rewrite formulas to equivalent but command-free formulas we now implement all reduction schemes from Theorem 56. The function `rew` performs one replacement step and pushes the commands further inside. First, formulas which do not start with a command do not have to be rewritten, but their subformulas should be:

```haskell
rew :: Form -> Form
rew (Top)          = Top
rew (PrpF q)       = PrpF q
rew (Equal p e)    = Equal p e
rew (Neg formula) = Neg (rew formula)
rew (Conj forms)   = Conj (map rew forms)
rew (K i formula) = K i (rew formula)
rew (G formula)    = G (rew formula)
```

For positive Announcements we use P0 to P6. Unfortunately, the scheme P5 for formulas of the shape $\langle !p = E\rangle \widehat{K_i}\phi$ can not be implemented easily because the epistemic diamond is not a primitive in our formula data type. We therefore use an alternative reduction scheme which uses the command diamond and the epistemic box and is of the same shape as P6. While this mixed axiom is not equivalent to P5 in general, it is still valid in our S5 setting and the proof is almost the same as the one for P6 given on page 28.

```haskell
rew (Com (AnnounceEqual p e) Top)           = Equal p e
rew (Com (AnnounceEqual p e) (PrpF q))      = Conj [Equal p e, PrpF q]
rew (Com (AnnounceEqual _ _) (Equal q f))  = Equal q f
rew (Com (AnnounceEqual p e) (Neg form))    =
  Conj [Equal p e, Neg $ (Com (AnnounceEqual p e) form)]
rew (Com (AnnounceEqual p e) (Conj forms)) =
  Conj (map (\f -> (Com (AnnounceEqual p e) f)) forms)
rew (Com (AnnounceEqual p e) (K i form))    =
  Conj [Equal p e, K i (implies (Equal p e) (Com (AnnounceEqual p e) form))]
rew (Com (AnnounceEqual p e) (G form))      =
  Conj [Equal p e, G   (implies (Equal p e) (Com (AnnounceEqual p e) form))]
```

For negative announcements we use N0 to N6. Again note that the fifth line implements a mixed reduction scheme and not the original N5 from above.

```haskell
rew (Com (AnnounceNotEqual p e) Top)          = Neg (Equal p e)
rew (Com (AnnounceNotEqual p e) (PrpF q))     = Conj [Neg (Equal p e), PrpF q]
rew (Com (AnnounceNotEqual p e) (Equal q f)) = Conj [Neg (Equal p e), Equal q f]
rew (Com (AnnounceNotEqual p e) (Neg f))      =
  Conj [Neg (Equal p e), Neg (Com (AnnounceNotEqual p e) f)]
rew (Com (AnnounceNotEqual p e) (Conj fs))    =
  Conj (map (\f -> (Com (AnnounceNotEqual p e) f)) fs)
rew (Com (AnnounceNotEqual p e) (K i f))      =
  Conj [Neg (Equal p e), K i (implies (Neg (Equal p e)) (Com (AnnounceNotEqual p e) f)
      )]
rew (Com (AnnounceNotEqual p e) (G f))        =
  Conj [Neg (Equal p e), G (implies (Neg (Equal p e)) (Com (AnnounceNotEqual p e) f))]
```

Finally, the rewriting axioms R0 to R8 deal with register creation. Particularly interesting are the different cases of R3:

```haskell
rew (Com (Create p _ _) Top )       = G (Neg (PrpF p))
rew (Com (Create p _ _) (PrpF q) ) =
  if (p==q)
    then G (Neg (PrpF p))
    else Conj [G (Neg (PrpF p)), PrpF q]
rew (Com (Create p _ n) (Equal q (Nmbr m))) =
  if (p==q)
    then
      if (n==m)
```

```
        then G (Neg (PrpF p))                           -- R3a1
        else bot                                        -- R3a1'
    else
      Conj [G (Neg (PrpF p)), Equal q (Nmbr m)]         -- R3a2
rew (Com (Create p _ n) (Equal q (PrpE r))) =
  if (p==q)
    then
      if (r==p)
        then G (Neg (PrpF p))                           -- R3b1
        else Conj [G (Neg (PrpF p)), Equal q (Nmbr n)] -- R3b1'
    else
      if (r==p)
        then Conj [G (Neg (PrpF p)), Equal q (Nmbr n)] -- R3b2
        else Conj [G (Neg (PrpF p)), Equal q (PrpE r)] -- R3b2'
rew (Com (Create p i n) (Neg form))   =
  Conj [G (Neg (PrpF p)), Neg (Com (Create p i n) form)]
rew (Com (Create p i n) (Conj forms)) =
  Conj (map (\f -> (Com (Create p i n) f)) forms)
rew (Com (Create p i n) (K j form))   =
  if (i==j)
    then Conj [G (Neg (PrpF p)), K i (implies (G (Neg (PrpF p))) (Com (Create p i n)
        form))]
    else Conj [G (Neg (PrpF p)), K j form]
rew (Com (Create p i n) (G form))          = G (Com (Create p i n) form)
```

Now that we have a function on formulas which performs one step of rewriting, we know that the command-free formulas are exactly the fixed points of this function. Luckily, this observation can directly be translated into Haskell. A single-line definition suffices to get the least fixed point under `rew`.

```
cmdFree :: Form -> Form
cmdFree = lfp rew
```

**Example 61.** *As a short example what the function* `cmdFree` *does, consider the formula which we also used in Example 58, namely* $[p \overset{a}{\leftarrow} 5]K_a(p = 5)$.

```
*GGEXAMPLE> cmdFree (box (Create (P 0) jan 5) (K jan (Equal (P 0) (Nmbr 5))))
Neg (Conj [G (Neg (PrpF p)),Neg (Conj [G (Neg (PrpF p)),K Jan (Neg (Conj [Neg (Neg (G
    (Neg (PrpF p)))),Neg (G (Neg (PrpF p)))]))])])
```

*In a more human-readable form (obtained by using* `ggTexForm`*, see p. 41), and after removing unnecessary brackets and the double negation, this is the formula:*

$$\neg(G\neg p \wedge \neg(G\neg p \wedge K_{Jan}(\neg(G\neg p \wedge \neg(G\neg p)))))$$

*Remember that* $\phi \to \psi$ *is just an abbreviation for* $\neg(\phi \wedge \neg\psi)$*. We can thus see that this formula is indeed equivalent to the one which we had produced manually:*

$$\neg(G\neg p \wedge \neg(G\neg p \wedge K_{Jan}(G\neg p \to G\neg p)))$$

### 2.5.8 Visualization

In order to use the functions provided by `KRIPKEVIS` which is listed in Appendix 5 we first define functions that take propositions, valuations and the global information about models as input and return a string that can be used in LaTeX source code. Note that the constants `begintab`, `newline` and `endtab` are already defined in `KRIPKEVIS`.

```haskell
ggShowProp :: Prp -> String
ggShowProp prp = replace (replace (show prp) " 0" "") " " "_"

ggShowCnstr :: [Constraint] -> [Constraint] -> String
ggShowCnstr  []  []  = ""
ggShowCnstr pcs ncs = sepBy (positives ++ negatives) " \\text{ and } "
  where
    positives = map (niceCon "   =  ") pcs
    negatives = map (niceCon " \\neq ") ncs
    niceCon b (pA,pB) = " $ " ++ (ggShowProp pA) ++ b ++ (ggShowProp pB) ++ " $ "

ggShowVal :: Valuation -> String
ggShowVal (fcts,reg,pcs,ncs) = sepBy [niceprops,nicereg,(ggShowCnstr pcs ncs)] newline
  where
    nicereg = sepBy (map niceregSingle reg) newline
    niceregSingle (p,(n,m,x)) = if (n /= m)
      then "$" ++ (show n) ++ "\\leq " ++ (ggShowProp p) ++ "\\leq " ++ (show m) ++ " $ and $ "++(ggShowProp p)++"\\not\\in \\{" ++ (sepBy (map show (sort x)) "," ) ++ "\\} $"
      else " $ " ++ (ggShowProp p) ++ "=" ++ (show n) ++ " $ "
    niceprops = " $ " ++ sepBy (map ggShowProp fcts) "," ++ " $ "

ggInfo :: GuessM -> String
ggInfo m = begintab ++ "Agents: " ++ (sepBy (map show (agents m)) ", ") ++ endtab
```

Now we can define our own visualization functions which write LaTeX code to a file or directly compile it and open the result. For details see the listing of `KRIPKEVIS` in the appendix on page 91.

```haskell
ggTexModel :: GuessM -> String -> IO String
ggTexModel model@(Mo sts rel val cur) =
  texModel show show ggShowVal (ggInfo model) (VisModel sts rel val cur)

ggDispModel :: GuessM -> IO String
ggDispModel model@(Mo sts rel val cur) =
  dispModel show show ggShowVal (ggInfo model) (VisModel sts rel val cur)

ggTexForm :: Form -> String
ggTexForm Top           = "\\top"
ggTexForm (PrpF p)      = show p
ggTexForm (Equal p e)   = show p ++ "=" ++ (ggTexExp e)
ggTexForm (Neg f)       = " \\lnot ( " ++ ggTexForm f ++ " ) "
ggTexForm (Conj forms)  = "(" ++ (sepBy (map ggTexForm forms) " \\land ") ++ ")"
ggTexForm (K i f)       = " K_{\\text{" ++ show i ++ "}} " ++ "(" ++ ggTexForm f ++ ")"
ggTexForm (G f)         = " G " ++ ggTexForm f
ggTexForm (Com c f) =   "\\langle " ++ (ggTexCom c) ++ "\\rangle " ++ ggTexForm f

ggTexExp :: Exp -> String
ggTexExp (Nmbr n) = show n
ggTexExp (PrpE p) = show p

ggTexCom :: Com -> String
ggTexCom (Create p i n)          = " " ++ show p ++ " \\stackrel{"++(show i)++"}{\\leftarrow} " ++ show n ++ " "
ggTexCom (AnnounceEqual p e)     = " ! " ++ show p ++ " = " ++ (ggTexExp e) ++ " "
ggTexCom (AnnounceNotEqual p e)  = " ! " ++ show p ++ " \neq " ++ (ggTexExp e) ++ " "
ggTexCom (com1 :- com2)          = ggTexCom com1 ++ " ; " ++ ggTexCom com2

ggDispForm :: Form -> IO String
ggDispForm form = dispTexCode (" \\[ " ++ (ggTexForm form) ++ " \\] ")
```

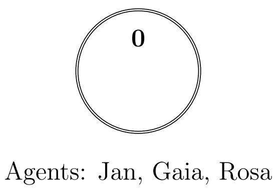
### 2.5.9 A Full Example

To conclude our implementation of GG we will present one complete round of the game and visualize all the stages of the game. To allow for easy modification, all code of this subsection is placed in a separate module.

```haskell
module GGEXAMPLE
where
import GG
```

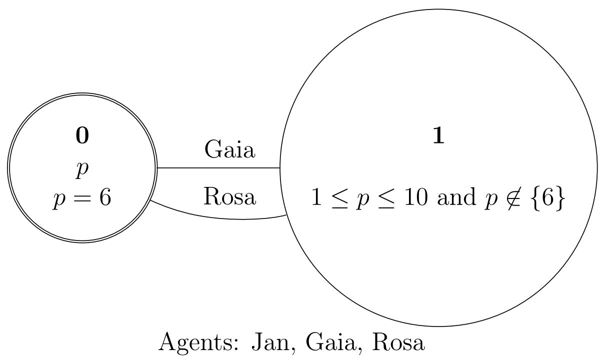
0. We start with the blissful-ignorance model m0 for Jan, Gaia and Rosa.

```haskell
m0, m1, m2, m3 :: GuessM
m0 = m0for [jan,gaia,rosa]
```



1. Our first update creates a register $p$ for Jan with his secret number 6.

```haskell
m1 = update m0 (Create (P 0) jan 6)
```



We can check that Jan knows $p$ but Gaia and Rosa do not:

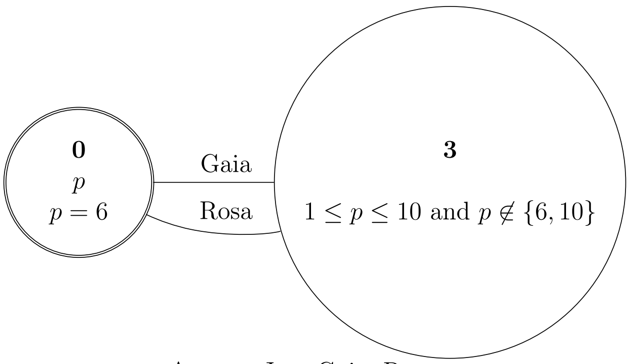
```
*GGEXAMPLE> eval m1 (K jan (PrpF (P 0)))
True
*GGEXAMPLE> eval m1 (K gaia (PrpF (P 0)))
False
*GGEXAMPLE> eval m1 (K rosa (PrpF (P 0)))
False
```

Concerning meta-knowledge we can already observe a subtlety. Gaia does not know that Jan knows *that* $p$, but she knows that he nows *whether* $p$ which we can formalize as $\phi = K_{\text{Jan}}p \lor K_{\text{Jan}}\neg p$.

```
*GGEXAMPLE> eval m1 (K rosa (K jan (PrpF (P 0))))
False
*GGEXAMPLE> phi <- return $ disj [K jan (PrpF (P 0)), K jan (Neg (PrpF (P 0)))]
*GGEXAMPLE> eval m1 (K rosa phi)
True
```

2. Suppose Rosa guesses 10, but it is wrong. Hence $p \neq 10$ is announced.

```
m2 = update m1 (AnnounceNotEqual (P 0) (Nmbr 10) )
```



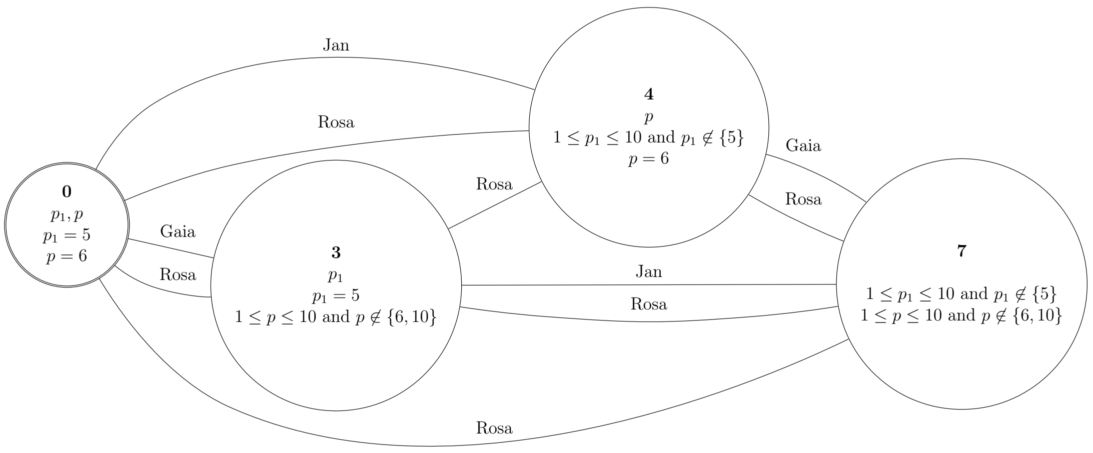
Agents: Jan, Gaia, Rosa

It is easy to check that now everyone knows that $p \neq 10$.

```
*GGEXAMPLE> map (\i -> eval m2 (K i (Neg$Equal (P 0) (Nmbr 10)))) (agents m3)
[Just True,Just True,Just True]
```

3. Next we consider the moment right before Gaia guesses 5. Her guess is saved in the new register $p_1$ and we get a model with four possible worlds. We can see that Rosa knows the least because she can not distinguish any of the worlds from another.

```
m3 = update m2 (Create (P 1) gaia 5)
```



Jan
Rosa
4
$p$
$1 \leq p_1 \leq 10$ and $p_1 \notin \{5\}$
$p = 6$
Gaia
Rosa
0
$p_1, p$
$p_1 = 5$
$p = 6$
Gaia
Rosa
Rosa
3
$p_1$
$p_1 = 5$
$1 \leq p \leq 10$ and $p \notin \{6, 10\}$
Jan
Rosa
7
$1 \leq p_1 \leq 10$ and $p_1 \notin \{5\}$
$1 \leq p \leq 10$ and $p \notin \{6, 10\}$
Rosa

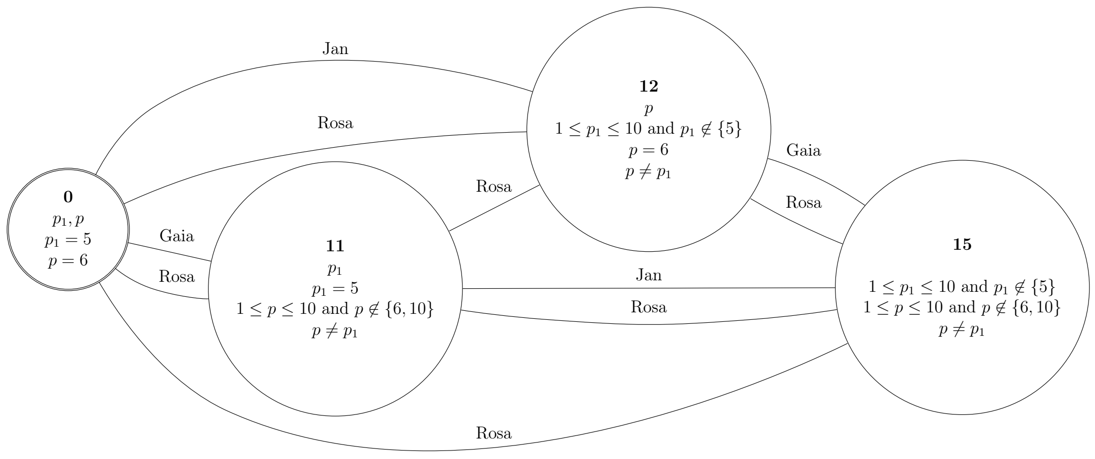
Agents: Jan, Gaia, Rosa

Now it is the case that $p \neq p_1$ but nobody knows.

```
*GGEXAMPLE> eval m3 (Neg $ Equal (P 0) (PrpE (P 1)))
Just True
*GGEXAMPLE> map (\i -> eval m3 (K i (Neg$Equal (P 0) (PrpE (P 1))))) (agents m3)
[Just False,Just False,Just False]
```

4. Now suppose someone else announces truthfully that Gaia's guess will be wrong. Note that for simplicity we do not include this extra agent in our model.

```
m4 = update m3 (AnnounceNotEqual (P 0) (PrpE (P 1) ) )
```


Jan
Rosa
12
$p$
$1 \leq p_1 \leq 10$ and $p_1 \notin \{5\}$
$p = 6$
$p \neq p_1$
Gaia
Rosa
Rosa
0
$p_1, p$
$p_1 = 5$
$p = 6$
Gaia
Rosa
11
$p_1$
$p_1 = 5$
$1 \leq p \leq 10$ and $p \notin \{6, 10\}$
$p \neq p_1$
Jan
Rosa
15
$1 \leq p_1 \leq 10$ and $p_1 \notin \{5\}$
$1 \leq p \leq 10$ and $p \notin \{6, 10\}$
$p \neq p_1$
Rosa

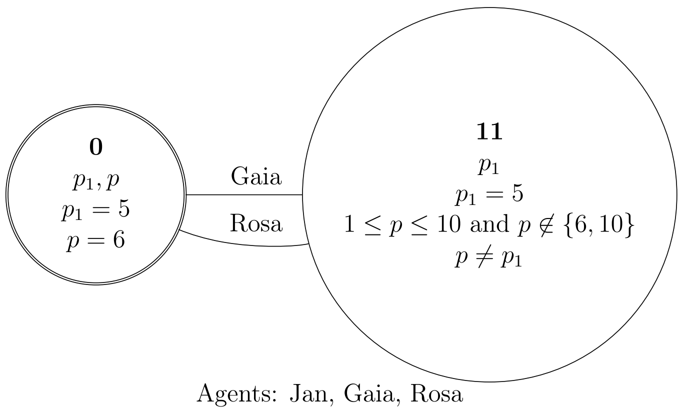
Agents: Jan, Gaia, Rosa

Note that it could not have been announced that the guess will be right, because that is false in the previous model:

```
*GGEXAMPLE> update m3 (AnnounceEqual (P 0) (PrpE (P 1) ) )
*** Exception: The actual precondition 'Conj [PrpF p,PrpF p 1,Equal p (PrpE p 1)
    ]' is false!
```

5. When Gaias guess is announced we get back to a model with two possible worlds.

```
m5 = update m4 (AnnounceEqual (P 1) (Nmbr 5))
```



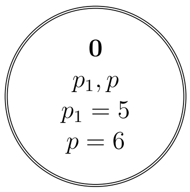
Agents: Jan, Gaia, Rosa

In this model, everyone knows that $p_1$ is not the right guess, i.e. not equal to $p$.

```
*GGEXAMPLE> map (\i -> eval m5 (K i (Neg$Equal (P 0) (PrpE (P 1))))) (agents m5)
[Just True,Just True,Just True]
```

6. Finally, suppose Rosa makes a correct guess and thereby ends the game. This means that $p = 6$ is announced, resulting in a single world.

```
m6 = update m5 (AnnounceEqual (P 0) (Nmbr 6))
```



Agents: Jan, Gaia, Rosa

The following code generates all drawings used in this subsection.

```
main :: IO ()
main = do
  ignore <- ggTexModel m0 "m0"
  putStrLn ignore
  ignore <- ggTexModel m1 "m1"
  putStrLn ignore
  ignore <- ggTexModel m2 "m2"
  putStrLn ignore
  ignore <- ggTexModel m3 "m3"
  putStrLn ignore
  ignore <- ggTexModel m4 "m4"
  putStrLn ignore
  ignore <- ggTexModel m5 "m5"
  putStrLn ignore
  ignore <- ggTexModel m6 "m6"
  putStrLn ignore
  putStrLn "Done."
```

# Chapter 3

# Epistemic Crypto Logic

The guessing games from the previous chapter elucidate which situations we can represent using register models. But while $\mathsf{GG}$ allows us to analyze games and puzzles very nicely, we can not yet represent complex protocols as they occur in cryptography – both our language and our models can not express enough *communication* and *computation.*
Thus, in this chapter we will elaborate on the shortcomings and define a new system called $\mathsf{ECL}$ (short for Epistemic Crypto Logic) in which we can be more specific about communication that is taking place and also allow our agents to do some computation on the values of registers.

## 3.1 Desiderata

### 3.1.1 Communication: Local Listener Sets

In models for $\mathcal{L}_{\mathsf{GG}}$ the equality and inequality constraints are local. Hence, in principle we already can model a situation where one agent knows whether two variables are different, and another does not, while none of the two knows the actual value of any of the two variables. But such a situation would have been unreachable in the sense that no sequence of commands available in $\mathcal{L}_{\mathsf{GG}}$ describes an update that yields this model. This is because in $\mathsf{GG}$ announcements of equality or inequality always reach all agents likewise. There is no way to send a message only to specific agents, which is an essential building block for cryptographic protocols. To model situations where announcements do not reach everyone (for example because someone is not paying attention or a message is sent via a secret channel) we will now add a *local set of listeners* to our valuations.
*Channels* between agents did not appear in number guessing games, but cryptographic protocols often describe them explicitly as in "Alice opens a channel to Bob in order to send him a message...". To model this precisely one could use local sets of channels which each are represented as a pair of agents. However, this would only suffice to model honest one-to-one communication and provide no way to represent eavesdropping situations. One could then add more structure to bring eavesdropping back into the picture but instead we can also simplify the models and remove structure, namely by ignoring channels completely. Our models for $\mathcal{L}_{\mathsf{ECL}}$ contain a local set of listeners which represents who is listening to whatever is announced by anyone. We can think of all listening agents being in the same room or - for a more technical analogy - a simple network hub rebroadcasting every package it receives to all connected clients.

The main idea is borrowed from [DHLS13]. In our new models for each $w \in W$ we get a valuation that is a tuple $(P_w, L_w, f_w, C_w^+, C_w^-)$ where $L_w \subseteq I$ is the set of listeners at $w$. The design choice for a local and not global set of listeners also allows us to model *knowledge about who is listening.* We can thus model a well-known situation from cryptography: Alice and Bob might very well believe that they are communicating privately when in fact Eve is spying on them. A detailed example is given in Section 4.1.3. In fact local listener sets are a bit too general and it seems reasonable to add a constraint, namely that all agents are self-aware about their attention. Every agent should know whether herself is listening or not.

**Definition 62** (Self-Awareness Constraint)**.** *We say that a model with local listener sets $L_w$ for each world $w$ satisfies the self-awareness constraint iff for all agents $i$ and all worlds $v$ and $w$ such that $vR_iw$ we have that $i \in L_v$ iff $i \in L_w$.*

We also add a new nullary connective $L_i$ to the language which expresses that agent $i$ is listening. The abbreviation $L_G$ says that exactly the agents in the set $G$ are listening.
The listener set should also determine what happens when new information is announced. Hence, we have to give new interpretations to $!\,p = E$ and $!\,p \neq E$ as they are only received by the local set of listeners. Basically the modifications described in Definitions 50 and 51 have to be done on copies of the previous worlds. Then we update the knowledge relations such that all listeners can distinguish the new worlds from their originals but everyone who was not paying attention confuses them.
The actions **Open** and **Close** add and remove agents from the listener set. Despite not having channels in our models their names were inspired by the usual phrases in cryptography. We can think of $\mathbf{Open}_i$ as a call for attention "Hey $i$, come here and listen!" and $\mathbf{Close}_i$ as the order to not listen any more "Okay $i$, you can go now or shut your ears." Two assumptions are crucial to the meaning of these commands: First, we assume that the agents always follow an order to listen or not listen. Second, if any agent is listening to announcements already, they will also hear calls for attention, no matter if they are the recipients, while other agents who are neither listening nor being addressed will not know who has been added to the listener set. This means that also **Open** and **Close** create copies of worlds and the order of calling for attention is relevant. In general the command $\mathbf{Open}_a; \mathbf{Open}_b$ can lead to a different result than $\mathbf{Open}_b; \mathbf{Open}_a$.
Unfortunately, our interpretations of the communication commands are no longer single action structures as in Definition 23, because not only the actions have to be filtered by preconditions but also the relation between them depend on who is listening in the original model. Therefore, our interpretation of $\mathbf{Open}_i$ depends on $L_w$ where $w$ is the current world and is not given by one single action structure.
This might seem worrisome, but our semantics are still well-defined and the implementation we give in Section 3.6 does what we want. What we have to give up is the easy embedding into the framework of [BMS98] or [BEK06]. An axiomatization of ECL will thus be more difficult than the one we gave for GG and we do not provide one here. However, we give a short argument that essentially our framework is still working with the same kind of actions.
A solution to fit our logic back into the well-known frameworks is the following syntactic trick. First, for any $G \subseteq I$, define the command ${}_G\mathbf{Open}_i$ which describes the action of calling for the attention of $i$ when the current set of listeners is $G$. This is just Definition 69 with $G$ instead of $L_w$ and the precondition $L_G$. The command will fail whenever the

set of listeners is not $G$ and any formula of the form $\langle {}_G\mathbf{Open}_i \rangle \phi$ will be false. Then we let $\mathbf{Open}_i := \bigcup_{G \subseteq I} {}_G\mathbf{Open}_i$ where $\bigcup$ is the PDL-style union. Alternatively, in order to keep $\cup$ out of our language we can define $\langle \mathbf{Open}_i \rangle \phi$ as an abbreviation:

$$\langle \mathbf{Open}_i \rangle \phi := \bigvee_{G \subseteq I} \langle {}_G\mathbf{Open}_i \rangle \phi$$

This yields the same truth conditions for $\langle \mathbf{Open}_i \rangle \phi$ as our definitions in the next section because $\langle \alpha \cup \beta \rangle \phi \leftrightarrow \langle \alpha \rangle \phi \vee \langle \beta \rangle \phi$ is a validity in PDL. Similarly, $[\alpha \cup \beta]\phi \leftrightarrow [\alpha]\phi \wedge [\beta]\phi$ is a PDL-validity and transfers to $[\mathbf{Open}_i]$:

$$[\mathbf{Open}_i]\phi \leftrightarrow \bigwedge_{G \subseteq I} [{}_G\mathbf{Open}_i]\phi$$

In the same way, we could first define ${}_G\mathbf{Close}_i$, ${}_G!p = E$ and ${}_G!p \neq E$ and let their general variants be the appropriate unions.
Admittedly, this is not very beautiful, but it does the job and saves the claim that our logic is not more expressive than those presented in [BMS98] and [BEK06]. More elegant solutions might be obtained by using other existing literature on dynamic updates of relations, see Section 5.

**Example 63** (Sending a Message)**.** *Our goal is to encode an act of communication from one agent to another in the form of "a sends $\phi$ to b". The whole sequence of commands is:*

$$?K_a\phi;\ \mathbf{Open}_b;\ !\phi;\ \mathbf{Close}_b$$

*We first use $?K_a\phi$ to test whether $K_a\phi$ is true in the actual world, because agent a should only be able to communicate $\phi$ to someone if she knows it. Second,* $\mathbf{Open}_b$ *ensure that b is listening. Then the announcement $!\phi$ is made. Note that we still only allow announcements of equality and inequality statements, so $\phi$ has to be of the form $p = E$ or $p \neq E$. Finally,* $\mathbf{Close}_b$ *removes b from the set of agents that are listening.*
*We note that, as far as b is concerned, the information $\phi$ could come from anywhere, which means our models provide* no authentication*. Furthermore, also anyone besides b who is listening already receives the info, i.e. the communication is* not secret*.*

### 3.1.2 Computation: Fast Modular Arithmetic

The second direction in which we want to extend the guessing game language is the second part of cryptographic protocols: Computation. We note that in $\mathcal{L}_{\mathsf{GG}}$ we can only say that Jan knows the value of a variable $p$ but not that he knows the value of an expression like $p + 5$ or $p + q$. We will now include such statements and also extend our semantics in a way that we get the consequences one would expect: If Jan knows $p$ then he should also know the value of $p + 5$.
A question which arises now is which calculations we want to allow. Besides keeping the language small our selection of expressions is motivated from a practical perspective on cryptography: Which arithmetic operations can be efficiently implemented? What are reasonable assumptions about the computational power of our agents that will play the role both of honest parties and adversaries? We only add statements to our language that capture feasible computation, given by the existing fast algorithms. Concretely, we allow

primality testing (e.g., the probabilistic Miller-Rabin test [Mil76, Rab80]), co-primality testing (Euclid's GCD algorithm) as well as addition, multiplication and exponentiation modulo (see the example below, and compare [DK02, PP10]).
By leaving out other operations we implicitly import articles of faith from public key cryptography, that for example factorization, discrete logarithm, etc. are *not* feasible (see [KL08, p. 271]).
We can then refine the meaning of "knowing a number" to two different conditions that can both be checked on Kripke semantics: An agent knows (i) the numbers that have a unique value in an accessible register, and (ii) the numbers that she can feasibly compute from numbers she knows. In section 4.2 we will see that this fragment of arithmetic is already expressive enough for a real-world protocol.

**Example 64** (Fast modular exponentiation)**.** *The algorithm is based on repeatedly squaring modulo $N$. For example* $x^{33} \bmod 5$ *can be computed by first computing* $x^{32} \pmod 5$ *in five steps by means of repeatedly squaring modulo* 5*:*

$$x \bmod N \rightsquigarrow x^2 \bmod 5 \rightsquigarrow x^4 \bmod 5 \rightsquigarrow \ldots \rightsquigarrow x^{32} \bmod 5.$$

*and then in one last step* $x^{33} \bmod 5 = (x^{32} \bmod 5) \times x \bmod 5$.

## 3.2 Existing Literature

An implementation of model checking for DEL with factual change via substitutions was done by Floor Sietsma in [Sie07]. It is also written in Haskell and based on the model checker DEMO by Jan van Eijck which was published in [Eij07]. While our implementation does not use code from either of the two, it is still very much inspired by this work.
A module which we do import here is `EREL` from [Eij14]. It was originally made to optimize DEMO-S5, a separate version of the already mentioned model checker which works on equivalence relations. As it provides bisimulation minimization for models with any kind of valuation, we can also employ it for our implementation of ECL.
A similar framework as the one we are presenting here has been studied by Francien Dechesne and Yanjing Wang in [DW07]. While their framework is also based on DEL and action structures, it still differs from our system in many aspects. First, instead of modular arithmetic which we use to describe computation, in Dechesne and Wang's system all agents are equipped with "cryptographic reasoning": The set of messages which an agent knows is closed under simple derivation rules, namely concatenation, splitting and applying hash functions. Another limitation of their framework concerns statements about knowledge:

> "By not having several worlds with the same message distribution, we will not be able to model higher order statements like 'A does not know that B knows that A has [the message] m'."[DW07, p. 8]

In contrast, our models here can be any Kripke structure and in particular they might include duplicate worlds with different relations. Hence, we can also model situations with interesting meta-, meta-meta- statements and so on.
In some sense we are directly continuing the work in [DW07] because they state that their ultimate goal is "to build up a dynamic epistemic framework of security verification with tool support" and suggest that a "good candidate tool is DEMO"[DW07, p. 12].

## 3.3 Syntax

**Definition 65** (Register Language for Cryptographic Protocols)**.** *Let $p$ range over $\mathbf{P}$, let $N$ range over $\mathbb{N}$ and let $i$ range over a finite set of agents $I$. The register language for cryptographic protocols $\mathcal{L}_{\mathsf{ECL}}$ consists of the following three layers which we call formulas, expressions and commands.*

$$\begin{array}{lll}\phi & ::= & \top \mid p \mid L_i \mid p = E \mid \neg\phi \mid \phi \wedge \phi \mid K_i\phi \mid G\phi \mid \langle C\rangle\phi \mid \mathbf{Prime}E \mid \mathbf{Coprime}EE \\ C & ::= & p \overset{i}{\leftarrow} E \mid \mathbf{Open}_i \mid \mathbf{Close}_i \mid !p \mid !p = N \mid !p = p \mid !p \neq N \mid !p \neq p \mid ?\phi \\ E & ::= & p \mid N \mid E + E \bmod E \mid E \times E \bmod E \mid E^E \bmod E\end{array}$$

*Furthermore, we use the same abbreviations $\bot$, $\vee$, $\to$, $\leftrightarrow$ and $[C]$ as in Definition 43. For any $G \subseteq I$ we let $L_G$ abbreviate that $G$ are the listeners: $L_G := \bigwedge_{i\in G} L_i \wedge \bigwedge_{i\notin G} \neg L_i$.*

## 3.4 Semantics

**Definition 66.** *(Crypto Models) A* crypto model *for a finite set of agents $I$ and the set of propositions $\mathbf{P}$ is a tuple $\mathcal{M} = (W, \mathcal{R}, V)$ where*

- *$(W, \mathcal{R})$ is a multi-agent $\mathsf{S5}$ frame for $I$ according to Definition 15,*
- *$V$ is a valuation function for some $Q \subseteq \mathbf{P}$ (the global set of used variables): It assigns to each world $w \in W$ a valuation which is a tuple $(P_w, L_w, f_w, C_w^+, C_w^-)$ where*
  - *$P_w \subseteq \mathbf{P}$ (the basic propositions true at $w$),*
  - *$L_w \subseteq I$ (the agents listening at $w$) satisfying self-awareness as in Definition 62,*
  - *$f_w$ is a function on $Q$ that assigns to each $q \in Q$ a triple $(n, m, X)$ with $n, m \in \mathbb{N}$, $n \leq m$, $X \subseteq \mathbb{N}$, satisfying two constraints:*
    *(i) whenever $q \in P_w$ then for $f_w(q) = (n, m, X)$ we have $n = m$ and $X = \varnothing$*
    *(ii) whenever $p \in P_v \cap P_w$ for $v, w \in W$ then $f_v(p) = f_w(p)$,*
  - *$C_w^+$ is a subset of $Q^2$ (the equality constraints of $w$)*
  - *$C_w^-$ is another subset of $Q^2$ (the inequality constraints of $w$) which is consistent with $C_w^+$ according to Definition 45 above.*

*The meaning of $G$ will now be given by the reflexive-transitive closure of the union of all relations. We refer to this relation by $\mathcal{R}^* := \left(\bigcup_{i\in I} R_i\right)^*$.*

The following definition provides the semantics for $+$, $\times$, mod and exponentiation by importing them from a theory of arithmetic which we take to be given.

**Definition 67** (Lifting assignment functions)**.** *Given an assignment function $h$ that is defined on a set of proposition letters $Q \subseteq \mathbf{P}$, we inductively define $h'$ on the set of all expressions built out of elements of $Q$ and natural numbers:*

$$h'(E) := \begin{cases} n & \text{if } E = n \text{ for some } n \in \mathbb{N} \\ h(p) & \text{if } E = p \text{ for some } p \in Q \\ h'(F) + h'(G) \bmod h'(H) & \text{if } E = F + G \bmod H \text{ for some } F, G, H \\ h'(F) \times h'(G) \bmod h'(H) & \text{if } E = F \times G \bmod H \text{ for some } F, G, H \\ h'(F)^{h'(G)} \bmod h'(H) & \text{if } E = F^G \bmod H \text{ for some } F, G, H \end{cases}$$

*Here* $+$, $\times$, mod *and exponentiation each occur as two formally different symbols. Only on the right side they are symbols from* $\mathcal{L}_{\mathsf{ECL}}$. *On the left side they refer to their ordinary meaning. Because* $h'$ *extends* $h$, *from now on we will just write* $h$ *for both.*

**Definition 68.** *(Interpretation of* $\mathcal{L}_{\mathsf{ECL}}$ *in crypto models) We define the satisfaction relation* $\mathcal{M}, w, h \models \phi$ *saying that* $\phi$ is true at $w$ with regard to $h$. *Let* $\mathcal{M} = (W, \mathcal{R}, V)$ *be a crypto model,* $w \in W$ *and* $h$ *an assignment that agrees with* $w$ *in the sense of Definition 47. Inductively we define for all* $\phi \in \mathcal{L}_{\mathsf{ECL}}$*:*

$$\begin{array}{rll}
\mathcal{M}, w, h \models \top & & \textit{always} \\
\mathcal{M}, w, h \models p & \textit{iff} & p \in P_w \\
\mathcal{M}, w, h \models L_i & \textit{iff} & i \in L_w \\
\mathcal{M}, w, h \models p = E & \textit{iff} & h(p) = h(E) \\
\mathcal{M}, w, h \models \neg\phi & \textit{iff} & \textit{not } \mathcal{M}, w, h \models \phi \\
\mathcal{M}, w, h \models \phi_1 \wedge \phi_2 & \textit{iff} & \mathcal{M}, w, h \models \phi_1 \textit{ and } \mathcal{M}, w, h \models \phi_2 \\
\mathcal{M}, w, h \models K_i\phi & \textit{iff} & wR_iw' \textit{ implies that for all } h' \multimap w' : \mathcal{M}, w', h' \models \phi \\
\mathcal{M}, w, h \models G\phi & \textit{iff} & w\mathcal{R}^*w' \textit{ implies that for all } h' \multimap w' : \mathcal{M}, w', h' \models \phi \\
\mathcal{M}, w, h \models \langle \mathbf{Open}_i \rangle\phi & \textit{iff} & \mathcal{M}^{\mathbf{Open}_i}, (w, \alpha), h \models \phi \\
\mathcal{M}, w, h \models \langle \mathbf{Close}_i \rangle\phi & \textit{iff} & \mathcal{M}^{\mathbf{Close}_i}, (w, \alpha), h \models \phi \\
\mathcal{M}, w, h \models \langle !\, p \rangle\phi & \textit{iff} & \mathcal{M}, w, h \models p \textit{ and } \mathcal{M}^{!p}, (w, \alpha), h \models \phi \\
\mathcal{M}, w, h \models \langle !\, p = E \rangle\phi & \textit{iff} & \mathcal{M}, w, h \models p = E \textit{ and } \mathcal{M}^{!p=E}, (w, \alpha), h \models \phi \\
\mathcal{M}, w, h \models \langle !\, p \neq E \rangle\phi & \textit{iff} & \mathcal{M}, w, h \models p \neq E \textit{ and } \mathcal{M}^{!p\neq E}, (w, \alpha), h \models \phi \\
\mathcal{M}, w, h \models \langle p \overset{i}{\leftarrow} N \rangle\phi & \textit{iff} & \mathcal{M}, w, h \models G\neg p \textit{ and } \mathcal{M}^{p \overset{i}{\leftarrow} N}, (w, \alpha), h \cup \{(p, N)\} \models \phi \\
\mathcal{M}, w, h \models \langle ?\psi \rangle\phi & \textit{iff} & \mathcal{M}, w, h \models \psi \wedge \phi \\
\mathcal{M}, w, h \models \langle A_1; A_2 \rangle\phi & \textit{iff} & \mathcal{M}, w, h \models \langle A_1 \rangle \langle A_2 \rangle\phi \\
\mathcal{M}, w, h \models \mathbf{Prime}E & \textit{iff} & h(E) \textit{ is a prime number} \\
\mathcal{M}, w, h \models \mathbf{Coprime}E_1E_2 & \textit{iff} & h(E_1) \textit{ and } h(E_2) \textit{ are coprime}
\end{array}$$

*where the models with superscripts are given by the Definitions 69 and 71.*

Note that $G$ is no longer the global modality but the common knowledge operator. This change is in order to keep the truth of our formulas invariant under bisimulation and generated submodels – a very useful property for optimizing our model checking algorithms. It is also one of the reasons why the axiomatization of $\mathsf{GG}$ does not directly generalize to $\mathsf{ECL}$. See Section 5 on how to axiomatize announcements and common knowledge.
As noted in Section 3.1.1 we can no longer define the meaning of communication commands as one single action structure because we have to refer to the current listener set of the model in which we are interpreting the command. The following definition therefore strictly speaking provides *schemes* for the actions.

**Definition 69.** *(Action Structures for* $\mathsf{ECL}$*) The updates from* $\mathcal{M}$ *to* $\mathcal{M}^{\mathbf{Open}_i}$, $\mathcal{M}^{\mathbf{Close}_i}$, $\mathcal{M}^{!p=E}$ *and* $\mathcal{M}^{!p\neq E}$ *are given by the following schemes of action structures with factual change, depending on the current set of listeners* $L_w$.

*(i)* $\mathbf{Open}_i$ *is given by* $(\{\alpha, \beta\}, \mathcal{R}, \alpha)$ *where*

$$\begin{array}{ll}
\mathsf{pre}(\alpha) := \top & \mathsf{change}_\alpha(P, L, f, C^+, C^-) := (P, L \cup \{i\}, f, C^+, C^-) \\
\mathsf{pre}(\beta) := \top & \mathsf{change}_\beta := \mathsf{id}
\end{array}$$

$$R_j := \begin{cases} \alpha \mid \beta & \textit{if } j \in L_w \cup \{i\} \\ \alpha\beta & \textit{otherwise} \end{cases}$$

*(ii)* $\mathbf{Close}_i$ *is given by the same action structure as* $\mathbf{Open}_i$ *up to a different change of the valuation at* $\alpha$*, namely* $\mathsf{change}_\alpha(P, L, f, C^+, C^-) := (P, L \setminus \{i\}, f, C^+, C^-)$.

*(iii) The command* $!p$ *is the same as* $!p = h(p)$ *where* $h$ *is some agreeing assignment.*

*(iv) The command* $!p = N$ *is given by* $(\{\alpha, \beta\}, \mathcal{R}, \alpha)$ *where*

$$\begin{array}{ll} \mathsf{pre}(\alpha) := (p = N) & \mathsf{change}_\alpha := \mathsf{id} \\ \mathsf{pre}(\beta) := \neg(p = N) & \mathsf{change}_\beta := \mathsf{id} \end{array}$$

$$R_j := \begin{cases} \alpha \mid \beta & \textit{if } j \in L_w \\ \alpha\beta & \textit{otherwise} \end{cases}$$

*(v) The command* $!p = q$ *is given by* $(\{\alpha, \beta, \gamma\}, \mathcal{R}, \alpha)$ *where*

$$\begin{array}{ll} \mathsf{pre}(\alpha) := p \wedge q \wedge (p = q) & \mathsf{change}_\alpha := \mathsf{id} \\ \mathsf{pre}(\beta) := \neg p \wedge \neg q & \mathsf{change}_\beta(P, f, C^+, C^-) := (P, f, C^+ \cup \{(p, q)\}, C^-) \\ \mathsf{pre}(\gamma) := \top & \mathsf{change}_\gamma := \mathsf{id} \end{array}$$

$$R_j := \begin{cases} \alpha\beta \mid \gamma & \textit{if } j \in L_w \\ \alpha\beta\gamma & \textit{otherwise} \end{cases}$$

*(vi) The command* $!p \neq N$ *is given by* $(\{\alpha, \beta, \gamma\}, \mathcal{R}, \alpha)$ *where*

$$\begin{array}{ll} \mathsf{pre}(\alpha) := p \wedge (p \neq N) & \mathsf{change}_\alpha := \mathsf{id} \\ \mathsf{pre}(\beta) := \neg p & \mathsf{change}_\beta(P, f, C^+, C^-) := (P, \mathsf{new}(f), C^+, C^-) \\ \mathsf{pre}(\gamma) := \top & \mathsf{change}_\gamma := \mathsf{id} \end{array}$$

$$\mathsf{new}(f)(q) := \begin{cases} (f^0(p), f^1(p), f^2(p) \cup \{N\}) & \textit{if } q = p \\ f(q) & \textit{otherwise} \end{cases}$$

$$R_j := \begin{cases} \alpha\beta \mid \gamma & \textit{if } j \in L_w \\ \alpha\beta\gamma & \textit{otherwise} \end{cases}$$

*(vii) The command* $!p \neq q$ *is given by* $(\{\alpha, \beta, \gamma\}, \mathcal{R}, \alpha)$ *where*

$$\begin{array}{ll} \mathsf{pre}(\alpha) := p \wedge q \wedge (p \neq q) & \mathsf{change}_\alpha := \mathsf{id} \\ \mathsf{pre}(\beta) := \neg(p \wedge q) & \mathsf{change}_\beta(P, f, C^+, C^-) := (P, f, C^+, C^- \cup \{(p, q)\}) \\ \mathsf{pre}(\gamma) := \top & \mathsf{change}_\gamma := \mathsf{id} \end{array}$$

$$R_j := \begin{cases} \alpha\beta \mid \gamma & \textit{if } j \in L_w \\ \alpha\beta\gamma & \textit{otherwise} \end{cases}$$

It follows from these definitions that for example the command $!p \neq q$ can only be executed successfully at $\mathcal{M}, w$ iff $\mathcal{M}, w \vDash p \neq E$.
This concludes our definitions for communication in $\mathcal{L}_{\mathsf{ECL}}$ and it remains to define register creation. The following generalizes Definition 52: The function changing the valuation now also copies the listener set and we allow the value of any determined expression to be mapped to the variable.

**Definition 70** (Determined Expressions)**.** *We say that an expression* $E$ *is* determined *at world* $w$ *in* $\mathcal{M}$ *iff for all* $h$ *and* $h'$ *that agree with* $w$ *we have* $h(E) = h'(E)$*. If this is the case we also write* $[\![E]\!]^{\mathcal{M},w}$ *for the value of* $E$ *at* $w$ *in* $\mathcal{M}$*.*

**Definition 71** (Register creation)**.** *For any pointed crypto model $\mathcal{M}, w$ where $E$ is determined at $w$ the action $p \overset{i}{\leftarrow} E$ is given by $(\{\alpha, \beta\}, \mathcal{R}, \alpha)$ where*

$$\begin{array}{rcl}
N & := & [\![E]\!]^{\mathcal{M},w} \\
\mathsf{pre}(\alpha) & := & G\neg p \\
\mathsf{pre}(\beta) & := & G\neg p \\
\mathsf{change}_\alpha(P, L, f, C^+, C^-) & := & (P \cup \{p\}, L, f \cup \{(p, (N, N, \varnothing))\}, C^+, C^-) \\
\mathsf{change}_\beta(P, L, f, C^+, C^-) & := & (P, L, f \cup \{(p, (0, \mathsf{regsize}, \{N\}))\}, C^+, C^-) \\
R_i & := & \begin{cases} \{\alpha \mid \beta\} & \textit{if } i = a \\ \{\alpha\beta\} & \textit{otherwise} \end{cases}
\end{array}$$

We emphasize that this action maps *the value of $E$ and not $E$ itself* to the register. It is important to realize that we only check that the expression is determined at the current world. In particular this does not mean that the agent for whom we are creating a register has all the necessary information to evaluate the expression. For now we leave this to be checked in the specification of a protocol and do not include it into our semantics. In Section 5 we also discuss the alternative idea of adding a general precondition.

**Definition 72** (World-level Truth and Validity for $\mathsf{ECL}$)**.** *For any $\phi \in \mathcal{L}_{\mathsf{ECL}}$ we define:*

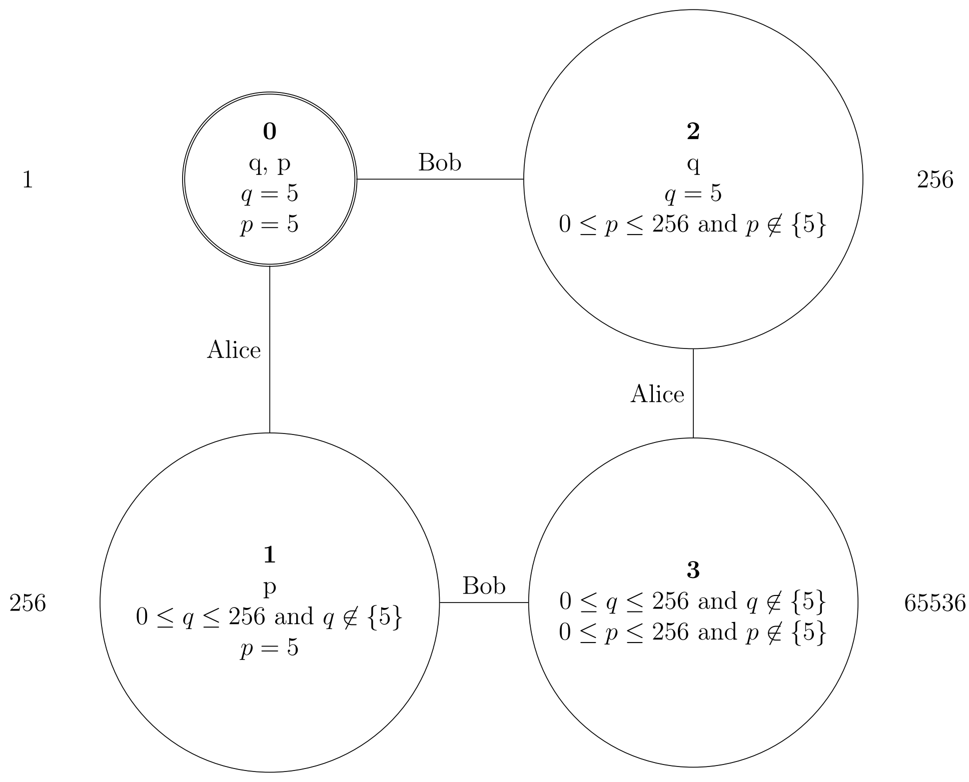
$$\begin{array}{rcl}
\mathcal{M}, w \models \phi & \textit{iff} & \forall h \textit{ with } w \multimap h : \mathcal{M}, w, h \models \phi. \\
\mathcal{M}, w \models\!\!\!| \, \phi & \textit{iff} & \forall h \textit{ with } w \multimap h : \mathcal{M}, w, h \not\models \phi.
\end{array}$$

*A formula $\phi$ is* valid *iff for all $\mathcal{M}, w$ we have that $\mathcal{M}, w \vDash \phi$. We then write $\vDash \phi$.*

Again, this definition creates truth value gaps which are closed by epistemic modalities.

## 3.5 Monte Carlo Methods

Register models allow us to focus on what matters: Instead of creating possible worlds for all possible values we only double the amount of worlds for every variable, creating one world where it has the actual value and one where it can have any other. This becomes very useful if we allow larger and larger numbers as they occur in real world applications of cryptography. A good example is the OpenPGP standard defined in [CDF+07] which is widely used for encryption and authentication of emails. Most implementations of it allow keys of a length up to 4096 bits which means that numbers up to $2^{4096}$ can occur.
But so far we only have an easier representation and visualization. To verify or falsify a formula on a register model we still have to go through just as many possibilities as we would have to on normal Kripke models. The only difference is that we now call these possibilities assignments instead of possible worlds.
This is where so-called Monte Carlo methods can help us, by providing an easier way to verify or falsify formulas. They are based on the observation that situations where the registers do not contain the correct information are vastly more probable than situations with the correct values. To illustrate this, in the following model we count the agreeing assignments at each world in the result of $p \overset{a}{\leftarrow} 5; q \overset{b}{\leftarrow} 5$, given a register size of 8.


1
0
q, p
$q = 5$
$p = 5$
Bob
2
q
$q = 5$
$0 \leq p \leq 256$ and $p \notin \{5\}$
256
Alice
Alice
256
1
p
$0 \leq q \leq 256$ and $q \notin \{5\}$
$p = 5$
Bob
3
$0 \leq q \leq 256$ and $q \notin \{5\}$
$0 \leq p \leq 256$ and $p \notin \{5\}$
65536

Now suppose we want to check whether $K_a K_b(p = q)$ is true or false at **0**. That is, we want to know if Alice knows that Bob knows that $p = q$. Because Alice confuses **0** and **1** and Bob confuses **1** and **3** this means we will also have to check the statement $p = q$ at the world **3**. But do we really have to go through 65536 different assignments? We argue that this is not necessary. If we randomly pick an assignment $h$ that agrees with **3** we get the following probabilities:

$$P(h(p) = h(q)) = \frac{255}{65536} \approx 0.389\%$$

$$P\left(h(p) \neq h(q)\right) = \frac{65281}{65536} \approx 99.611\%$$

When we are checking $p = q$ at **3** in order to check $K_a K_b(p = q)$ at **0** we are particularly interested in assignments $h$ for which $h(p) \neq h(q)$: Finding one of them suffices to say that $K_a K_b(p = q)$ is false at **0**.

Hence such statements can be checked by means of an approximate model checking algorithm. Because the results obtained in this way are probabilistic we call this a Monte Carlo method. More generally, they are based on this idea:

$$\mathcal{M}, w \;\models\!\!\approx\; \phi \text{ iff for “enough” } h \text{ with } w \multimap h : \mathcal{M}, w, h \models \phi.$$

The symbol $\models\!\!\approx$ can be read as “probably makes true” or “probably models”. To specify “enough” we choose a rather small $n \in \mathbb{N}$ – our implementation in section 3.6.7 uses $n = 2$. Then to evaluate a statement at a world $w$ we first randomly generate a list of $n$ assignments $h_1, \ldots, h_n$ that all agree with $w$. For each $h_i$ in the list we check $\mathcal{M}, w, h_i \models\!\!\approx \phi$ which is the same as $\vDash$ from Definition 68 up to the clause for commands and modalities which are explained in the next definition. We then say that $\phi$ is probably true/false iff it is true/false with regard to $h_i$ for all $i \leq n$ while in any mixed case it is undefined.

**Definition 73** (Monte Carlo Update and Semantics). *The* Monte Carlo update *of a crypto model $\mathcal{M}, w$ with an action is the same as the product update according to Definition 24 up to the change that $W' := \{(w, \alpha) \in W \times A \mid w \mathrel{|\!\!\approx} \mathsf{pre}(\alpha)\}$. That is, also the preconditions are to be checked with regard to $n$ randomly picked and not all assignments. For formulas of the following shapes we define:*

$$\begin{array}{rll}
\mathcal{M}, w, h \mathrel{|\!\!\approx} \neg\phi & \textit{iff} & \textit{not } \mathcal{M}, w, h \mathrel{|\!\!\approx} \phi \\
\mathcal{M}, w, h \mathrel{|\!\!\approx} \phi_1 \wedge \phi_2 & \textit{iff} & \mathcal{M}, w, h \mathrel{|\!\!\approx} \phi_1 \textit{ and } \mathcal{M}, w, h \mathrel{|\!\!\approx} \phi_2 \\
\mathcal{M}, w, h \mathrel{|\!\!\approx} K_i\phi & \textit{iff} & (w, w') \in R_i \textit{ implies } \mathcal{M}, w' \mathrel{|\!\!\approx} \phi \\
\mathcal{M}, w, h \mathrel{|\!\!\approx} G\phi & \textit{iff} & (w, w') \in \mathcal{R}^* \textit{ implies } \mathcal{M}, w' \mathrel{|\!\!\approx} \phi \\
\mathcal{M}, w, h \mathrel{|\!\!\approx} \langle \mathbf{Open}_i \rangle\phi & \textit{iff} & \mathcal{M}^{\mathbf{Open}_i}, w, h \mathrel{|\!\!\approx} \phi \\
\mathcal{M}, w, h \mathrel{|\!\!\approx} \langle \mathbf{Close}_i \rangle\phi & \textit{iff} & \mathcal{M}^{\mathbf{Close}_i}, w, h \mathrel{|\!\!\approx} \phi \\
\mathcal{M}, w, h \mathrel{|\!\!\approx} \langle !\ p = E \rangle\phi & \textit{iff} & \mathcal{M}, w, h \mathrel{|\!\!\approx} p = E \textit{ and } \mathcal{M}^{p=E}, w, h \mathrel{|\!\!\approx} \phi \\
\mathcal{M}, w, h \mathrel{|\!\!\approx} \langle !\ p \neq E \rangle\phi & \textit{iff} & \mathcal{M}, w, h \mathrel{|\!\!\approx} p \neq E \textit{ and } \mathcal{M}^{p\neq E}, w, h \mathrel{|\!\!\approx} \phi \\
\mathcal{M}, w, h \mathrel{|\!\!\approx} \langle p \overset{i}{\leftarrow} N \rangle\phi & \textit{iff} & \mathcal{M}^{p \overset{i}{\leftarrow} N}, w, (h \cup \{(p, N)\}) \mathrel{|\!\!\approx} \phi \\
\mathcal{M}, w, h \mathrel{|\!\!\approx} \langle A_1; A_2 \rangle\phi & \textit{iff} & \mathcal{M}, w, h \mathrel{|\!\!\approx} \langle A_1 \rangle \langle A_2 \rangle\phi
\end{array}$$

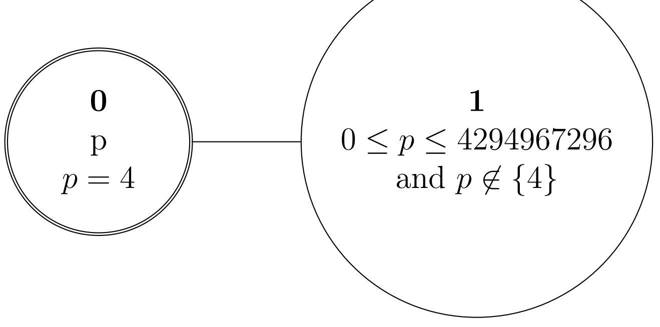
*where the models with superscripts are given by the Monte Carlo update with the actions described in Definitions 69 and 71. For all other formulas $\phi$ we define:*

$$\mathcal{M}, w, h \mathrel{|\!\!\approx} \phi \quad \textit{iff} \quad \mathcal{M}, w, h \vDash \phi$$

**Definition 74** (Monte Carlo Truth). *Fix a number $n > 0$ and assume that agreeing assignments can be picked randomly. We say that $\phi$ is* probably true *at $\mathcal{M}, w$ and write $\mathcal{M}, w \mathrel{|\!\!\approx} \phi$ iff for $n$ randomly picked assignments $h_1, \dots, h_n$ we have $\mathcal{M}, w, h_i \mathrel{|\!\!\approx} \phi$.*

For many interesting formulas the probability of disagreement between $\mathrel{|\!\!\approx}$ and $\vDash$ can be made arbitrarily small by using a larger $n$. The choice of $n$ then provides a trade-off between computation time and reliability. Furthermore, the probability gets better for larger register sizes.
Among the formulas which can be checked this way is our formalization of the Diffie-Hellman key exchange in Section 4.2. Relying on such results can also be compared to articles of faith of cryptanalysis where for example brute force attacks on number secrets are assumed to be impossible.
However, the results of a Monte Carlo algorithm should be used carefully. As the next example shows, it is also easy to come up with models and formulas for which a probabilistic method will almost certainly return the wrong result.

**Example 75** (Monte Carlo Failure). *Consider the formula $\langle p \overset{a}{\leftarrow} 4 \rangle K_b(p \neq 8)$, evaluated on a blissful ignorance model for Alice and Bob and using a registersize of $2^{32}$. This means that $K_b(p \neq 8)$ has to be checked in the following model:*



*Here there are 4294967295 assignments that agree with the world* **1**. *Only one of them renders $p \neq 8$ false. Our algorithm will thus most probably say that Bob knows that $p \neq 8$. Indeed we can reproduce this failure in our implementation of the Monte Carlo evaluation, see Section 4.1.4.*

This example shows that the reliability of Monte Carlo methods varies for different types of formulas. Many cases are unproblematic, including the combinations of equality and knowledge statements which we will use to describe cryptographic protocols and their result. For inequalities and the knowledge thereof, probabilities should be taken into account. The details can be worked out in a probabilistic logic of communication and change as presented in [Ach14, Chapter 5], but we will not do so here.

## 3.6 Implementation

This section contains our implementation of ECL. We first define the necessary data types, this time based on our valuations with local listener sets. Then we go on to write evaluation functions for expressions, formulas and commands. After the normal implementation we will also provide a Monte Carlo evaluation. Whenever the logic and therefore the implementation do not differ substantially from the one presented in Section 2.5 we keep our annotations to a minimum. We import the same modules as in GG and some more: Libraries for plotting and primality testing from ghc, the module EREL from [Eij14] and our own libraries for modular exponentiation and pseudo-randomness.

```haskell
module ECL where
-- from ghc:
import Data.List
import Data.Numbers.Primes
import Graphics.Gnuplot.Simple
-- local files:
import REL
import MODEXP
import RAND
import EREL (minimize,convertMapping)
import KRIPKEVIS
```

### 3.6.1 Agents, Propositions and Models

```haskell
data Agent = Ag Integer deriving (Eq,Ord)
alice, bob, carol, dave, eve, mallory :: Agent
alice   = Ag 0; bob     = Ag 1;
carol   = Ag 2; dave    = Ag 3;
eve     = Ag 4; mallory = Ag 5

instance Enum Agent where
  fromEnum = (\(Ag n) -> fromIntegral n)
  toEnum   = (\n -> Ag (fromIntegral n))

instance Show Agent where
  show (Ag 0) = "Alice";  show (Ag 1) = "Bob"
  show (Ag 2) = "Carol";  show (Ag 3) = "Dave"
  show (Ag 4) = "Eve";    show (Ag 5) = "Mallory"
  show (Ag n) = ('a': show n)
```

We use four different letters for propositions and enumerate them with prpIndex.

```haskell
data Prp = P Integer | Q Integer | R Integer | S Integer deriving (Eq,Ord)
instance Show Prp where
  show (P 0) = "p";  show (P n) = "p "++(show n)
  show (Q 0) = "q";  show (Q n) = "q "++(show n)
  show (R 0) = "r";  show (R n) = "r "++(show n);
  show (S 0) = "s";  show (S n) = "s "++(show n);

prpIndex :: Prp -> Integer
prpIndex (P k) = k*4
prpIndex (Q k) = k*4 + 1
prpIndex (R k) = k*4 + 2
prpIndex (S k) = k*4 + 3
```

Before models we introduce data types for states, partitions, registers and constraints. It is also here that we fix a global registersize of $2^8$.

```haskell
type State      = Integer
type Partition  = [[State]]
type Register   = (Integer,Integer,[Integer])
type Constraint = (Prp,Prp)

registersize :: Integer
registersize = 2^(8::Int)

fullregister :: Register
fullregister = (0,registersize-1,[])

without :: Register -> Integer -> Register
without (low,high,excl) n = (low,high,nub (n:excl))
```

A valuation now consists of the facts, listeners, some registers, positive constraints and negative constraints. The next code also defines pointed models, how to show them and the blissful ignorance model for any set of agents.

```haskell
type Valuation = ([Prp],[Agent],[(Prp,Register)],[Constraint],[Constraint])

data CryptoM = Mo [State] [(Agent,Partition)] [(State,Valuation)] State deriving (Eq)

instance Show CryptoM where
  show (Mo sts rel val cur) = "(Mo "
    ++ show sts ++ "\n    "
    ++ show rel ++ "\n    "
    ++ show val ++ "\n    "
    ++ show cur ++ " )"

cm0for :: [Agent] -> CryptoM
cm0for ags = (Mo
  [0]
  [(a,[[0]]) | a <- ags]
  [ (0,([],[],[],[],[] )) ]
  0 )
```

The following functions provide convenient access to various properties of our models.

```haskell
states :: CryptoM -> [State]
states (Mo sts _ _ _) = sts
agents :: CryptoM -> [Agent]
agents (Mo _ rel _ _) = map fst rel
reachable :: CryptoM -> [State]
reachable model = nub $ concat $ map (reachableBy model) (agents model)
reachableBy :: CryptoM -> Agent -> [State]
reachableBy (Mo _ rel _ cur) agent
  = head $ filter (\set -> elem cur set) (apply rel agent)
reachableByFrom :: CryptoM -> Agent -> State -> [State]
reachableByFrom (Mo _ rel _ _) agent state
  = head $ filter (\set -> elem state set) (apply rel agent)
```

```
reachableFrom :: CryptoM -> State -> [State]
reachableFrom model state
  = nub $ concat $ map (\a -> reachableByFrom model a state) (agents model)
size :: CryptoM -> Int
size (Mo sts _ _ _) = length sts
facts :: CryptoM -> [Prp]
facts (Mo _ _ val cur) = fst5 (apply val cur)
factsAt :: CryptoM -> State -> [Prp]
factsAt (Mo _ _ val _) state = fst5 (apply val state)
listeners :: CryptoM -> [Agent]
listeners (Mo _ _ val cur) = snd5 (apply val cur)
listenersAt :: CryptoM -> State -> [Agent]
listenersAt (Mo _ _ val _) state = snd5 (apply val state)
nonlisteners :: CryptoM -> [Agent]
nonlisteners model = foldr delete (agents model) (listeners model)
nonlistenersAt :: CryptoM -> State -> [Agent]
nonlistenersAt model state = foldr delete (agents model) (listenersAt model state)
registers :: CryptoM -> [(Prp,Register)]
registers (Mo _ _ val cur) = trd5 (apply val cur)
registersAt :: CryptoM -> State -> [(Prp,Register)]
registersAt (Mo _ _ val _) state = trd5 (apply val state)
posConstraints :: CryptoM -> [Constraint]
posConstraints (Mo _ _ val cur) = fth5 (apply val cur)
posConstraintsAt :: CryptoM -> State -> [Constraint]
posConstraintsAt (Mo _ _ val _) state = fth5 (apply val state)
negConstraints :: CryptoM -> [Constraint]
negConstraints (Mo _ _ val cur) = fft5 (apply val cur)
negConstraintsAt :: CryptoM -> State -> [Constraint]
negConstraintsAt (Mo _ _ val _) state = fft5 (apply val state)
```

This function changes the current world of a model. It will only be used for testing because our language does not contain any commands that change change the actual world.

```
makeActual :: CryptoM -> State -> CryptoM
makeActual (Mo sts rel val _) newcur = if (elem newcur sts)
  then (Mo sts rel val newcur)
  else error ("World "++(show newcur)++" does not exist in this model!")
```

### 3.6.2 Bisimulation and Generated Submodels

The following generates smaller equivalent models using the methods from Section 1.4. Our function bisiMin employs convertMapping from the module EREL from [Eij14] to obtain a bisimilar model. Note that we have to track the mapping of worlds to set the correct actual world in the new model. The function genMin finds the generated submodel by marking all reachable worlds until it reaches a fixpoint.

```
bisiMin :: CryptoM -> CryptoM
bisiMin (Mo oldstates oldrel oldval oldcur) = (Mo newstates newrel newval newcur)
  where
    newval          = nub $ map (\(x,v) -> (apply bisim x, v)) newvalEntries
    newvalEntries   = filter (\x -> (elem (apply bisim (fst x)) newstates)) oldval
    newstates       = nub $ map (apply bisim) oldstates
    (newrel,bisim)  = convertMapping [0..] $ minimize oldrel oldval
    newcur          = apply bisim oldcur

genMin :: CryptoM -> CryptoM
genMin model@(Mo _ oldrel oldval cur) = (Mo newstates newrel newval cur)
  where
    newstates    = lfp (\set -> mark set) (reachable model)
    newval       = filter (\x -> elem (fst x) newstates) oldval
    newrel       = [ (a, newrelfor a) | a <- agents model ]
    newrelfor a = filter (\part -> elem (head part) newstates) (apply oldrel a)
    mark marked = nub $ concat $ map (reachableFrom model) marked
```

### 3.6.3 Formulas

The following data types represent all three layers of the language $\mathcal{L}_{\mathsf{ECL}}$ according to Definition 65.

```haskell
data Form = Top | PrpF Prp | L Agent | Equal Exp Exp
  | Neg Form | Conj [Form]
  | K Agent Form | G Form
  | Com Com Form
  | Prime Exp | Coprime Exp Exp
  deriving (Eq,Ord,Show)

data Com =  Open Agent | Close Agent
  | Create Prp Agent Exp | CreateSized Prp Agent Exp Integer
  | Announce Prp | AnnounceEqual Prp Exp | AnnounceNotEqual Prp Exp
  | Test Form | Com :- Com
  deriving (Eq,Ord,Show)

data Exp = PrpE Prp | Nmbr Integer
  | PlusMod Exp Exp Exp | TimesMod Exp Exp Exp | PowerMod Exp Exp Exp
  deriving (Eq,Ord,Show)
```

Disjunctions, implications and boxes are again defined as abbreviations and the helper function `lst2cmd` allows us to specify longer commands as lists:

```haskell
bot :: Form
bot = Neg Top

disj :: [Form] -> Form
disj list = Neg $ Conj [ Neg d | d <- list ]

implies :: Form -> Form -> Form
implies a b = disj [Neg a, b]

box :: Com -> Form -> Form
box com form = Neg ( Com com ( Neg form ) )

lst2cmd :: [Com] -> Com
lst2cmd [] = error "empty list"
lst2cmd [c] = c
lst2cmd [c1,c2] = c1 :- c2
lst2cmd (c1:c2s) = c1 :- (lst2cmd c2s)
```

The following functions compute the set of propositions occurring in a formula.

```haskell
propsInForm :: Form -> [Prp]
propsInForm Top             = []
propsInForm (PrpF aprop)    = [aprop]
propsInForm (Neg formula)   = propsInForm formula
propsInForm (Conj forms)    = nub $ concat (map propsInForm forms)
propsInForm (K _ formula)   = propsInForm formula
propsInForm (G formula)     = propsInForm formula
propsInForm (L _ )          = []
propsInForm (Com c formula) = nub $ (propsInForm formula) ++ (propsInCom c)
propsInForm (Equal a b)     = nub $ propsInExp a ++ propsInExp b
propsInForm (Prime a)       = propsInExp a
propsInForm (Coprime a b)   = nub $ propsInExp a ++ propsInExp b
```

The same is needed for expressions and now includes more cases in $\mathsf{GG}$.

```haskell
propsInExp :: Exp -> [Prp]
propsInExp (PrpE aprop)      = [aprop]
propsInExp (Nmbr _)          = []
propsInExp (PlusMod  a b c) = nub $ concat $ map propsInExp [a,b,c]
propsInExp (TimesMod a b c) = nub $ concat $ map propsInExp [a,b,c]
propsInExp (PowerMod a b c) = nub $ concat $ map propsInExp [a,b,c]
```

Also to list the propositions occurring in a command we need some additional cases.

```haskell
propsInCom :: Com -> [Prp]
propsInCom (Open _)                  = []
propsInCom (Close _)                 = []
propsInCom (Create p _ e)            = nub $ [p] ++ propsInExp e
propsInCom (CreateSized p _ e _)     = nub $ [p] ++ propsInExp e
propsInCom (Announce p)              = [p]
propsInCom (AnnounceEqual p e)       = nub $ [p] ++ propsInExp e
propsInCom (AnnounceNotEqual p e)    = nub $ [p] ++ propsInExp e
propsInCom (comA :- comB)            = nub $ propsInCom comA ++ propsInCom comB
propsInCom (Test f)                  = nub $ propsInForm f
```

The following implements definition 67. For addition, multiplication and modulo we use the built-in functions of Haskell which are efficient enough for our purposes. In contrast, the built-in exponentiation function ^ is rather slow. Instead we use `exM` which is a fast algorithm for modular exponentiation from the module `MODEXP` which is listed in the appendix (p. 90).

```haskell
type Assignment = [(Prp,Integer)]
evalEAss :: Assignment -> Exp -> Integer
evalEAss _   (Nmbr n) = n
evalEAss ass (PrpE p) = apply ass p
evalEAss ass (PlusMod a b m)  =
  mod ( (evalEAss ass a) + (evalEAss ass b) ) (evalEAss ass m)
evalEAss ass (TimesMod a b m) =
  mod ( (evalEAss ass a) * (evalEAss ass b) ) (evalEAss ass m)
evalEAss ass (PowerMod a b m) =
  exM (evalEAss ass a) (evalEAss ass b) (evalEAss ass m)
```

Next, we define consistency and generate partial assignments. As in the previous implementation we use `aALoop` to build up assignments step by step, dealing with one propositional variable at a time.

```haskell
consistent :: [Constraint] -> [Constraint] -> Assignment -> Bool
consistent pcs ncs ass = and [all equal pcs, all (not.equal) ncs]
  where
    equal (p1,p2) = ( (apply ass p1) == (apply ass p2) )

allAss :: CryptoM -> [Assignment]
allAss model = filter (consistent pcs ncs) (aALoop [] (registers model))
  where
    pcs = posConstraints model
    ncs = negConstraints model

aALoop :: [ Assignment ] -> [ (Prp,Register) ] -> [ Assignment ]
aALoop []   []     = [ [] ]
aALoop done []     = done
aALoop []   (x:xs) = aALoop [ [ ((fst x),v) ] | v <- reg2lst (snd x) ] xs
aALoop done (x:xs) = aALoop [ (((fst x),v):o) | v <- reg2lst (snd x), o <- done ] xs

reg2lst :: Register -> [Integer]
reg2lst (low,high,excl) = foldr delete [low..high] excl

allRelevantAss :: CryptoM -> [Prp] -> [Assignment]
allRelevantAss model props =
  filter (consistent pcs ncs) (aALoop [] (restrict (registers model) relprops))
    where
      relprops = nub $ props ++ (\l -> (map fst l)++(map snd l)) (pcs++ncs)
      pcs      = posConstraints model
      ncs      = negConstraints model

restrict :: Eq a => [(a,b)] -> [a] -> [(a,b)]
restrict rel domain = filter (\pair -> elem (fst pair) domain) rel
```

### 3.6.4 Evaluation

As for GG we implement evaluation of formulas with regard to assignments and on the world-level. New cases of formulas are the primality tests and the atomic propositions for listening. The truth value of some formulas only depend on the model or the assignment but not both. This allows us to use the _ sign in their Haskell definitions.

```haskell
evalAss :: CryptoM -> Assignment -> Form -> Bool

evalAss _     _   Top           = True
evalAss model _   (PrpF prp)    = elem prp (facts model)
evalAss _     ass (Equal a b)   = (evalEAss ass a) == (evalEAss ass b)
evalAss _     ass (Prime e)     = isPrime (evalEAss ass e)
evalAss _     ass (Coprime a b) = (gcd (evalEAss ass a) (evalEAss ass b) == 1)
evalAss model ass (Neg form)    = not (evalAssMC model ass form)
evalAss model ass (Conj forms)  = and (map (evalAssMC model ass) forms)

evalAss model _ (K agent form) = and results
  where results   = map evalthere (reachableBy model agent)
        evalthere = (\v -> (evalAt model v form==Just True))

evalAss model _ (G form) = and results
  where results   = map evalthere (states (genMin model))
        evalthere = (\v -> (evalAt model v form==Just True))

evalAss model _ (L agent) = elem agent (listeners model)

evalAss model ass (Com com form) =
  if (assSet /= [])
    then and results
    else error ("No compatible assignments!")
  where
    newmodel = update model com
    assSet   = filter (subs ass) (allRelevantAss newmodel props)
    props    = nub $ propsInCom com ++ propsInForm form
    chkFct   = (\newass -> evalAss (newmodel) newass form)
    results  = map chkFct assSet
    subs a b = all (\x -> (apply a x == apply b x)) (map fst a)
```

As in GG, the evaluation at the world level returns a `Maybe Bool`.

```haskell
eval :: CryptoM -> Form -> Maybe Bool
eval model formula =
  if (and results)
    then
      Just True
    else
      if (and $ map not results)
        then Just False
        else Nothing
  where
    results = [ evalAss model ass formula | ass <- assSet ]
    assSet  = allRelevantAss model (propsInForm formula)

evalAt :: CryptoM -> State -> Form -> Maybe Bool
evalAt (Mo sts rel val _) newcur form = eval (Mo sts rel val newcur) form
```

For convenience we also implement an abbreviation which evaluates a given formula at all states of a model and returns a list of results. This is mainly useful for testing and does not have a counterpart in our formal definitions of ECL.

```haskell
evalAtAll :: CryptoM -> Form -> [(State,Maybe Bool)]
evalAtAll m form = zip (states m) (map (\w -> evalAt m w form) (states m))
```

### 3.6.5 Product Update

We now implement actions and updates with factual change as given by Definitions 23 and 24 respectively. Note that we can not use the code from our implementation of GG because the data types `GuessM` and `CryptoM` are different. Furthermore, we want to use the same function in our Monte Carlo implementation in Section 3.6.7. Therefore we also parameterize on the function used to evaluate the preconditions.

```haskell
type ValChange = Valuation -> Valuation
type ActionS = ( [State], [(State,Form)], [(State,ValChange)], [(Agent,Partition)] )
type Action = (ActionS,State)

productUpdateWithEvAtFct ::
  (CryptoM -> State -> Form -> Maybe Bool) -> CryptoM -> Action -> CryptoM
productUpdateWithEvAtFct evAtFct model (actionStructure,faction) =
  let
    (Mo oldstates oldrel oldval oldcur) = model
    (actions, tests, changes, actrel)   = actionStructure
    startcount         = (maximum oldstates) + 1
    copiesOf (s,a)     = if (evAtFct model s (apply tests a) == Just True)
                           then [ (s,a,(a*startcount + s)) ]
                           else [ ]
    newstatesTriples   = concat [ copiesOf (s,a) | s <- oldstates, a <- actions ]
    newstates          = map trd3 newstatesTriples
    newValFor (s,a,t)  = (t, (apply changes a) (apply oldval s))
    newval             = map newValFor newstatesTriples
    listFor ag         = cartProd (apply oldrel ag) (apply actrel ag)
    newPartsFor ag     = [ cartProd as bs | (as,bs) <- listFor ag ]
    translSingle pair  = filter (\x-> elem x newstates) $ map trd3 $ copiesOf (pair)
    transEqClass list  = concat $ map translSingle list
    nTransPartsFor ag  = filter (\x-> x/=[]) $ map transEqClass (newPartsFor ag)
    newrel             = [ (a, nTransPartsFor a)  | a <- (agents model) ]
    newcur             = trd3 $ head $ copiesOf (oldcur,faction)
    factTest           = apply tests faction
  in
    if (sort $ nub (agents model)) == (sort $ nub (map fst actrel))
      then if (evAtFct model oldcur factTest == Just True)
        then genMin $ bisiMin $ (Mo newstates newrel newval newcur)
        else error ("Actual precondition '" ++ (show factTest) ++ "' is false!")
      else error "Agents of model and actionStructure are not the same!"

productUpdate :: CryptoM -> Action -> CryptoM
productUpdate = productUpdateWithEvAtFct evalAt
```

Note the commands `genMin` and `bisiMin` in front of the new model. In contrast to the previous implementation, this time we optimize the result of any product update under bisimulation and submodel generation. This will ensure our models do not get redundant.

### 3.6.6 Commands

The implementation of commands gives us another reason not to use the syntactic trick of interpreting the command $\mathbf{Open}_i$ as a PDL-style union of $\{_G\mathbf{Open}_i \mid G \subseteq I\}$ as described in Section 3.1.1. Directly translating this idea into Haskell code would make our implementation unnecessarily complicated and inefficient. Instead we follow definition 69 and interpret the commands depending on the listener set at the current world.

```haskell
openAction, closeAction :: CryptoM -> Agent -> Action
openAction model a = (([0,1], [(0,Top),(1,Top)], [(0,id),(1,addLst)], actrel), 1)
  where
    addLst = \(fs,lstnrs,reg,pcs,ncs) -> (fs,nub(a:lstnrs),reg,pcs,ncs)
    actrel = [ (i,[[0],[1]]) | i <- hear  ] ++ [ (i,[[0,1]]) | i <- dumb ]
    hear   = nub (a : listeners model)
    dumb   = foldr delete (agents model) hear
```

```haskell

closeAction model a = (([0,1], [(0,Top),(1,Top)], [(0,id),(1,remLst)], actrel), 1)
  where
    remLst = \(fs,lstnrs,reg,pcs,ncs) -> (fs,delete a lstnrs,reg,pcs,ncs)
    hear   = nub (a : listeners model)
    dumb   = foldr delete (agents model) hear
    actrel = [ (i,[[0],[1]]) | i <- hear  ] ++ [ (i,[[0,1]]) | i <- dumb ]

announceAction :: CryptoM -> Prp -> Action
announceAction model p = ( ( [0,1],
  [ (0,PrpF p), (1,Neg $ (PrpF p)) ],
  [ (0,id    ), (1,id             ) ], actrel ), 0 )
  where
    actrel = [ (j,[[0],[1]]) | j <- (listeners model)    ]
          ++ [ (j,[ [0,1] ]) | j <- (nonlisteners model) ]

announceEqualNAction, announceNotEqualNAction :: CryptoM -> Prp -> Integer -> Action
announceEqualNAction model p n = ( ( [0,1],
  [ (0,Equal (PrpE p) (Nmbr n)), (1,Neg $ Equal (PrpE p) (Nmbr n)) ],
  [ (0,id                     ), (1,id                           ) ], actrel ), 0 )
  where
    actrel = [ (j,[[0],[1]]) | j <- (listeners model)    ]
          ++ [ (j,[ [0,1] ]) | j <- (nonlisteners model) ]

announceEqualPAction, announceNotEqualPAction :: CryptoM -> Prp -> Prp -> Action
announceEqualPAction model p q = ( ( [0,1,2],
  [ (0,Conj[PrpF p,PrpF q,Equal (PrpE p) (PrpE q)]),
    (1,Conj[Neg (PrpF p),Neg (PrpF q)]),
    (2,Top) ],
  [ (0,id), (1,addPC), (2,id) ], actrel ), 0 )
  where
    addPC  = \(fcts,ls,regs,pc,nc) -> (fcts,ls,regs,(p,q):pc,nc)
    actrel = [ (j,[[0,1],[2]]) | j <- (listeners model)    ]
          ++ [ (j,[ [0,1,2] ]) | j <- (nonlisteners model) ]

announceNotEqualNAction model p n = ( ( [0,1],
  [(0,Conj [PrpF p, Neg $ Equal (PrpE p) (Nmbr n)]),
   (1,Neg (PrpF p)),
   (2,Top) ],
  [(0,id), (1,exclN), (2,id) ], actrel ), 0 )
  where
    exclN  = \(fcts,ls,regs,ncs,pcs) -> (fcts,ls, map change regs,ncs,pcs)
    change (prp,reg) = if (prp == p) then (prp,without reg n) else (prp,reg)
    actrel = [ (j,[[0,1],[2]]) | j <- (listeners model)    ]
          ++ [ (j,[ [0,1,2] ]) | j <- (nonlisteners model) ]

announceNotEqualPAction model p q = ( ( [0,1,2],
  [(0,Conj[PrpF p, PrpF q, Neg $ Equal (PrpE p) (PrpE q)]),
   (1,Neg(Conj[PrpF p,PrpF q])),
   (2,Top)],
  [(0,id), (1,addNC), (2,id)], actrel ), 0 )
  where
    addNC  = \(fcts,ls,regs,pcs,ncs) -> (fcts,ls,regs,pcs,((p,q):ncs))
    actrel = [ (j,[[0,1],[2]]) | j <- (listeners model)    ]
          ++ [ (j,[ [0,1,2] ]) | j <- (nonlisteners model) ]

testAction :: CryptoM -> Form -> Action
testAction model form = ( ( [0,1],
  [ (0,form), (1,Neg form) ],
  [ (0,id  ), (1,id      ) ], actrel ), 0 )
  where
    actrel = [ (j,[[0,1]]) | j <- (agents model) ]
```

To allow for easier benchmarking of our implementation later on, we make the action structure for $p \overset{i}{\leftarrow} E$ slightly more general: We add the register size as an additional parameter in the command `CreateSized`. The command `Create` is then just the instance of the former using the globally fixed `registersize`.

Also note that we now set the lower bound of registers to 0 instead of 1 as we did in the guessing games. This is particularly useful in combination with modular arithmetic.

```haskell
createSizedAction :: CryptoM -> Prp -> Agent -> Exp -> Integer -> Action
createSizedAction model p i e regmax = ( ( [0,1],
  [ (0, pre    ), (1, pre    ) ],
  [ (0, addFct), (1, addReg) ],
  actrel ), 0 )
  where
    assSet = allRelevantAss model (propsInExp e)
    ass    = if (assSet /= []) then (head assSet) else error "No assignment."
    n      = evalEAss ass e
    pre    = G (Neg (PrpF p))
    addFct = \(fcts,ls,reg,pc,nc) -> (p:fcts,ls,(p,(n,n      ,[ ])):reg,pc,nc)
    addReg = \(fcts,ls,reg,pc,nc) -> (  fcts,ls,(p,(0,regmax,[n])):reg,pc,nc)
    others = delete i (agents model)
    actrel = [ (i,[[0],[1]]) ] ++ [ (j,[[0,1]]) | j <- others ]
```

Now that all actions are defined we only need to link the commands to them.

```haskell
update :: CryptoM -> Com -> CryptoM
update model (com1 :- com2) = update (update model com1) com2
update model (Open a)       = productUpdate model (openAction model a)
update model (Close a)      = productUpdate model (closeAction model a)
update model (Announce p)   = productUpdate model (announceAction model p)
update model (Test form)    = productUpdate model (testAction model form)
update model (Create p i e) = update model (CreateSized p i e registersize)
update model (CreateSized p i e regmax) =
  productUpdate model (createSizedAction model p i e regmax)
update model (AnnounceEqual p (Nmbr n)) =
  productUpdate model (announceEqualNAction model p n)
update model (AnnounceEqual p (PrpE q)) =
  productUpdate model (announceEqualPAction model p q)
update model (AnnounceNotEqual p (Nmbr n)) =
  productUpdate model (announceNotEqualNAction model p n)
update model (AnnounceNotEqual p (PrpE q)) =
  productUpdate model (announceNotEqualPAction model p q)
update _ _ = error "Update is not defined for this expression."
```

The following function is useful for testing chains of commands. It updates a given model with a list of commands, showing the size of the model and all registers after every step. After the last command it also evaluates a given formula on the final model.

```haskell
stepwiseUpdateEval :: CryptoM -> [Com] -> Form -> String
stepwiseUpdateEval model []     f = (show f) ++ " is " ++ (show $ eval model f)
stepwiseUpdateEval model (x:xs) f = "After "++(show x)++":"
  ++"\n size="++(show $ size nextm)
  ++"\n regs="++(show $ map (\(p,reg) -> (p,(head $ reg2lst reg))) (registers nextm))
  ++"\n\n"++(stepwiseUpdateEval nextm xs f)
  where nextm = update model x
```

### 3.6.7 Monte Carlo Evaluation

Checking all statements with regard to all (relevant) possible assignments is not very efficient. One could even say that our implementation as it is until here gives up the original idea of what it means to know a number because checking all assignments only differs notationally from using a many-worlds approach. In fact our models and assignments are just encodings of much larger ordinary Kripke models and our model checker implicitly unravels the encoded models.

Fortunately, we can trade absolute certainty for the amount of work. We implement the ideas from Section 3.5 and define Monte Carlo algorithms to evaluate formulas. While there is no need to revise the evaluation of expressions, we also have to rewrite the evaluation of commands because otherwise the preconditions of actions would still be checked using the normal methods. We fix a very small number of randomly picked assignments to be used in every step. This variable is used globally throughout the remaining code.

```haskell
assAmount :: Integer
assAmount = 2
```

### Pseudo-randomly picking relevant assignments

First we have to come up with a way to pick assignments randomly. In order to avoid using monads we fix a set of 10000 random numbers. The module `RAND` provides the list `myRandSeed10000`. For tests where the desired number of assignments plus the highest `prpIndex` reaches more than 10000, we repeat the set under the map $n+$.

```haskell
myRandSeed :: [Integer]
myRandSeed = concat $ [ map (n+) myRandSeed10000 | n <- [0..] ]
```

Using this seed and given another integer we pseudo-randomly generate a partial relevant assignment. Note that we do not check for consistency yet but for now ignore the positive and negative constraints.

```haskell
rndRelAssSingle :: CryptoM -> [Prp] -> Integer -> Assignment
rndRelAssSingle model givenprops seed = [ (p, pickFor p) | p <- props ]
  where
    modprops   = map fst (registers model)
    props      = filter (\x -> (elem x modprops)) givenprops
    domFor p   = reg2lst (apply (registers model) p)
    lenFor p   = fromIntegral $ length (domFor p)
    seedFor p = mod (seed*(myRandSeed!!(fromIntegral $ (prpIndex p+seed)))) (lenFor p)
    pickFor p = (domFor p)!!(fromIntegral $ seedFor p)
```

In formal notation, `rndRelAssSingle` works as follows. As input it takes a pointed model $\mathcal{M}, w$ and an additional integer seed $n \geq 1$. Furthermore it employs a constant infinite list $\mathbf{M}$ of random numbers which in our implementation is given by `myRandSeed`. For any $p$ let $\mathbf{idx}(p)$ be the index of $p$ as implemented by `prpIndex` on page 56. For any ordered set of numbers $A$ we write $A[k]$ for its $k$-th element. In Haskell the notation is `A!!k`. We write $\mathcal{H}_w(p)$ for the set of values that $p$ can take at $w$. Then the output of `rndRelAssSingle` applied to $\mathcal{M}$ and $n$ is the assignment function defined by

$$V_n(p) := \mathcal{H}_w(p)\,[\; n \cdot \mathbf{M}[\mathbf{idx}(p) + n] \mod |\mathsf{dom}p|\;]$$

The following function then pseudo-randomly generates a given number of assignments, incrementing the integer seed $n \geq 1$ in every step. It is now that we also make sure they are consistent with the constraints. Note that there might be duplicates.

```haskell
rndRelAss :: CryptoM -> [Prp] -> Integer -> [Assignment]
rndRelAss model props amount = take (fromIntegral amount) consSet
  where
    pcs = posConstraints model
    ncs = negConstraints model
    consSet = filter (consistent pcs ncs) fullSet
    fullSet = [ rndRelAssSingle model props seed | seed <- [1..] ]
```

It is clear that myRandSeed, our list of random numbers is finite and gets repeated. Still, as the sequential seed keeps increasing, different numbers will be picked for different propositions and thus the assignments will not repeat after 10000 steps but much later.

## How random are our assignments?

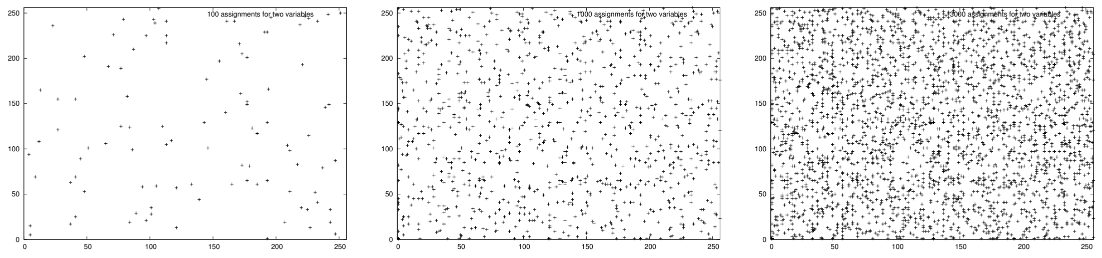
We should check that rndRelAss really generates random assignments. To do so we plot the values that get assigned to two variables which each have a a register allowing any number (up to the registersize $n$) except 0.

```haskell
rndTestModel :: Integer -> CryptoM
rndTestModel n = ( Mo [0] [ (alice,[[0]]) ]
  [ (0,([],[],[ ((P 0),(0,n,[0])), ((Q 0),(0,n,[0]))  ],[],[])) ] 0 )

rndTestRun :: Integer -> Integer -> IO ()
rndTestRun rs amount = plotListStyle arg1 arg2 coords
  where
    arg1 = [EPS filename, XRange (0,fromIntegral rs), YRange (0,fromIntegral rs)]
    arg2 = (defaultStyle {plotType = Points, lineSpec = CustomStyle [LineTitle ((show
        amount)++" assignments for two variables")]})
    coords   = map (\list -> (head list, head $ tail list)) numbers
    numbers  = map (\list -> (map snd list)) assSet
    assSet   = rndRelAss model [(P 0),(Q 0)] amount
    model    = (rndTestModel rs)
    filename = "img/rndtest_"++(show rs)++"_regsize_"++(show amount)++"ass.eps"

rndTestAllRuns :: IO ()
rndTestAllRuns = do
  rndTestRun (2^(8::Int)) 100
  rndTestRun (2^(8::Int)) 1000
  rndTestRun (2^(8::Int)) 3000
```

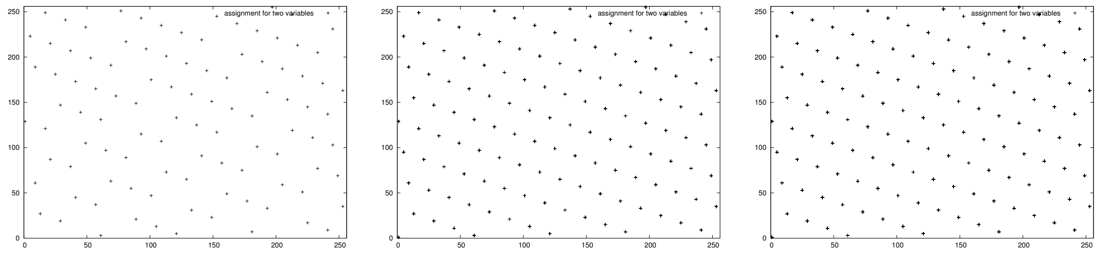
Now we call rndTestAllRuns to generate the following three plots on which we can see that the assignments are fairly randomly picked:



100, 1000 and 3000 random assignments for a registersize of $2^8$.

In contrast, if we use the following very similar line in the definition of rndRelAssSingle on page 64, a clear pattern becomes visible:

```haskell
seedFor p = mod (seed * myRandSeed!!(fromIntegral $ (prpIndex p))) (lenFor p)
```



100, 1000 and 3000 not-so-random assignments for a registersize of $2^8$.

We can also relate our test for "real randomness" to the usage of pseudorandomness as in cryptography (See for example [KL08, p. 70]): An adversary could easily distinguish these assignments from truly random ones that e.g. were obtained by rolling a die. Moreover, this would enable her to systematically come up with models and formulas for which our Monte Carlo algorithm would always return wrong results.

## Monte Carlo Evaluation

The following implements Definitions 73 and 74.

```
evalAssMC :: CryptoM -> Assignment -> Form -> Bool

evalAssMC model _   (PrpF prp)    = elem prp (facts model)
evalAssMC _     ass (Equal a b)   = (evalEAss ass a) == (evalEAss ass b)
evalAssMC _     ass (Prime e)     = isPrime (evalEAss ass e)
evalAssMC _     ass (Coprime a b) = (gcd (evalEAss ass a) (evalEAss ass b) == 1)
evalAssMC _     _   Top           = True
evalAssMC model ass (Neg form)    = not (evalAssMC model ass form)
evalAssMC model ass (Conj forms)  = and (map (evalAssMC model ass) forms)

evalAssMC model _   (K i form)    = and results
  where results = map (\v -> (evalMCAt model v form==Just True)) (reachableBy model i)

evalAssMC model _   (G form)      = and results
  where results = map (\v -> (evalMCAt model v form==Just True)) (states $ genMin
      model)

evalAssMC model _   (L agent)     = elem agent (listeners model)

evalAssMC model ass (Com com form) =
  if (assSet /= [])
    then and results
    else error ("No compatible assignments!")
  where
    newmodel = updateMC model com
    assSet   = filter (subset ass) (rndRelAss newmodel props assAmount)
    props    = nub $ propsInCom com ++ propsInForm form
    chkFct   = (\newass -> evalAssMC newmodel newass form)
    results  = map chkFct assSet
    subset assA assB = all (\x -> (apply assA x == apply assB x)) (map fst assA)

evalMC :: CryptoM -> Form -> Maybe Bool
evalMC model formula =
    if (and results)
      then
        Just True
      else
        if (all (\x -> x==False) results)
          then Just False
          else Nothing
  where
    results = [ evalAssMC model ass formula | ass <- assSet ]
    assSet  = rndRelAss model (propsInForm formula) assAmount

evalMCAt :: CryptoM -> State -> Form -> Maybe Bool
evalMCAt (Mo sts rel val _) nc f = evalMC (Mo sts rel val nc) f

evalMCAtAll :: CryptoM -> Form -> [(State,Maybe Bool)]
evalMCAtAll m form = zip (states m) (map (\w -> evalMCAt m w form) (states m))
```

## Monte Carlo Updates

Now two generalizations of our previous code come in handy. First, we can define the Monte-Carlo update simply by replacing `evalAt` with `evalMCAt`.

```haskell
productUpdateMC :: CryptoM -> Action -> CryptoM
productUpdateMC = productUpdateWithEvAtFct evalMCAt
```

Second, because we specified the action structures separately from the interpretation of commands we do not have to repeat them here. We can simply run `productUpdateMC` with the the same actions.

```haskell
updateMC :: CryptoM -> Com -> CryptoM
updateMC model (com1 :- com2) = updateMC (updateMC model com1) com2
updateMC model (Open a)       = productUpdateMC model (openAction model a)
updateMC model (Close a)      = productUpdateMC model (closeAction model a)
updateMC model (Announce p)   = productUpdateMC model (announceAction model p)
updateMC model (Test form)    = productUpdateMC model (testAction model form)
updateMC model (Create p i e) = updateMC model (CreateSized p i e registersize)
updateMC model (CreateSized p i e regmax) =
  productUpdateMC model (createSizedAction model p i e regmax)
updateMC model (AnnounceEqual p (Nmbr n)) =
  productUpdateMC model (announceEqualNAction model p n)
updateMC model (AnnounceEqual p (PrpE q)) =
  productUpdateMC model (announceEqualPAction model p q)
updateMC model (AnnounceNotEqual p (Nmbr n)) =
  productUpdateMC model (announceNotEqualNAction model p n)
updateMC model (AnnounceNotEqual p (PrpE q)) =
  productUpdateMC model (announceNotEqualPAction model p q)
updateMC _ _ = error "Update is not defined for this expression."
```

We also implement the function `stepwiseUpdateEval` from page 63 again with Monte Carlo methods.

```haskell
stepwiseUpdateEvalMC :: CryptoM -> [Com] -> Form -> String
stepwiseUpdateEvalMC model []     f = (show f) ++ " is " ++ (show $ evalMC model f)
stepwiseUpdateEvalMC model (x:xs) f = "After "++(show x)++":"
  ++"\n size = "++(show $ size nextmodel)
  ++" // regs = "++(show $ map (\(p,reg) -> (p,(head $ reg2lst reg))) (registers
      nextmodel))
  ++"\n\n"++(stepwiseUpdateEvalMC nextmodel xs f)
  where nextmodel = updateMC model x
```

## 3.6.8 Visualization

The following helper functions produce LaTeX-strings from our new valuations.

```haskell
eclShowProp :: Prp -> String
eclShowProp prp = replace (replace (show prp) " 0" "") " " "_"

eclShowConstraints :: [Constraint] -> [Constraint] -> String
eclShowConstraints []  []  = ""
eclShowConstraints pcs ncs =
  sepBy (positives ++ negatives) " \\text{ and } "
    where
      positives = map (niceconSingle "   =   ") pcs
      negatives = map (niceconSingle " \\neq ") ncs
      niceconSingle b (prpA,prpB) = " $ " ++ (eclShowProp prpA) ++ b ++ (eclShowProp
          prpB) ++ " $ "

eclShowVal :: Valuation -> String
eclShowVal (fcts,lstnrs,regs,pcs,ncs) =
  sepBy nonEmptyItems newline
    where
      nonEmptyItems = filter (\x -> x/="") items
      items = [nicefacts, nicelisteners, nicereg, eclShowConstraints pcs ncs]
      nicelisteners = sepBy (map show lstnrs) ", "
      nicefacts     = sepBy (map show fcts) ", "
      nicereg       = concat $ map eclShowReg regs
```

```

eclShowReg :: (Prp,Register) -> String
eclShowReg (prp,(low,high,excl)) =
  if (low /= high)
    then
      " $ " ++ (show low) ++ " \\leq " ++ (show prp) ++ " \\leq " ++ (show high) ++ "
          \\text{ and }"++(show prp)++"\\not\\in \\{" ++ (concat $ map show excl) ++ "
          \\} $ " ++ newline
    else
      " $ " ++ (show prp) ++ "=" ++ (show low) ++  " $ " ++ newline

eclInfo :: CryptoM -> String
eclInfo model = begintab ++ "Agents: " ++ (sepBy (map show (agents model)) ", ") ++
    endtab
```

The few lines below employ KRIPKEVIS which can be found in the appendix on page 91.

```
eclTexModel :: CryptoM -> String -> IO String
eclTexModel model = texModel show show eclShowVal (eclInfo model) visModel
  where
    (Mo sts rel val cur) = model
    visModel = (VisModel sts rel val cur)

eclDispModel :: CryptoM -> IO String
eclDispModel model = dispModel show show eclShowVal (eclInfo model) visModel
  where
    (Mo sts rel val cur) = model
    visModel = (VisModel sts rel val cur)
```

# Chapter 4

# Applications

## 4.1 Small Examples

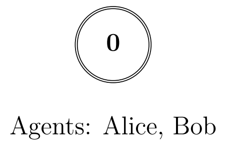
```
module ECLEXAMPLE where
import ECL
```

### 4.1.1 Alice and Bob have Secrets

We start with the blissful ignorance model for Alice and Bob.

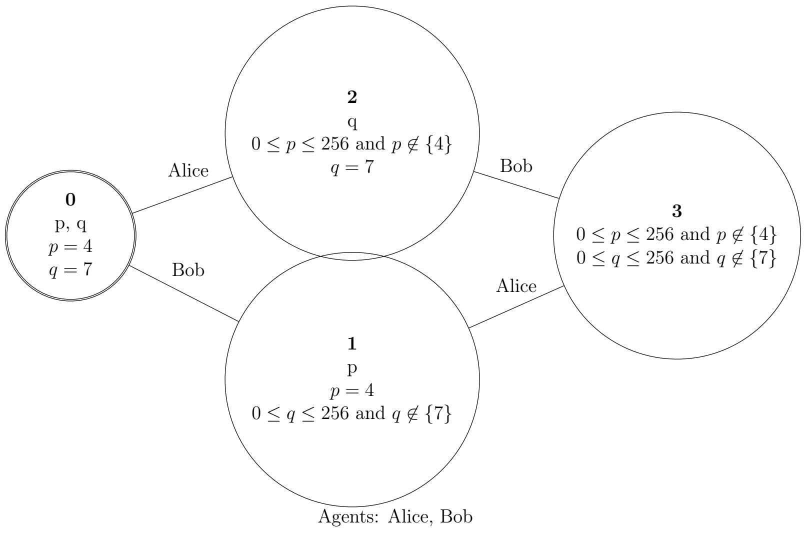
```
m0 :: CryptoM
m0 = cm0for [alice,bob]
```

```
*ECLEXAMPLE> m0
(Mo [0]
    [(Alice,[[0]]),(Bob,[[0]])]
    [(0,([],[],[],[],[]))]
    0 )
```



Now we let both Alice and Bob come up with a secret number and call the result `m1`.

```
m1 :: CryptoM
m1 = update m0 (Create (Q 0) alice (Nmbr 7) :- Create (P 0) bob (Nmbr 4))
```


0
p, q
$p = 4$
$q = 7$
2
q
$0 \leq p \leq 256$ and $p \notin \{4\}$
$q = 7$
1
p
$p = 4$
$0 \leq q \leq 256$ and $q \notin \{7\}$
3
$0 \leq p \leq 256$ and $p \notin \{4\}$
$0 \leq q \leq 256$ and $q \notin \{7\}$
Alice
Bob
Bob
Alice
Agents: Alice, Bob

### 4.1.2 Tautologies about Undefined Statements

The following example shows that our agents also know tautologies if they do not know the values of variables used to formulate them. In the model `m1` (from the previous example) both Alice and Bob do not know that $p \neq q$.

```
*ECLEXAMPLE> eval m1 (K alice (Neg $ Equal (PrpE (P 0)) (PrpE (Q 0))))
Just False
*ECLEXAMPLE> eval m1 (K bob (Neg $ Equal (PrpE (P 0)) (PrpE (Q 0))))
Just False
```

We can also see that in the three non-actual worlds of `m1` both the sentence $p = q$ and its negation $p \neq q$ are undefined:

```
*ECLEXAMPLE> evalAtAll m1 (Equal (PrpE (P 0)) (PrpE (Q 0)))
[(0,Just False),(2,Nothing),(1,Nothing),(3,Nothing)]
*ECLEXAMPLE> evalAtAll m1 (Neg $ Equal (PrpE (P 0)) (PrpE (Q 0)))
[(0,Just True),(2,Nothing),(1,Nothing),(3,Nothing)]
```

Still, the disjunction of the equality and its negation is a tautology and true everywhere. Furthermore, both Alice and Bob know that it is true even though they do not know about $p$ and $q$ respectively.

```
equality, tautology :: Form
equality  = Equal (PrpE (P 0)) (PrpE (Q 0))
tautology = disj [ equality, Neg equality ]
```

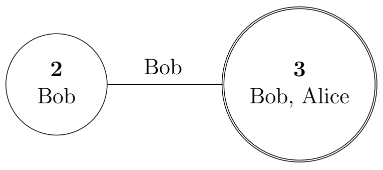
```
*ECLEXAMPLE> evalAtAll m1 tautology
[(0,Just True),(2,Just True),(1,Just True),(3,Just True)]
*ECLEXAMPLE> eval m1 (K bob tautology)
Just True
*ECLEXAMPLE> eval m1 (K alice tautology)
Just True
```

### 4.1.3 Knowing Who is Listening

The following model shows that two agents can but do not have to know if the other agent is listening. In particular this does not contradict the self-awareness constraint.

```
mL :: CryptoM
mL = update (cmOfor [alice,bob]) (Open alice :- Open bob)
```

```
*ECLEXAMPLE> eval mL (K alice (L bob))
Just True
*ECLEXAMPLE> eval mL (K bob (L alice))
Just False
```

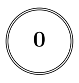


Agents: Alice, Bob

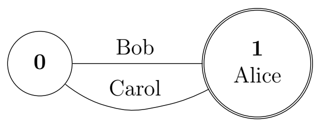
This example already suggests that the order in which agents start listening is important. In particular all agents but one have to be called for attention twice in order to let everyone know that everyone is listening. However, once we reach a situation where everyone is listening and this is common knowledge according to definition 17 the model we obtain is bisimilar to a one-world model. The following sequence of updates shows this effect of our built-in optimization.

```
mAttention0, mAttention1, mAttention2, mAttention3, mAttention4, mAttention5 ::
    CryptoM

mAttention0 = cm0for [alice,bob,carol]
mAttention1 = update mAttention0 (Open alice)
```

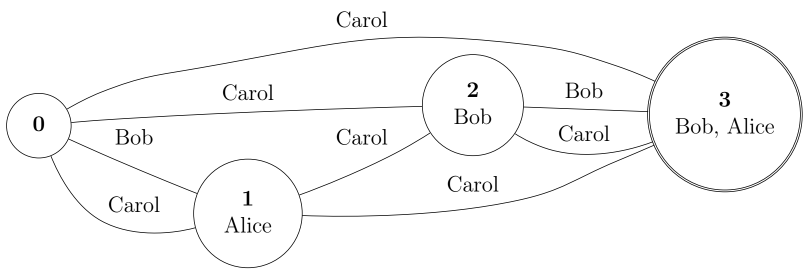


Agents: Alice, Bob, Carol



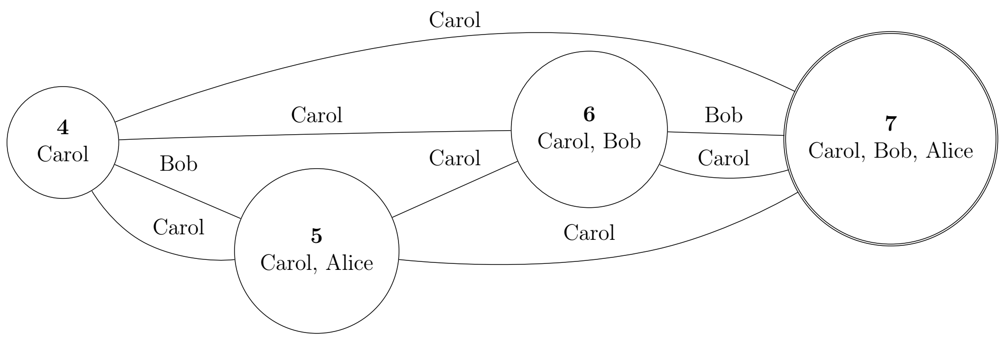
Agents: Alice, Bob, Carol

```
mAttention2 = update mAttention1 (Open bob)
```



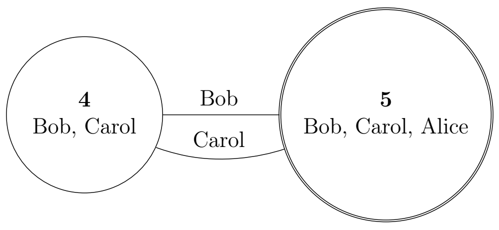
Agents: Alice, Bob, Carol

```
mAttention3 = update mAttention2 (Open carol)
```

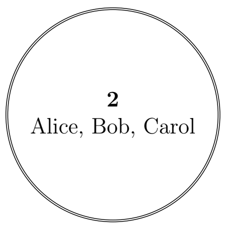


Agents: Alice, Bob, Carol

```
mAttention4 = update mAttention3 (Open bob)
mAttention5 = update mAttention4 (Open alice)
```



Agents: Alice, Bob, Carol



Agents: Alice, Bob, Carol

While the last two updates do not change facts, they still induce epistemic change. Only after the last update Bob and Carol know that Alice is listening. More generally, we have the following theorem.

**Theorem 76.** *For any set of agents* $\{a_1, \dots, a_n\} = G \subseteq I$ *the sequence of* ECL*-commands*

$$\mathbf{Open}_{a_1}; \mathbf{Open}_{a_2}; \dots; \mathbf{Open}_{a_{n-1}}; \mathbf{Open}_{a_n}; \mathbf{Open}_{a_{n-1}} \dots; \mathbf{Open}_{a_2}; \mathbf{Open}_{a_1}$$

*generates common knowledge among $G$ that everyone in $G$ is listening.*

*Proof.* Note that **Open** commands do not have a precondition and therefore never fail. Suppose after running the command there would be a formula of the shape $K_{i_1} \dots K_{i_m} L_n$ for some $n \in G$ that is false. But $L_n$ has to be true because $\mathbf{Open}_n$ was executed at least once. Hence suppose that $K_{i_m} L_n$ is false. Then there is an $i_m$-reachable world where $L_n$ is false. But this cannot be because the sequence contains an $\mathbf{Open}_n$ command after an $\mathbf{Open}_{i_m}$ command. Iterating this reasoning for $m$ steps leads to a contradiction. □

### 4.1.4 Monte Carlo Failure

In section 3.5 we gave an example for a model and a formula which a Monte Carlo algorithm most probably judges wrong. The following shows that our implementation indeed falls into this trap.

```
*ECL> eval (update (cmOfor [alice,bob]) (Create (P 0) bob (Nmbr 4))) (K bob (Neg (
    Equal (PrpE (P 0)) (Nmbr 8))))
Just True
*ECL> eval (update (cmOfor [alice,bob]) (Create (P 0) bob (Nmbr 4))) (K alice (Neg (
    Equal (PrpE (P 0)) (Nmbr 8))))
Just False
*ECL> evalMC (update (cmOfor [alice,bob]) (Create (P 0) bob (Nmbr 4))) (K alice (Neg )
     Equal (PrpE (P 0)) (Nmbr 8))))
Just True
```

### 4.1.5 Generating Drawings

The following code generates all drawings used in this section.

```
main :: IO ()
main = do
  s1 <- eclTexModel m0 "ECLm0"
  putStrLn s1
  s2 <- eclTexModel m1 "ECLm1"
  putStrLn s2
  s3 <- eclTexModel mAttention0 "mAttention0"
  putStrLn s3
  s4 <- eclTexModel mAttention1 "mAttention1"
  putStrLn s4
  s5 <- eclTexModel mAttention2 "mAttention2"
  putStrLn s5
  s6 <- eclTexModel mAttention3 "mAttention3"
  putStrLn s6
  s7 <- eclTexModel mAttention4 "mAttention4"
  putStrLn s7
  s8 <- eclTexModel mAttention5 "mAttention5"
  putStrLn s8
  putStrLn "Done."
```

## 4.2 The Diffie-Hellman key exchange

### 4.2.1 Definition

Whitfield Diffie and Martin Hellman revolutionized the field of cryptography with their proposal for public key encodings in [DH76]. Here is their famous protocol for establishing a shared secret key over an insecure channel:

**Definition 77** (Diffie-Hellman Key Exchange Over Insecure Channel)**.**

1. *Alice and Bob agree on a prime $p$ and a base $g < p$ such that $g$ and $p-1$ are coprime.*
2. *Alice picks a secret $a$ and sends $g^a \bmod p = A$ to Bob.*
3. *Bob picks a secret $b$ and sends $g^b \bmod p = B$ to Alice.*
4. *Alice calculates $k = B^a \bmod p$.*
5. *Bob calculates $k = A^b \bmod p$.*
6. *They now have a shared key $k$ because $k = (g^a)^b = (g^b)^a \bmod p$.*

The established key $k$ can then be used by Alice and Bob to encrypt and decrypt messages before and after they are sent via the insecure channel, respectively. Let $p$ be the prime that Alice and Bob have agreed on, and let $k$ be their shared key. Then a message represented as the number $m$ is encoded as

$$m \times k \bmod p.$$

Such messages can efficiently be decoded by both Alice and Bob as follows. Alice knows $p$, $k$, $g^b$ and $a$. She decodes cipher $c$ with

$$c \times (g^b)^{(p-1)-a} \bmod p.$$

This yields the correct $m$ because of Fermat's Little Theorem (see below). We have:

$$(g^a)^{(p-1)-b} = g^{a((p-1)-b)} = g^{a(p-1)} \times g^{-ab} = (g^{p-1})^a \times g^{-ab} \overset{\text{Fermat}}{=} 1^a \times g^{-ab} = g^{-ab} \bmod p$$

And therefore:

$$c \times (g^a)^{(p-1)-b} = (m \times g^{ab}) \times g^{-ab} = m \times (g^{ab} \times g^{-ab}) = m \bmod p.$$

Similarly, Bob knows $p$, $k$, $g^a$ and $b$ and can decode a cipher $c$ with $c \times (g^a)^{(p-1)-b} \bmod p$.

**Theorem 78** (Fermat's Little Theorem)**.** *If $p$ is prime, then for every $a$ such that $1 \leq a < p$ we have $a^{p-1} = 1 \bmod p$.*

The language $\mathcal{L}_{\mathsf{ECL}}$ as given in Definition 65 allows us to formulate the entire Diffie-Hellman key exchange and its goal as a register language protocol. For this section we fix a set of three agents Alice, Bob and Eve. The parameters $g, p, N, M$ are the public base and prime and the private numbers of Alice and Bob, respectively.

**Definition 79** (Diffie-Hellman Key Exchange in ECL)**.** *For any $g, p, N, M \in \mathbb{N}$ such that $p$ is prime, $g \in [1..p]$ and $g$ and $(p-1)$ are coprime, let $\mathbf{DH}_{g,p,N,M}$ abbreviate the following sequence of commands.*

$$\begin{aligned}
&q_1 \overset{a}{\leftarrow} N \text{ ; } r_1 \overset{a}{\leftarrow} (g^{q_1} \bmod p) \text{ ;}\\
&\mathbf{Open}_b \text{ ; } !r_1 \text{ ; } \mathbf{Close}_b \text{ ;}\\
&q_2 \overset{b}{\leftarrow} M \text{ ; } r_2 \overset{b}{\leftarrow} (g^{q_2} \bmod p) \text{ ;}\\
&\mathbf{Open}_a \text{ ; } !r_2 \text{ ; } \mathbf{Close}_a \text{ ;}\\
&s_1 \overset{a}{\leftarrow} {r_2}^{q_1} \bmod p \text{ ;}\\
&s_2 \overset{b}{\leftarrow} {r_1}^{q_2} \bmod p
\end{aligned}$$

*The goal of the key-exchange $\psi_{\mathbf{DH}}$ consists of three conjuncts, namely that the values of $s_1$ and $s_2$ are equal, Alice and Bob know them and Eve does not.*

$$\psi_{\mathbf{DH}} := (s_1 = s_2) \wedge (K_a s_1 \wedge K_b s_2) \wedge (\neg K_e s_1 \wedge \neg K_e s_2)$$

*The claim that a pointed model $\mathcal{M}, m$ allows a successful run of the key-exchange with the parameters $g$, $p$, $N$ and $M$ is now given by $\mathcal{M}, m \vDash \langle \mathbf{DH}_{g,p,N,M} \rangle \psi_{\mathbf{DH}}$.*

### 4.2.2 Implementation

```haskell
import Data.Numbers.Primes
import Criterion.Main
import ECL
```

To run the D-H key exchange in our implementation we first define the starting model, the command and the goal of the protocol.

```haskell
dhStart :: CryptoM
dhStart = update (cmOfor [alice,bob,eve]) (Open eve)

dhCommandList :: (Integer,Integer,Integer,Integer) -> Integer -> [Com]
dhCommandList (popen,gopen,asecret,bsecret) regmax = [
  CreateSized (Q 1) alice (Nmbr asecret) regmax,
  CreateSized (R 1) alice (PowerMod (Nmbr gopen) (PrpE (Q 1)) (Nmbr popen)) regmax,
  (Open bob), (Announce (R 1)), (Close bob),
  CreateSized (Q 2) bob (Nmbr bsecret) regmax,
  CreateSized (R 2) bob (PowerMod (Nmbr gopen) (PrpE (Q 2)) (Nmbr popen)) regmax,
  (Open alice), (Announce (R 2)), (Close alice),
  CreateSized (S 1) alice (PowerMod (PrpE (R 2)) (Nmbr asecret) (Nmbr popen)) regmax,
  CreateSized (S 2) bob (PowerMod (PrpE (R 1)) (Nmbr bsecret) (Nmbr popen)) regmax
  ]

dhCommand :: (Integer,Integer,Integer,Integer) -> Integer -> Com
dhCommand (popen,gopen,asecret,bsecret) regmax =
  if and [ isPrime popen, gopen <= popen, gcd (popen-1) gopen == 1 ]
    then lst2cmd $ dhCommandList (popen,gopen,asecret,bsecret) regmax
    else error ("Invalid Diffie-Hellman parameters!")

dhGoal :: Form
dhGoal = Conj [
  Equal (PrpE (S 1)) (PrpE (S 2)),
  K alice (PrpF (S 1)), K bob (PrpF (S 2)),
  Neg (K eve (PrpF (S 1))), Neg (K eve (PrpF (S 2)))
  ]
```

We now call `dhCommand` with the public parameters $p$, $g$ and the private keys $a$ and $b$ for Alice and Bob respectively. We provide a concrete example and a set which contains all possible runs of the protocol. The latter should be used carefully – depending on the `registersize` it becomes unmanageably big.

```
dhSample :: (Integer,Integer,Integer,Integer)
dhSample = (23,5,6,15)
allPrimes :: [Integer]
allPrimes = filter (registersize >) (take (fromIntegral registersize) primes)
dhAllSamples :: [(Integer,Integer,Integer,Integer)]
dhAllSamples = [ (p,g,a,b) | p <- allPrimes, g<-allN, a<-allN, b<-allN ]
  where allN = [0..registersize]
```

We can now run the protocol and check the result.

```
*Main> eval dhStart (Com (dhCommand dhSample registersize) dhGoal)
Just True
*Main> evalMC dhStart (Com (dhCommand dhSample registersize) dhGoal)
Just True
```

By varying `registersize` and timing the computation we can observe the difference between the normal and the Monte Carlo evaluation methods. The following table shows how many seconds the commands take to complete in `ghci` with the option `:set +s`.

| `registersize` | Normal | Monte Carlo |
|---|---|---|
| $2^{10}$ | 4.07 | 5.42 |
| $2^{11}$ | 6.72 | 5.50 |
| $2^{12}$ | 12.02 | 5.53 |

For a more thorough runtime analysis we use the `criterion` library by Bryan O'Sullivan which can be found at `github.com/bos/criterion`. The `main` routine listed below is a benchmark measuring the runtime of `eval` and `evalMC` on registers from 8 to 16 bit.

```
main :: IO ()
main = defaultMain $ concat [ [
  bench (show n) $ nf (eval   dhStart) (Com (dhCommand dhSample (2^n)) dhGoal),
  bench (show n) $ nf (evalMC dhStart) (Com (dhCommand dhSample (2^n)) dhGoal)
  ] | n <- [8..(16::Integer)] ]
```

```
ghc -O --make DH.lhs
./DH -g -s 20 -u DH_results.csv
```

We compile the program using the `-O` switch to activate all optimizations that `ghc` offers. Hence the resulting program `DH` is much faster than interactively testing the module with `ghci`. A sample of the average results that get written into `DH_results.csv` is listed in the following table. We can observe that for a small `registersize` the normal implementation is faster. This is to be expected because it does not have to spend time on the generation of random assignments. However, we can see again that from $2^{11}$ onwards the Monte Carlo algorithm is faster while the normal one roughly doubles its runtime whenever `registersize` is doubled.

| `registersize` | Normal | Monte Carlo |
|---|---|---|
| $2^{8}$ | 1.07 | 2.74 |
| $2^{9}$ | 1.36 | 2.82 |
| $2^{10}$ | 2.13 | 3.41 |
| $2^{11}$ | 3.59 | 3.24 |
| $2^{12}$ | 5.17 | 2.8 |
| $2^{13}$ | 11.56 | 3.28 |
| $2^{14}$ | 22.66 | 3.57 |
| $2^{15}$ | 44.44 | 4.1 |
| $2^{16}$ | 81.26 | 3.52 |

## 4.3 Man-in-the-Middle vs Diffie-Hellman

### 4.3.1 Definition

It is essential that at the end of the DH protocol we have $s_1 = s_2$ and the protocol is secure against passive eavesdroppers like Eve. But suppose now that a malicious agent called Mallory can not only obtain but also alter the messages between Alice and Bob. The following table shows a Man-in-the-Middle (MitM) attack as in [DY83]. Mallory can trick Alice and Bob into generating two separate keys with him instead of each other. We write $x \leftarrow$ for randomly choosing a secret. The symbols $\xrightarrow{x}$ and $\xleftarrow{x}$ indicate that an agent sends $x$ to the neighboring agent right or left respectively.

**Definition 80** (Man-in-the-Middle Attack on Diffie-Hellman Key Exchange)**.**

| ***Alice*** | ***Mallory*** | ***Bob*** |
|---|---|---|
| $q_1 \leftarrow;\ r_1 = g^{q_1} \mod p$ | | |
| $\xrightarrow{r_1}$ | $q_1' \leftarrow,\ r_1' = g^{q_1'} \mod p$ | |
| | $\xrightarrow{r_1'}$ | $q_2 \leftarrow;\ r_2 = g^{q_2} \mod p$ |
| | $q_2' \leftarrow;\ r_2' = g^{q_2'} \mod p$ | $\xleftarrow{r_2}$ |
| | $\xleftarrow{r_2'}$ | |
| $s_1 = {r_2'}^{q_1} = g^{q_2' q_1}$ | $s_1' = {r_1}^{q_2'} = g^{q_1 q_2'};\ s_2' = {r_2}^{q_1'} = g^{q_2 q_1'}$ | $s_2 = {r_1'}^{q_2} = g^{q_1' q_2}$ |

We write $r_1'$ instead of $r_1$ after the value has been altered by Mallory, but this is supposed to be unknown to Bob. We can thus see that Alice and Bob behave exactly as in Definition 77 before, but in general we end up with $s_1 = s_1' \neq s_2' = s_2$. Any later message sent from Alice to Bob could now be $s_1'$-decrypted and $s_2'$-encrypted by Mallory.
To represent this attack in our framework we have to tackle a few questions. How can we model the interception of communication? Are the commands $\mathbf{Open}_i$ and $\mathbf{Close}_i$ expressive enough? How do we represent $r_1'$ and $r_2'$?
Our design choice for now is to model attacks as additional commands and substitutions of variables. In particular for the MitM attack we insert additional commands before the two communication lines and replace for example $s_1$ with $s_1'$ in every line afterwards. The additional parameter $O$ is the secret value used by Mallory as $q_1'$ and $q_2'$ above.

**Definition 81** (MitM Attack on Diffie-Hellman Key Exchange in ECL)**.** *For any given parameters $g, p, N, M, O \in \mathbb{N}$ such that $p$ is prime, $g \in [1..p]$ and $g$ and $(p-1)$ are coprime, let $\mathbf{DHMITM}_{g,p,N,M}$ abbreviate the following sequence of commands.*

$$\begin{gathered}
q_1 \overset{a}{\leftarrow} N\ ;\ r_1 \overset{a}{\leftarrow} (g^{q_1} \bmod p)\ ;\ \mathbf{Open}_m\ ;\ !r_1\ ;\ \mathbf{Close}_m\ ;\\
q_1' \overset{m}{\leftarrow} O\ ;\ r_1' \overset{m}{\leftarrow} (g^{q_1'} \bmod p)\ ;\ \mathbf{Open}_b\ ;\ !r_1'\ ;\ \mathbf{Close}_b\ ;\\
q_2 \overset{b}{\leftarrow} M\ ;\ r_2 \overset{b}{\leftarrow} (g^{q_2} \bmod p)\ ;\ \mathbf{Open}_m\ ;\ !r_2\ ;\ \mathbf{Close}_m\ ;\\
q_2' \overset{m}{\leftarrow} O\ ;\ r_2' \overset{m}{\leftarrow} (g^{q_2'} \bmod p)\ ;\ \mathbf{Open}_a\ ;\ !r_2'\ ;\ \mathbf{Close}_a\ ;\\
s_1 \overset{a}{\leftarrow} {r_2'}^{q_1} \bmod p\ ;\ s_2 \overset{b}{\leftarrow} {r_1'}^{q_2} \bmod p\ ;\ s_1' \overset{m}{\leftarrow} {r_1}^{q_2'} \bmod p\ ;\ s_2' \overset{m}{\leftarrow} {r_2}^{q_1'} \bmod p
\end{gathered}$$

*The goal of the attack $\psi_{\mathbf{DHMITM}}$ is again a conjunction of multiple claims. The values of $s_1$ and $s_2$ should not be the same, but they are equal to $s_1'$ and $s_2'$ which are both known by Mallory.*

$$\psi_{\mathbf{DHMITM}} := (s_1 \neq s_2 \wedge s_1 = s_1' \wedge s_2 = s_2' \wedge K_m s_1' \wedge K_m s_2')$$

*The claim that a pointed model $\mathcal{M}, m$ allows a successful run of the attack with the parameters $g$, $p$, $N$, $M$, $O$ is now given by $\mathcal{M}, m \vDash \langle \mathbf{DHMITM}_{g,p,N,M,O} \rangle \psi_{\mathbf{DHMITM}}$.*

### 4.3.2 Implementation

In the following code we represent $r'_1$ by `R 11`, $r'_2$ by `R 22` and so on. Also note that instead of creating private registers for the secret numbers we use the variables `asecret`, `bsecret` and `msecret` directly in order to keep the model size small.

```haskell
import Data.Numbers.Primes
import ECL

dhMitmStart :: CryptoM
dhMitmStart = cmOfor [alice,bob,mallory]

dhMitmCommandList :: (Integer,Integer,Integer,Integer,Integer) -> Integer -> [Com]
dhMitmCommandList (popen,gopen,asecret,bsecret,msecret) rs = [
  CreateSized (R 1) alice (PowerMod (Nmbr gopen) (Nmbr asecret) (Nmbr popen)) rs,
  Open mallory, Announce (R 1), Close mallory,
  CreateSized (R 11) mallory (PowerMod (Nmbr gopen) (Nmbr msecret) (Nmbr popen)) rs,
  Open bob, Announce (R 11), Close bob,
  CreateSized (R 2) bob (PowerMod (Nmbr gopen) (Nmbr bsecret) (Nmbr popen)) rs,
  Open mallory, Announce (R 2), Close mallory,
  CreateSized (R 22) mallory (PowerMod (Nmbr gopen) (Nmbr msecret) (Nmbr popen)) rs,
  Open alice, Announce (R 22), Close alice,
  CreateSized (S 1)  alice   (PowerMod (PrpE (R 22)) (Nmbr asecret) (Nmbr popen)) rs,
  CreateSized (S 2)  bob     (PowerMod (PrpE (R 11)) (Nmbr bsecret) (Nmbr popen)) rs,
  CreateSized (S 11) mallory (PowerMod (PrpE (R 1))  (Nmbr msecret) (Nmbr popen)) rs,
  CreateSized (S 22) mallory (PowerMod (PrpE (R 2))  (Nmbr msecret) (Nmbr popen)) rs
  ]

dhMitmCommand :: (Integer,Integer,Integer,Integer,Integer) -> Integer -> Com
dhMitmCommand (popen,gopen,asecret,bsecret,msecret) rs =
  if and [ isPrime popen, gopen <= popen, gcd (popen-1) gopen == 1 ]
    then lst2cmd $ dhMitmCommandList (popen,gopen,asecret,bsecret,msecret) rs
    else error ("Invalid Diffie-Hellman parameters!")

dhMitmSample :: (Integer,Integer,Integer,Integer,Integer)
dhMitmSample = (23,5,6,15,13)

dhMitmGoal :: Form
dhMitmGoal = Conj [
  Neg $ Equal (PrpE (S 1)) (PrpE (S 2)),
  Equal (PrpE (S 1)) (PrpE (S 11)),
  Equal (PrpE (S 2)) (PrpE (S 22)),
  K mallory (PrpF (S 11)),
  K mallory (PrpF (S 22))
  ]
```

Because the whole command takes almost two days to run with a registersize of $2^8$, we use the function `stepwiseUpdateEvalMC` from page 67 to observe the process as the updates are executed one after another. Lazy evaluation of Haskell ensures that the intermediate results are printed as soon as they are available, even though the goal formula only gets evaluated on the last model. Finally, note that we do all this using the fast Monte Carlo methods – the normal implementation is unable to cope with such huge models.

```haskell
main :: IO ()
main = do
  putStr $ stepwiseUpdateEvalMC dhMitmStart (dhMitmCommandList dhMitmSample
      registersize) dhMitmGoal
```

```
$ ghc -O --make DHMITM.lhs && (date +%F\ @\ %T; ./DHMITM ;date +%F\ @\ %T)
[7 of 7] Compiling Main             ( DHMITM.lhs, DHMITM.o )
Linking DHMITM ...
2014-05-27 @ 14:48:38
After CreateSized r 1 Alice (PowerMod (Nmbr 5) (Nmbr 6) (Nmbr 23)) 256:
  size = 2 // regs = [(r 1,8)]

After Open Mallory:
  size = 4 // regs = [(r 1,8)]

After Announce r 1:
  size = 4 // regs = [(r 1,8)]

After Close Mallory:
  size = 4 // regs = [(r 1,8)]

After CreateSized r 11 Mallory (PowerMod (Nmbr 5) (Nmbr 13) (Nmbr 23)) 256:
  size = 8 // regs = [(r 11,21),(r 1,8)]

After Open Bob:
  size = 16 // regs = [(r 11,21),(r 1,8)]

After Announce r 11:
  size = 16 // regs = [(r 11,21),(r 1,8)]

After Close Bob:
  size = 16 // regs = [(r 11,21),(r 1,8)]

After CreateSized r 2 Bob (PowerMod (Nmbr 5) (Nmbr 15) (Nmbr 23)) 256:
  size = 32 // regs = [(r 2,19),(r 11,21),(r 1,8)]

After Open Mallory:
  size = 48 // regs = [(r 2,19),(r 11,21),(r 1,8)]

After Announce r 2:
  size = 48 // regs = [(r 2,19),(r 11,21),(r 1,8)]

After Close Mallory:
  size = 48 // regs = [(r 2,19),(r 11,21),(r 1,8)]

After CreateSized r 22 Mallory (PowerMod (Nmbr 5) (Nmbr 13) (Nmbr 23)) 256:
  size = 96 // regs = [(r 22,21),(r 2,19),(r 11,21),(r 1,8)]

After Open Alice:
  size = 192 // regs = [(r 22,21),(r 2,19),(r 11,21),(r 1,8)]

After Announce r 22:
  size = 192 // regs = [(r 22,21),(r 2,19),(r 11,21),(r 1,8)]

After Close Alice:
  size = 192 // regs = [(r 22,21),(r 2,19),(r 11,21),(r 1,8)]

After CreateSized s 1 Alice (PowerMod (PrpE r 22) (Nmbr 6) (Nmbr 23)) 256:
  size = 384 // regs = [(s 1,18),(r 22,21),(r 2,19),(r 11,21),(r 1,8)]

After CreateSized s 2 Bob (PowerMod (PrpE r 11) (Nmbr 15) (Nmbr 23)) 256:
  size = 768 // regs = [(s 2,7),(s 1,18),(r 22,21),(r 2,19),(r 11,21),(r 1,8)]

After CreateSized s 11 Mallory (PowerMod (PrpE r 1) (Nmbr 13) (Nmbr 23)) 256:
  size = 1536 // regs = [(s 11,18),(s 2,7),(s 1,18),(r 22,21),(r 2,19),(r 11,21),(r
      1,8)]

After CreateSized s 22 Mallory (PowerMod (PrpE r 2) (Nmbr 13) (Nmbr 23)) 256:
  size = 3072 // regs = [(s 22,7),(s 11,18),(s 2,7),(s 1,18),(r 22,21),(r 2,19),(r
      11,21),(r 1,8)]

Conj [Neg (Equal (PrpE s 1) (PrpE s 2)),Equal (PrpE s 1) (PrpE s 11),Equal (PrpE s 2)
    (PrpE s 22),K Mallory (PrpF s 11),K Mallory (PrpF s 22)] is Just True

2014-05-29 @ 10:31:41
```

# Chapter 5

# Conclusion and Future Work

Combining several ideas from the literature on *Dynamic Epistemic Logic*, we defined and implemented two systems based on a new representation of what it means to know a number. The register models we presented can encode Kripke frames of exponentially larger size and allow us to focus on the relevant information in multi-agent situations.
The logic GG can represent knowledge and updates in guessing games. It shows in a simple setting how announcements of equalities and inequalities can be defined as action structures for the product update of register models. Our main technical result is a sound and complete axiomatization of GG using reduction axioms.
The second system we presented is *Epistemic Crypto Logic*, short ECL. It allows the analysis of directed communication and explicit computation as part of the language. Announcements to local sets of listeners and calls for attention are defined as action structures. We gave no axiomatization for ECL so far but sketched how it could be obtained via general axiomatizations for action structures. We also defined so-called *Monte Carlo Semantics* that allow us to estimate the truth of certain formulas without going through all assignments.
Real-world protocols can be translated to ECL and we did so for the Diffie-Hellman key exchange as a prime example that is both well-studied and used in practice.
For both GG and ECL we implemented model checking in *Haskell*. All examples and drawings of Kripke models have been generated with our program. Furthermore, we ran several experiments on our formalization of the Diffie-Hellman key exchange and could verify the efficiency of the proposed Monte Carlo method.

Coming to the end, we sketch some ideas how the presented work can be continued.

1. An obvious gap in our work on ECL is the missing axiomatization. So far we were unable to extend the GG system from Section 2.4 to a sound and complete system for ECL. But given that ECL is based on structures for which [BMS98] provides a general axiomatization, we expect that such a system can be found.

   First, to obtain a sound and complete axiomatization for common knowledge instead of the global modality as it occurred in GG, one can employ *relativized* common knowledge as it is described in [BEK06, p. 1633].

   Second, the main difficulty is the introduction of local listener sets that forces us to revise the reduction axioms for announcements and to find reduction axioms for **Open** and **Close**. The syntactic trick we discussed in Section 3.1.1 seems to make an aesthetic axiomatization impossible. A promising base for axiomatizing ECL seems to be [DHLS13, Section 6] which also includes adding and removing of listeners.

The following idea generalizing the framework of action structures might also yield a solution: So far we have preconditions on the different elements in an action structure. Why not do the same for the edges between them? We could then define announcements to the current set of listeners as an action model with two elements $\alpha$ and $\beta$ where the bidirectional edges $\alpha R_i \beta$ have the precondition $\neg L_i$ for all $i \in I$. In an S5-based approach the precondition of an edge has to hold at both ends for the edge to appear in the product. A similar alternative might be to use Arrow Update Logic from [KR11] where updates on accessibility relations can depend on which formulas are true at the two worlds that are connected by an edge.

It is notable that this idea has been suggested earlier, also motivated by security protocols: "A promising extension is to introduce conditional epistemic relations in the action model which depend on the epistemic states of the agents."[DW07]

2. We no longer include a global anchor function in to our models as previously suggested in [Gat13]. Instead we follow the motto of modal logic to "keep things local" and can connect to research in abstract modal logic. For example our GG models can be seen as coalgebras (See [Ven06]) for this functor:

$$X \mapsto (\mathcal{P}(\mathbf{P}) \times (\mathbb{N} \times \mathbb{N} \times \mathcal{P}(\mathbb{N}))^{\mathbf{P}} \times \mathbf{P}^2 \times \mathbf{P}^2) \times \mathcal{P}(X)^I$$

3. Both in $\mathcal{L}_{\mathsf{GG}}$ and $\mathcal{L}_{\mathsf{ECL}}$ announcements are restricted to equality and inequality statements. In particular we can express but not announce the following sentences:

| | |
|---|---|
| $p \vee q$ | "Either $p$ or $q$ is true." |
| $K_b p \vee K_b \neg p$ | "Bob knows whether $p$." |
| $\neg K_a(p = q)$ | "Alice does not know if $p$ and $q$ have the same numeric value." |

To model these correctly one would like to allow arbitrary announcements of the form $[!\phi]$ as presented in Section 1.2. However, deleting the worlds where $\phi$ is false is not the right thing to do in our framework, because $\phi$ could also be undecided in some worlds. Instead we need appropriate modifications on the valuation for each type of formula, as we presented them for announcements of equalities and inequalities.

A strategy to find the right action structures for announcements could be to first observe what the effect of the announcement in the big unraveled model is and then try to encode the resulting big model into a register model again.

4. For simplicity we did not add modalities for common knowledge to GG and ECL. As presented in [VVK07, Section 7.7], common knowledge and action structures can be combined in an axiomatizable logic. Hence a desirable generalization of our logics would be to include the modality $C$ to the language and explore the interplay of common knowledge and registers. As [WKvE09] shows there are interesting use cases for such logics, for example one would want to verify that common knowledge of the protocol does not endanger its security.

5. At the beginning of Section 3.1.1 we saw that some situations could be modeled in our framework but were unreachable with commands in $\mathcal{L}_{\mathsf{GG}}$. We can ask the same question about $\mathcal{L}_{\mathsf{ECL}}$: Given our set of commands, which models are reachable from one-world models of blissful ignorance and which are not? And an interesting follow-up question: Are there formulas which are valid on all reachable models but not on arbitrary $\mathsf{ECL}$-models?

6. Our Definition 71 for register creation does not demand an agent for which a register is created to know how to evaluate the given expression. Intuitively, this means that registers are created *for* and not *by* agents. To really represent the latter, one could add the precondition to $p \overset{i}{\leftarrow} E$ that $E$ is determined in all worlds which $i$ confuses with the actual one.

7. Our implementation can be extended in various ways including the following.

   (a) Given an axiomatization based on reduction schemes, one could also implement rewriting to command-free formulas for $\mathcal{L}_{\mathsf{ECL}}$.

   (b) The Monte Carlo evaluation functions could be altered to not just return a truth value but also say how reliable the result is according to the probabilities we briefly sketched in section 3.5. Working out the details of such an implementation would also allow us to find the “Monte Carlo fragment”, i.e. the set of formulas for which $\vDash$ and $\approx\!\!\!\!|$ coincide.

   (c) Throughout our experiments we sticked to relatively small registersizes. Especially the generation of random assignments should be improved for large numbers.

8. Finally, an ambitious goal would be to find attacks on a given protocol automatically. This could be done by defining a protocol as a list of commands with designated attack-points, for example whenever communication occurs. An attack on this protocol then would be a list of sequences meant to be inserted at these points. And just like a protocol, also an attack has a goal which can be stated as a formula.

   Now, an ideal implementation for automated attack finding would take as input the protocol and an attack goal and a set of allowed commands. By brute-forcing the combinations of commands that can be inserted at the attack points, it should then be able to find attacks like the Man-in-the-Middle attack on the Diffie-Hellman protocol. However, we have to admit that at this stage our implementation is too slow to make this practical. Further optimization is needed.

# List of Symbols

| **Symbol** | **Usage** |
|---|---|
| $\phi$, $\psi$, $\xi$, $\chi$, … | Formulas |
| $\Diamond$ | Basic modality "possible" |
| $\Box$ | Dual basic modality "necessary" |
| $K$ | Knowledge modality |
| $K_i$ | Knowledge modality for agent $i$ |
| $a$, $b$, $i$, $j$, … | Agents |
| $\alpha$, $\beta$, $\gamma$, … | Actions |
| $C$, $C_1$, $C_2$, … | Commands |
| $\mathsf{K}$ | Basic Normal Modal Logic |
| $\mathsf{S5}$ | Modal Logic on equivalence relations |
| $\mathsf{DEL}$ | Dynamic Epistemic Logic |
| $\mathsf{GG}$ | Guessing Game Logic |
| $\mathsf{ECL}$ | Epistemic Crypto Logic |
| $\mathsf{PDL}$ | Propositional Dynamic Logic |
| $\mathcal{L}_\Diamond$ | Basic Modal Language |
| $\mathcal{L}_{\mathsf{GG}}$ | Language for Guessing Games |
| $\mathcal{L}_{\mathsf{ECL}}$ | Language for Cryptographic Protocols |
| $\mathcal{F}$ | Frames |
| $\mathcal{M}$, $\mathcal{M}_1$, $\mathcal{M}_2$ | Models |
| $\mathcal{M}^\alpha$ | Model after product update with $\alpha$ |
| $\mathfrak{F}$ | Classes of frames |
| $\mathfrak{M}$ | Classes of models |
| $\mathbb{N}$ | The set of natural numbers $\{0, 1, 2, 3, \dots\}$ |
| $\mathcal{P}$ | Powerset functor |
| $\mathsf{dom}$ | Domain of a function |
| $\circ$ | Consecutive execution of functions |
| $f_{\upharpoonright X}$ | Restriction of $f$ to the set $X$ |

# Bibliography

[Ach14] Andreea Achimescu. Games and Logics for Informational Cascades. Master's thesis, Universiteit van Amsterdam, 2014.

[BDV01] Patrick Blackburn, Maarten De Rijke, and Yde Venema. *Modal Logic*, volume 53 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2001.

[BEK06] Johan van Benthem, Jan van Eijck, and Barteld Kooi. Logics of communication and change. *Information and computation*, 204(11):1620–1662, 2006.

[BMS98] Alexandru Baltag, Lawrence S. Moss, and Slawomir Solecki. The logic of public announcements, common knowledge, and private suspicions. In I. Bilboa, editor, *Proceedings of TARK'98*, pages 43–56, 1998.

[CDF$^+$07] J. Callas, L. Donnerhacke, H. Finney, D. Shaw, and R. Thayer. RFC 4880: OpenPGP Message Format, November 2007.

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2009.

[DH76] Whitfield Diffie and Martin Hellman. New directions in cryptography. *Information Theory, IEEE Transactions on*, 22(6):644–654, 1976.

[DHLS13] Hans van Ditmarsch, Andreas Herzig, Emiliano Lorini, and François Schwarzentruber. Listen to me! Public announcements to agents that pay attention – or not. In *Proceedings of LORI*, 2013.

[DK02] Hans Delfs and Helmut Knebel. *Introduction to Cryptography: Principles and applications*. Springer, 2002.

[Dv04] Kees Doets and Jan Eijck van. *The Haskell Road to Logic, Maths and Programming*. College Publications, May 2004.

[DW07] Francien Dechesne and Yanjing Wang. Dynamic epistemic verification of security protocols: Framework and case study. In *A Meeting of the Minds: Proceedings of the Workshop on Logic, Rationality, and Interaction, Beijing*, 2007.

[DY83] Danny Dolev and Andrew Chi-chih Yao. On the security of public key protocols. *Information Theory, IEEE Transactions on*, 29(2):198–208, Mar 1983.

[Eij07] Jan van Eijck. DEMO—a demo of epistemic modelling. In *Interactive Logic. Selected Papers from the 7th Augustus de Morgan Workshop, London*, volume 1, pages 303–362, 2007.

[Eij14] Jan van Eijck. Relations, Equivalences, Partitions. Technical report, CWI, March 2014.

[Fin74] Kit Fine. An incomplete logic containing S4. *Theoria*, 40(1):23–29, 1974.

[Fre92] Gottlob Frege. Über Sinn und Bedeutung. Translated as 'On Sense and Reference' in Geach and Black (eds.), Translations from the Philosophical Writings of Gottlob Frege, Blackwell, Oxford (1952), 1892.

[Gar13] James Garson. Modal Logic. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy.* Stanford University, spring 2013 edition, 2013.

[Gat13] Malvin Gattinger. Epistemic Crypto Logic - Functional Programming and Model Checking of Cryptographic Protocols. Technical report, ILLC, 2013.

[GN00] Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *Software - Practice And Experience*, 30(11):1203–1233, 2000.

[KL08] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography.* Cryptography and Network Security. Chapman & Hall/CRC, Boca Raton, FL, USA, 2008. inf 1.71: 2008 A 2847.

[Knu84] Donald E. Knuth. Literate Programming. *The Computer Journal*, 27(2):97–111, 1984.

[KR11] Barteld Kooi and Bryan Renne. Arrow update logic. *The Review of Symbolic Logic*, 4(04):536–559, 2011.

[Kri72] Saul A. Kripke. Naming and Necessity. In D. Davidson and G. Harman, editors, *Semantics of Natural Language*, pages 253–355. Reidel, Dordrecht, 1972.

[Mil76] Gary L. Miller. Riemann's hypothesis and tests for primality. *Journal of computer and system sciences*, 13(3):300–317, 1976.

[PP10] Christof Paar and Jan Pelzl. *Understanding cryptography: a textbook for students and practitioners.* Springer, 2010.

[Qui60] Willard Van Orman Quine. *Word and Object.* MIT Press, Cambridge, Mass., 1960.

[Rab80] Michael O. Rabin. Probabilistic algorithm for testing primality. *Journal of number theory*, 12(1):128–138, 1980.

[Sah75] Henrik Sahlqvist. Completeness and correspondence in the first and second order semantics for modal logic. *Studies in Logic and the Foundations of Mathematics*, 82:110–143, 1975.

[Sch14] Eric Schwitzgebel. Introspection. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Stanford University, summer 2014 edition, 2014.

[Sie07] Floor Sietsma. Model checking for dynamic epistemic logic with factual change. *Bachelor's Thesis, University of Amsterdam*, 2007.

[Tho74] S. K. Thomason. An incompleteness theorem in modal logic. *Theoria*, 40(1):30–34, 1974.

[Vel96] Frank Veltman. Defaults in update semantics. *Journal of Philosophical Logic*, 25(3):221–261, 1996.

[Ven06] Yde Venema. Algebras and coalgebras. In *Handbook of Modal Logic*, number 3 in Studies in Logic and Practical Reasoning, pages 331–426. Elsevier Science, 2006.

[VVK07] Hans Van Ditmarsch, Wiebe Van Der Hoek, and Barteld Kooi. *Dynamic epistemic logic*, volume 1. Springer Heidelberg, 2007.

[WKvE09] Yanjing Wang, Lakshmanan Kuppusamy, and Jan van Eijck. Verifying epistemic protocols under common knowledge. In *Proceedings of the 12th Conference on Theoretical Aspects of Rationality and Knowledge*, pages 257–266. ACM, 2009.

# Appendix

## REL.hs

This module contains various functions to work with relations and tuples: Closure operations, getting specific elements of a tuple, applying a relation, the Cartesian product, replacing elements in a list and separating strings with a given separator.

```haskell
module REL where
import Data.List

-- a is a type variable, we allow all kinds of relations:
type Rel a = [(a,a)]

concatRel :: Eq a => Rel a -> Rel a -> Rel a
concatRel r s = nub [ (x,z) | (x,y) <- r, (w,z) <- s, y == w ]

lfp :: Eq a => (a -> a) -> a -> a -- least fixed point
lfp f x | x == f x  = x
        | otherwise = lfp f (f x)

dom :: Eq a => Rel a -> [a] -- "domain"
dom r = nub (foldr (\ (x,y) -> ([x,y]++)) [] r)

rtc :: Eq a => Rel a -> Rel a -- reflexive-transitive closure
rtc r = lfp (\ s -> (s `union` (concatRel r s))) i
  where xs = dom r
        i  = [(x,x) | x <- xs ]

symc :: Eq a => Rel a -> Rel a -- symmetric closure
symc r = nub $ r ++ [ (y,x) | (x,y) <- r ]

rtsc :: Eq a => Rel a -> Rel a -- reflexive-transitive-symmetric closure
rtsc r = symc (rtc r)

-- fst and friends for triples, quadruples and quintuples:
fst3 :: (a,b,c) -> a
fst3 (a,_,_) = a

snd3 :: (a,b,c) -> b
snd3 (_,b,_) = b

trd3 :: (a,b,c) -> c
trd3 (_,_,c) = c

fst4 :: (a,b,c,d) -> a
fst4 (a,_,_,_) = a

snd4 :: (a,b,c,d) -> b
snd4 (_,b,_,_) = b

trd4 :: (a,b,c,d) -> c
trd4 (_,_,c,_) = c

fth4 :: (a,b,c,d) -> d
fth4 (_,_,_,d) = d

```

```
fst5 :: (a,b,c,d,e) -> a
fst5 (a,_,_,_,_) = a

snd5 :: (a,b,c,d,e) -> b
snd5 (_,b,_,_,_) = b

trd5 :: (a,b,c,d,e) -> c
trd5 (_,_,c,_,_) = c

fth5 :: (a,b,c,d,e) -> d
fth5 (_,_,_,d,_) = d

fft5 :: (a,b,c,d,e) -> e
fft5 (_,_,_,_,e) = e

relevant :: Ord a => Eq a => Rel a -> Rel a -- relevant part of a relation
relevant r = filter (\p -> not $ impl p) s
  where s = filter (\(x,y) -> (x < y)) r
        impl (a,b) = any (\c -> ( elem (a,c) s && elem (c,b) s )) (dom r)

apply :: Show a => Eq a => [(a,b)] -> a -> b -- apply a relation, return the first
    match
apply rel left =
  if (elem left ( map fst rel ))
    then snd $ head $ filter (\(a,_) -> a==left) rel
    else
      error ("apply failed: Relation is not defined at "++(show left)++" but only at "
           ++ show (map fst rel) )

cartProd :: [a] -> [b] -> [(a,b)] -- Cartesian product
cartProd xs ys = [ (x,y) | x <- xs, y <- ys ]

replace :: Eq a => [a] -> [a] -> [a] -> [a] -- replace an element in a list
replace [] _ _ = []
replace s search repl =
    if take (length search) s == search
        then repl ++ (replace (drop (length search) s) search repl)
        else [head s] ++ (replace (tail s) search repl)

sepBy :: [String] -> String -> String -- separating a set of elements with a separator
sepBy []  _ = ""
sepBy [x] _ = x
sepBy (now:todo) sep = sepByStep todo sep now
sepByStep :: [String] -> String -> String -> String
sepByStep [] _ done = done
sepByStep (now:todo) sep done = sepByStep todo sep (done ++ sep ++ now)
```

## MODEXP.hs

Given $a$, $b$ and $n$, this algorithm from [CLRS09, p. 957] efficiently computes $a^b \bmod n$. Based on joint work with Nadine Theiler for course homework in 2013.

```
module MODEXP where

exM ::  Integer -> Integer -> Integer -> Integer
exM a b n = exM' a (toBin b) n 1

exM' ::  Integer -> [Integer] -> Integer -> Integer -> Integer
exM' _ [] _ d = d
exM' a (b_i:b_rest) n d
    | b_i == 1   = exM' a b_rest n (mod ((mod (d*d) n)*a) n)
    | otherwise = exM' a b_rest n (mod (d*d) n)

toBin :: Integer -> [ Integer ]
toBin 0 = [ 0 ]
toBin n = toBin (quot n 2) ++ [rem n 2]
```

## KRIPKEVIS.hs

The following module creates visualizations of Kripke models. It is independent of the valuation type because it takes several functions to translate different parts of the models into strings. The module heavily employs *Graphviz* from www.graphviz.org which is discussed in detail in [GN00].

```haskell
module KRIPKEVIS where
import Data.List
import System.IO
import System.Process

begintab,endtab,newline :: String
begintab  = "\\\\begin{tabular}{c}"
endtab    = "\\\\end{tabular}"
newline   = " \\\\\\\\[0pt] "

type PartitionOf a = [[a]]

data VisModel a b c = VisModel [a] [(b,PartitionOf a)] [(a,c)] a

stringModel :: Ord a => Eq a => Eq b => Show a => Show b =>
  (a -> String) -> (b -> String) -> (c -> String) -> String
    -> VisModel a b c
      -> String
stringModel showState showAgents showVal information model =
  "graph G { \n\
   \  node [shape=doublecircle,label=\""
    ++ labelof cur ++ "\"] w" ++ showState cur ++ ";\n"
    ++ concat [ ndlinefor s | s <- otherstates ]
    ++ "  rankdir=LR; \n size=\"6,5!\" \n"
    ++ concat [ "  w" ++ show x ++" -- w"++ show y
                ++"[ label = \""++ showAgents a ++"\" ]; \n"
              | (a,x,y) <- edges  ]
    ++ "  label = \""++information++"\"; \n "
    ++ "} \n"
  where
    (VisModel states rel val cur) = model
    labelof s = begintab ++ "\\\\textbf{" ++ showState s ++ "}" ++ newline ++ showVal
        (visApply val s) ++ endtab
    ndlinefor s = "  node [shape=circle,label=\""
                  ++ labelof s ++ "\"] w" ++ show s ++ ";\n"
    otherstates = delete cur states
    edges = nub $ concat $ concat $ concat [ [ [ if (x < y) then [(a,x,y)] else [] | x
         <- part, y <- part ] | part <- (visApply rel a) ] | a <- (agents model) ]
    agents (VisModel _ rel' _ _) = map fst rel'

dotModel :: Ord a => Eq a => Eq b => Show a => Show b =>
  (a -> String) -> (b -> String) -> (c -> String) -> String
    -> VisModel a b c
      -> String -> IO String
dotModel showState showAgents showVal info model filename =
  let gstring = stringModel showState showAgents showVal info model
  in do
    newFile <- openFile (filename) WriteMode
    hPutStrLn newFile gstring
    hClose newFile
    return ("Model was DOT'd to "++filename)

texModel :: Ord a => Eq a => Eq b => Show a => Show b =>
  (a -> String) -> (b -> String) -> (c -> String) -> String
    -> VisModel a b c
      -> String -> IO String
texModel showState showAgents showVal info model filename =
  do
    forget <- dotModel showState showAgents showVal info model ("tex/"++filename++".
        dot")
    putStrLn forget
    _ <- system ("cd tex/; dot2tex --figonly -ftikz -traw -p --autosize -w --
        usepdflatex "++filename++".dot > "++filename++".tex;" )
    return ("Model was TeX'd to tex/" ++ filename ++ ".tex")

dispModel :: Ord a => Eq a => Eq b => Show a => Show b =>
  (a -> String) -> (b -> String) -> (c -> String) -> String
    -> VisModel a b c
      -> IO String
dispModel showState showAgents showVal info model =
  do
    forget <- dotModel showState showAgents showVal info model "tmp/temp.dot"
    putStrLn forget
    _ <- system ("cd tmp/; dot2tex -ftikz -traw -p --autosize -w --usepdflatex temp.
        dot > temp.tex; pdflatex -interaction=nonstopmode temp.tex > temp.pdflatex.log
        ; okular temp.pdf;" )
    return ("Model was TeX'd and shown.")

minimalHeader :: String
minimalHeader = "\\documentclass[12pt]{article} \n\
  \ \\usepackage[utf8]{inputenc} \n\
  \ \\usepackage[vcentering,papersize={16cm,9cm},total={15cm,8cm}]{geometry} \n\
  \ \\begin{document} \n"
minimalFooter :: String
minimalFooter = "\n \\end{document} \n"

dispTexCode :: String -> IO String
dispTexCode code = do
  newFile <- openFile ("tmp/code.tex") WriteMode
  hPutStrLn newFile (minimalHeader ++ code ++ minimalFooter)
  hClose newFile
  _ <- system ("cd tmp/; pdflatex -interaction=nonstopmode code.tex > code.pdflatex.
      log; okular code.pdf;" )
  return ("Code was TeX'd and shown.")

-- A helper function to apply relations:

visApply :: Show a => Eq a => [(a,b)] -> a -> b
visApply rel left =
  if (elem left ( map fst rel ))
    then snd $ head $ filter (\(a,_) -> a==left) rel
    else
      error ("Applying of a relation failed. Cannot visualize this.")
```

## Installing and running the implementations

To use the presented implementations of GG and ECL, LaTeX, Haskell, graphviz and gnuplot should be installed. On current versions of Debian this can be achieved with:

```
sudo apt-get install texlive-latex-extra graphviz gnuplot ghc cabal-install
```

We also use some Haskell libraries which can be installed by:

```
cabal update
cabal install process gnuplot primes criterion
```

Now you can download and run the implementation:

```
wget https://www.w4eg.de/malvin/illc/thesis/code.zip
unzip code.zip; cd code
ghci ECL.lhs # or ghci GG.lhs} for the implementation of GG
```

Information and errata will be available at `https://www.w4eg.de/malvin/illc/thesis`.