Comenius University, Bratislava
Faculty of Mathematics, Physics and Informatics

Board game description language

Master's thesis

2020
Bc. Truc Lam Bui

Comenius University, Bratislava
Faculty of Mathematics, Physics and Informatics

Board game description language

Master's thesis

Study program:	Computer Science
Field of study:	2508 Computer Science
Department:	Department of Applied Informatics
Supervisor:	RNDr. Jozef Šiška, PhD.

Bratislava, 2020
Bc. Truc Lam Bui

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

THESIS ASSIGNMENT

Name and Surname: Bc. Truc Lam Bui
Study programme: Computer Science (Single degree study, master II. deg., full time form)
Field of Study: Computer Science
Type of Thesis: Diploma Thesis
Language of Thesis: English
Secondary language: Slovak

Title: Board game description language

Annotation: For the implementation of digital board games, it is necessary to not only implement the game's logic, but also various supporting tools, such as the game server (responsible for the game's progress), game client (representing one of the players and providing a GUI for the user), communication protocol between them, recording and playing game records, game statistics, AI, ...
When the game is in the process of development, the game rules change often. However, implementation of the aforementioned tools limits the possible changes that can be made, unless we change a large amount of the code. In principle though, all these parts require only the rules of the game, in some suitable language, to work properly.

Aim: The goal of the thesis is to design a language suitable for the description and implementation of digital board games, and to implement an interpreter and any additions support toolsfor it. Core of the thesis should be in the design (syntax and semantics) of alanguage, that allows various constructions and mechanics used in board games to be implemented naturally.

Keywords: General Game Playing, Game Description Language

Supervisor: RNDr. Jozef Šiška, PhD.
Department: FMFI.KAI - Department of Applied Informatics
Head of department: prof. Ing. Igor Farkaš, Dr.

Assigned: 13.12.2018

Approved: 19.12.2018 prof. RNDr. Rastislav Kráľovič, PhD.
Guarantor of Study Programme

.....
Student

.....
Supervisor

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Bc. Truc Lam Bui
Študijný program: informatika (Jednoodborové štúdium, magisterský II. st., denná forma)
Študijný odbor: informatika
Typ záverečnej práce: diplomová
Jazyk záverečnej práce: anglický
Sekundárny jazyk: slovenský

Názov: Board game description language
Jazyk na popis stolových hier

Anotácia: Pri implementácii digitálnych stolové hier je potrebné okrem hernej logiky implementovať aj veľa podporných nástrojov, akými sú herný server (zodpovedný za odohranie hry podľa pravidiel), herný klient (reprezentuje jedného hráča a poskytuje užívateľovi používateľské rozhranie), komunikácia medzi klientami a serverom, ukladanie záznamov hier a ich prehrávanie, štatistiky z hier, umelá inteligencia, ...
Keď je hra v procese vývoja, často sa pravidlá hry menia. Pritom implementácia týchto nástrojov obmedzuje to, aké zmeny môžeme robiť v pravidlách hry, ak nechceme meniť veľkú časť kódu. Principiálne by týmto nástrojom ale mohol stačiť popis hry v nejakom vhodnom jazyku.

Cieľ: Cieľom práce je navrhnúť jazyk vhodný na popis a implementáciu digitálnych stolných hier, a implementovať nástroje na prácu s ním. Jadrom práce by mal byť návrh (syntax a sémantika) jazyka, v ktorom sa dajú vhodne a flexibilne implementovať rôzne konštrukcie a mechaniky zo stolných hier.

Kľúčové slová:

Vedúci: RNDr. Jozef Šiška, PhD.
Katedra: FMFI.KAI - Katedra aplikovanej informatiky
Vedúci katedry: prof. Ing. Igor Farkaš, Dr.

Dátum zadania: 13.12.2018

Dátum schválenia: 19.12.2018 prof. RNDr. Rastislav Kráľovič, PhD.
garant študijného programu

.....
študent

.....
vedúci práce

Acknowledgment: Thanks to my parents for implicitly supporting me during the last 23.75 years of my life (as of writing this), and for explicitly supporting me during the final weeks of writing this thesis. Thanks to Mišo for numerous discussions on functional programming topics; as a researcher, it is invaluable to have someone else to speak to and exchange ideas. For similar reasons, I would like to thank IST Austria, for providing me with the opportunity to experience a true research environment. I would also like to thank my friends for filling the vast empty plains of my time with occasional board game sessions. Last but not least, I would like to thank my supervisor YoYo for his leniency and listening.

Abstrakt

Stolné hry sú zaujímavou doménou. Ich pravidlá musia byť pochopiteľné človekom a teda musia byť dostatočne jednoduché, aby sa podľa nich dalo hrať. Na druhej strane, pravidlá môžu byť vyjadrené v prirodzenom jazyku. To umožňuje konštrukcie, ktoré by sa len ťažko vyjadrili v bežných programovacích jazykoch. Toto je unikátnou výzvou pre oblasť programovacích jazykov. V tejto diplomovej práci sa zaoberáme práve týmto problémom, a urobíme prvé kroky k programovaciemu jazyku pre stolné hry. Navrhneme jazyk, ktorý pristupuje k deklaráciám rovnakým spôsobom ako ku príkazom (obe sú prvky bežne sa vyskytujúce v pravidlách).

Kľúčové slová: jazyk na popis hier, programovacie jazyky, teória typov

Abstract

Board games are an interesting domain. The rules must be such that a human can follow them, which places some limit on the allowed complexity of the game. On the other hand, describing rules in natural language allows for constructions that are difficult to express in usual programming languages. This provides a unique challange for the field of programming languages. In this thesis, we tackle precisely this problem and make first steps towards a programming language for board games. We devise a language that takes a unified approach to both declarative and imperative statements (which are both ubiquitous in board game rules).

Keywords: game description language, programming languages, type theory

Contents

List of Tables

List of Algorithms

List of Figures

Introduction

Board games are an interesting domain. Unlike video games, where the rules are enforced by the computer, the rules of a board game must be such that humans can follow them. On one hand, this is a limitation, because humans do not have as much focused computing power as computers do. On the other hand, the rules can be expressed in natural language, which allows for constructions that are difficult to express in usual programming languages. These constructions provide a unique challange for the field of programming languages. This is the main motivation behind this thesis.

Aside from posing a challenge, there are other motivations for a game description language. In a similar way that humans require only the game rules to have a rough representation of the game, the computer could also (in principle) require only the formal rules of the game. Various tools in the ecosystem could be derived automatically from the description: the game server, game client, communication protocol between them, recording games and replaying them back, game statistics, AI, ... Without a formal description, these tools would have to be implemented separately. Not only does this lead to more work, but there is also the overhead of coordination and consistency: making sure the components have the same game in mind.

One major subfield is derivation of the AI player from the game description; this is called *general game playing* (GGP) [13]. The games there are mostly described in logic, with emphasis on the AI being able to reason about the game formally. More recently, the advances in machine learning have reduced the need for formal reasoning about the description, instead relying on reinforcement learning and similar. This led to the subfield of *general video game playing* (GVGP) [23].

Another subfield where a formal description is of significance is *computational game creativity*, where the objecitve is to generate interesting games. Usually, this is achieved by an evolutionary search process, where various high-level game components are randomly mixed into a single game and the result is evaluated based on some criteria. Examples of types of components are the shape of the board, the moves allowed for individual pieces, the interactions between pieces, ... We mention Ludi [8] and "A Card Game Description Language" [11] as examples of such systems.

Our interest lies mainly in the representation of the game, not in any specific application. The ideal language is general and flexible: if a game designer comes up with

an arbitrary game, that game should be expressible in the language. Rule adjustments that are small in the designer's mind should ideally be small in the formalism as well. The above two subfields pursue different goals, thus languages there are usually not interesting from a programming language perspective. Instead, we draw inspiration from more fundamental fields, and specially from the language of mathematics.

The rest of the thesis is structured into three parts. In the first part, we present the theoretical background that has shaped our ideas the most. Chapter 1 presents *type theory*, which can be viewed as the language for constructing objects. It plays a fundamental role in the language of mathematics. Chapter 2 discusses the field of *logic programming*, which considers logic in a computational setting. Logic is the language for talking about the relations between objects in a certain domain. This makes it ubiquitous in mathematics, but also outside of it (e.g. for describing everyday situations).

In the second part, we look at the domain of board games. Chapter 3 analyzes them, identifying common patterns and constructions occuring in the rules. Chapter 4 reviews the literature on game representation.

In the final part, we present our findings. Chapter 5 introduces *causality graphs*, which can be viewed as extensions of type theory with notions of time. We prove some basic properties of causality graphs. Then, we present the associated computational problems, show their hardness and then derive algorithms for both the general case (which turns out to be fixed-parameter tractable) and for some restricted cases. We then discuss the syntax and semantics, evaluate the language with respect to the board game patterns identified in chapter 3, and conclude with some directions for future work.

Although we have not been able to come up with a full-fledged programming language and there is currently no implementation, we believe causality graphs are a good foundation.

Part I

Theoretical background

Chapter 1

Type theory

In this chapter, we introduce concepts from type theory which we consider relevant to our pursuit. At an intuitive level, we view type theory as a *language of constructions*. This makes the language of type theory generally useful for describing any sort of construction process, in a similar way that logic is useful for describing relationships in a domain. The full language is described in the section 1.2. Finally, we discuss the intuitive meaning of the word "type" itself in section 1.3, leading to the *types as propositions* interpretation.

1.1 A language of constructions

At a technical level, type theory studies a certain class of formal systems, which are themselves called type theories. The main distinguishing point of type theories is that they talk in *judgments*, stating that a certain term has a certain type; the judgment "x is of type σ" is denoted $x : \sigma$. This is in contrast to formal systems studied in logic, which talk in propositions.

However, such a view of type theories is unsatisfying, as it does not reveal the principles behind the theory. Thus, a more intuitive interpretation is desired. One possible interpretation is as languages for constructing terms (of various types). For example, consider the terms of the untyped lambda calculus, which can be defined as follows:

1. If x is a variable, then x is a lambda term.
2. If t is a lambda term and x is a variable (not necessarily appearing within t), then $\lambda x.\, t$ is a lambda term.
3. If f and t are lambda terms, then so is $f\, t$.
4. Nothing else is a lambda term (implicitly).

This can be straightforwardly translated into the following type theory:

$$\frac{x : \sigma \in \Gamma}{\Gamma \Vdash x : \sigma} \quad \text{(initial)} \qquad \frac{\Gamma \Vdash x : \mathrm{Var} \qquad \Gamma \Vdash t : \Lambda}{\Gamma \Vdash \lambda x.\, t} \quad \text{(rule 2)}$$

$$\frac{\Gamma \Vdash x : \mathrm{Var}}{\Gamma \Vdash x : \Lambda} \quad \text{(rule 1)} \qquad \frac{\Gamma \Vdash f : \Lambda \qquad \Gamma \Vdash t : \Lambda}{\Gamma \Vdash f\, t : \Lambda} \quad \text{(rule 3)}$$

Then, the lambda terms are precisely those terms of type Λ that can be derived when the typing context Γ contains all symbols considered variables.

This is slightly unintuitive, since we should be able to introduce arbitrary symbols as variables; we should not be required to introduce them at the beginning. This points to the following redundancy in the type theory (and in fact, in the original definition of lambda terms): when forming an abstraction over the variable x, we can also remove it from the typing context. That is, in the second rule, the lambda term $\lambda x.\, t$ no longer requires x to be a variable. This leads to a modified version of rule 2:

$$\frac{\Gamma,\, x : \mathrm{Var} \Vdash t : \Lambda}{\Gamma \Vdash \lambda x.\, t} \quad \text{(rule 2')}$$

Then, lambda terms are precisely those terms that can be shown to be of type Λ in the empty context $\Gamma = \emptyset$. This is more intuitive than the previous definition, since lambda terms are a "top-level definition" — even when there are no assumptions Γ, there are some terms that are considered lambda terms.

Simply typed lambda calculus. In the simply typed lambda calculus, lambda terms no longer operate on other lambda terms. Instead, they always operate on terms whose types are already known. In this way, all lambda terms are built bottom up, starting with non-lambda terms whose types are already known from the typing context Γ. The formal system follows. For brevity, we omit the initial rule (i.e. it is assumed implicitly).

$$\frac{\Gamma,\, x : \sigma \Vdash e : \tau}{\Gamma \Vdash (\lambda(x : \sigma).\, e) : (\sigma \to \tau)} \quad \text{(abstraction)}$$

$$\frac{\Gamma \Vdash f : \sigma \to \tau \qquad \Gamma \Vdash x : \sigma}{\Gamma \Vdash f\, x : \tau} \quad \text{(application)}$$

In some sense, the simply typed lambda calculus is the most basic type theory of function types. It is the base from which more advanced type theories are derived.

Describing data. As a language of constructions, type theory is uniquely suited for defining mathematical domains. This, in turn, allows for defining (inductive) data types. For example, we can define the natural numbers as follows:

1. Zero is a natural number.
2. If x is a natural number, then so is suc x.
3. Nothing else is a natural number (implicitly).

$$\frac{}{\Vdash 0 : \mathbb{N}}\ \text{(nothing)} \qquad \text{(zero)} \qquad \frac{\Vdash x : \mathbb{N}}{\Vdash \text{suc } x : \mathbb{N}}\ \text{(successor)}$$

In the presence of functions, it suffices to include the judgments $\{0 : \mathbb{N},\ \text{suc} : \mathbb{N} \to \mathbb{N}\}$ in the context Γ. Thus, such definitions of domains have no special role in type theories, in that no new rules need to be introduced.

1.2 Lambda cube

In lambda calculus, abstraction introduces a dependency, in the sense that the return value of a function depends on the arguments supplied. The only kind of dependency is between terms: the abstraction is over a term x, and the result is the term $\lambda x.\, t$.

In the simply typed variant, the distinction between types and terms may seem artificial. Why is there no function that takes a term and returns a type? Similar kinds of dependencies can be introduced; the resulting formal systems form the *lambda cube* [5]. Each dimension of the cube represents one direction in which the simply typed lambda calculus can be extended: terms depending on types, types depending on types, and types depending on terms.

In the following presentation, we use Agda-like notation [7] for dependent types (i.e. function types where the return type can depend on the provided arguments). The type of all other types is denoted $*$, i.e. the judgment $\alpha : *$ is read "α is a type".

Parametric polymorphism. In systems with parametric polymorphism, terms can depend on types. That is, when we have some term t, we can abstract over some type α (if there are no assumptions on that type in the context Γ) to obtain a new term $\lambda(\alpha : *).\, t$.

To illustrate the benefits of this, consider for example the identity function $\texttt{id} = \lambda x.\, x$. In the simply typed system, the best we could do with regards to expressing its type is to use a type variable, i.e. $\texttt{id} : A \to A$. But once this function is used in any concrete context, the type variable A becomes bound and further uses of the function must refer to this exact same type. On the other hand, if parametric polymorphism is available, the universal identity function can be expressed as follows:

$$\lambda(\alpha : *).(\lambda(x : \alpha).\, x) \quad : \quad (\alpha : *) \to \alpha \to \alpha$$

Type constructors. Type constructors introduce dependencies between types. That is, when we have some type σ, we can abstract over another type α (if there are no assumptions on that type) to obtain a function $\lambda(\alpha : *).\, \sigma$ (which can be viewed as a parameterized type).

Examples of type constructors include generic lists, sets, maps, but also the arrow $\to$ can be viewed as a type constructor. However, type constructors achieve little on their own; consider generic lists as an example. The type itself is declared as $\texttt{list} : * \to *$, but to actually construct instances, parametric polymorphism is needed:

$$\begin{aligned} \texttt{nil} &: (A : *) \to \texttt{list}\ A, \\ \texttt{cons} &: (A : *) \to A \to \texttt{list}\ A \to \texttt{list}\ A. \end{aligned}$$

Dependent types. Dependent types allow types to depend on terms. That is, when we have some type σ, we can abstract over some term t of type A (if there are no assumptions on it) to obtain a function $\lambda(t : A).\, \sigma$ (which can be viewed as a parameterized type).

For example, consider the type of lists of booleans, where the length of the list is also part of the type. The type can be declared as the dependent type $\texttt{bListOfLength} : \mathbb{N} \to *$. For constructing instances, the following functions are defined:

$$\begin{aligned} \texttt{bNil} &: \texttt{bListOfLength}\ 0, \\ \texttt{bCons} &: \texttt{bool} \to \texttt{bListOfLength}\ n \to \texttt{bListOfLength}\ (\text{suc}\ n). \end{aligned}$$

Note that in the above example, even though `bListOfLength` is a dependent type, the result does not depend on its argument (i.e. it is $*$, regardless of what number is supplied). To illustrate that there *can* be a dependency, consider the function

$$\texttt{repeatBool} : (n : \mathbb{N}) \to \texttt{bool} \to \texttt{bListOfLength}\ n,$$

which merely repeats the provided boolean value n times, and returns the resulting list. In the setting of functional programming (i.e. where there is normalization and pattern matching), it can be expressed using the above constructors for `bListOfLength` as follows:

$$\begin{aligned} \texttt{repeatBool}\ 0\ b &= \texttt{bNil} \\ \texttt{repeatBool}\ (\text{suc}\ n)\ b &= \texttt{bCons}\ b\ (\texttt{repeatBool}\ n\ b) \end{aligned}$$

1.2.1 Formal presentation

We now briefly sketch the main rules in the type theory and the intuition behind them. For a more in-depth presentation, see the original paper by Barendregt [5].

Each of the above extensions allow different modes of abstractions, based on whether the abstraction is over a term or a type and based on whether the original is a term or a type. To distinguish these different modes, two *sorts* are introduced:

1. The sort of *proper types*, denoted $*$, representing the types of terms.

2. The sort of *kinds*, denoted $\Box$, representing the type of proper types.

This yields the following axiom:

$$\frac{\text{(nothing)}}{\Vdash * : \Box} \qquad \text{(a kind)}$$

In further text, we use the notions "term" and "type" simply to refer to the components of a judgment. That is, in the context of $x : \sigma$, the symbol x is the term and the symbol τ is the type. To refer to the original notions of terms and types, we say that something is *term-like* if its type is of sort $*$, and it is *type-like* if its type is of sort $\Box$. In general, if the sort is s, we say that it is *s-like*.

Each of the abstraction modes can be summarized as a pair of sorts (s_1, s_2), with the interpretation that we can abstract over s_1-like terms inside s_2-like terms. Then, for each allowed mode (s_1, s_2), we have the following rule for abstraction:

$$\frac{\Gamma \Vdash A : s_1 \qquad \Gamma,\, x : A \Vdash b : B \qquad \Gamma,\, x : A \Vdash B : s_2}{\Gamma \Vdash (\lambda(x : A).\, b) : ((x : A) \to B)} \qquad \text{(abstraction)}$$

An abstraction introduces a new function type with the same sort as the original. This can be justified as follows: an abstraction merely creates a parameterized term or type, so the sort should be the same. This yields the following rule for each allowed mode (s_1, s_2):

$$\frac{\Gamma \Vdash A : s_1 \qquad \Gamma,\, x : A \Vdash B : s_2}{\Gamma \Vdash ((x : A) \to B) : s_2} \qquad \text{(deriving sorts)}$$

Finally, the rule for application needs to be slightly modified to adjust for the fact that the return type can depend on the argument itself:

$$\frac{\Gamma \Vdash f : ((x : A) \to B) \qquad \Gamma \Vdash a : A}{\Gamma \Vdash f\, a : B[x := a]} \qquad \text{(application)}$$

1.3 Meaning of types

Types as denoting use patterns. One point of view is that a type denotes how something can be used. This is, in some sense, the most general reading of the word "type". There are intersection types, denoting that something can be used in multiple ways. Then, there are subtypes, denoting that something can always be used in place of some other (more general) type. Functions are processes that consume objects of the argument types and produce an object of the return type.

Types as propositions. It is common to associate a type with the set of all objects of that type. In this sense, types and objects are the same. However, there is an important distinction which can be summarized as follows: sets are semantic (meaningful) whereas

types are syntactic (meaningless). We elaborate this further; the following presentation is based on the paper by H. Guevers [14].

Consider the following set:

$$\{n \in \mathbb{N} \mid \forall x, y, z \in \mathbb{N}^+ : x^n + y^n \neq z^n\}.$$

Would we consider this set a type? To show that something is in this set, one needs a proof that the number satisfies the property, and this proof can be non-trivial. On the other hand, deriving the type of some object is usually more immediate. No difficult proofs are required, just looking at the description of the object suffices. In other words, the description of the object is also a *proof* of the type of that object; the type of the object then corresponds to a *proposition*. This is the *types as propositions* interpretation.

For example, consider the term $3 \cdot 4 + 2^7$. Since all of the atomic components (the constants 3, 4, 2 and 7) are integers and the operators for addition, multiplication and exponentiation yield integers when their arguments are integers, the entire term is an integer.

Note that deriving the type of an object can be problematic if an object can have multiple types. In the above example, the numerical constants could also be interpreted as floats, doubles, complex numbers, ... and the resulting term would then also be of these types. But then, deriving all types of the object (or even checking that it has a given type) can be a non-trivial task that requires some reasoning; it is *not* merely proof checking. Thus, in the most pure (most extreme) interpretation, each object has exactly one type.

Formal counterpart. There is also a formal counterpart to this intuitive notion of types as propositions: if we consider any type theory and erase information about the terms (i.e. look at the types only), we obtain a logical system. The terms can then be viewed as a *language* in which proofs are written. This makes type theory well suited for theorem provers and proof assistants.

For example, consider the simply typed lambda calculus. Erasing information about objects, we obtain a sequent calculus for minimal logic:

$$\frac{A \in \Gamma}{\Gamma \Vdash A} \qquad \text{(axiom)}$$

$$\frac{\Gamma,\, A \Vdash B}{\Gamma \Vdash A \Rightarrow B} \qquad \text{(introduction)}$$

$$\frac{\Gamma \Vdash A \Rightarrow B \qquad \Gamma \Vdash A}{\Gamma \Vdash B} \qquad \text{(elimination)}$$

Implication corresponds to the function type constructor. The rules correspond in order

from top to bottom to the initial, abstraction and application rules for the simply typed lambda calculus.

Chapter 2

Logic programming

Logic is used to express truths about objects in the domain of discourse, and this makes it universally useful. However, logic itself does not take into account the computational difficulty of actually recognizing truths from falsehoods. If we do take computational aspects into account, we get the field of *logic programming*.

Our goal in this chapter is to understand logic programming from the inside, as this will potentially illuminate design spaces previously unexplored. First, we describe what exactly logic programming entails. Then, in section 2.2, we discuss the ideal properties of programming languages in general, and how they apply to logic programming. Suitable logic programming languages are identified in section 2.3, where two main approaches are presented: *model construction* (2.3.1) and *goal-directed proof search* (2.3.2). Extending the languages with negation is considered in section 2.4 and finally, we present examples of actual logic programming languages in section 2.5.

2.1 Introduction

From the perspective of logic programming, a *program* is a set of assumptions Γ, and the result of the program is either the set of valid formulae, or the set of satisfiable formulae, or any other conclusions from the given assumptions. To illustrate formally, we could be asking any of the following questions:

1. What are the models of Γ? Is there a model at all?
2. Which formulae are valid, given Γ? This form of reasoning is called *cautious reasoning*: we accept only those conclusions which are true in all models.
3. Which formulae are satisfiable, given Γ? This is called *brave reasoning*: we accept any conclusion which is true in at least one model.

Note that in cautious reasoning, the set of conclusions is not necessarily a model itself, because it may be incomplete. There may be some formula ϕ such that neither ϕ

nor $\neg\phi$ is true in all models. For example, consider the simple program $\{\neg a \Rightarrow b, \neg b \Rightarrow a\}$. However, cautious conclusions are guaranteed to be consistent.

Similarly, in brave reasoning, the set of all conclusions is not necessarily a model, because the set may be inconsistent. For example, consider the program $\{a \vee \neg a\}$. However, the set is guaranteed to be complete. Brave reasoning can be seen as the complement of cautious reasoning, since a formula A is valid iff its negation $\neg A$ is not satisfiable.

The set of all such conclusions can easily become too large to be tractable. For this reason, it is necessary to either find a finite representation of the set where possible, or to instead ask more restricted questions: instead of asking for all conclusions, we ask only about one specific formula ϕ, called the *query*.

4. Is ϕ valid, given Γ? In symbolic terms: $\Gamma \vDash \phi$?
5. Is ϕ satisfiable, given Γ? If so, what substitutions for free variables satisfy the formula?

To summarize, in logic programming, computation takes the form of a search for either a model, the set of satisfiable or valid formulae, or for a proof of the queried formula. Formulas themselves form the program. This is in contrast to *axiomatic semantics* of programming languages, where logic is merely used as a tool to describe the meaning of programs.

2.2 Programming concerns

Not every logic is suitable for logic programming. For a logic to be suitable, at least some of the above mentioned questions have to be decidable, and ideally also computationally tractable. For example, even in the simple case of propositional classical logic with no quantifiers, the problem of satisfiability (SAT) is NP-complete. For a summary of similar hardness results, see table 2.1.

However, computational hardness is not the ultimate concern. Hardness merely states that the language is capable of expressing complex problems; there may still be *easy* fragments of the language that *are* computationally tractable. It is in these fragments that the user is usually doing the programming.

To illustrate, one can formulate hard (even undecidable) problems in any sufficiently expressive programming language. If we can express the halting problem within the language, then deciding whether a program in the language halts or not is (in general) undecidable. This, however, does not stop the user from using the language for other purposes.

The actual concern is: how easily and how precisely can the user express his ideas, and how well can the user understand what is going on in the system? For example,

allowed form of formula	complete for
$\exists \bar{x}.\, \Phi(\bar{x})$ (SAT)	NP
$\exists \bar{x_1}.\, \forall \bar{x_2}.\, \Phi(\bar{x_1}, \bar{x_2})$	Σ_1^P
$\exists \bar{x_1}.\, \forall \bar{x_2}.\, \exists \bar{x_3}.\, \Phi(\bar{x_1}, \bar{x_2}, \bar{x_3})$	Σ_2^P
$\vdots$	$\vdots$
n alternations, starting with $\exists$	Σ_n^P
$\forall \overrightarrow{x}.\Phi(x)$ (TAUT)	coNP-complete
$\forall \bar{x_1}.\, \exists \bar{x_2}.\, \Phi(\bar{x_1}, \bar{x_2})$	Π_1^P
$\vdots$	$\vdots$
n alternations, starting with $\forall$	Π_n^P
any number of alternations (QBF)	PSPACE

Table 2.1: Hardness results for propositional classical logic. Unlike first-order logic, quantifiers range over possible boolean values $\{0, 1\}$ of the bound variable, instead of ranging over individuals.

consider a simple interpreted language for SAT that simply tries out random assignments to variables until it succeeds. With this knowledge, the user can reason that the language is well suited for problems (logical formulae Φ) for which a large proportion of the assignments to variables are correct. On the other hand, if some formula Φ does not have this property, there is no (external) way to make the system work faster. Even a simple formula such as $a_1 \wedge \neg a_2 \wedge \neg a_3 \wedge \ldots \wedge \neg a_n$, which can be satisfied by a human at glance, will take the system $O(2^n)$ time to solve (in expectation).

On the other hand, consider imperative programming languages. In their essence, programs describe some step-by-step processes; executing a program given its description is a straightforward, manual task. At this level, everything is easy to imagine. The range of behaviours of the program and its complexity is thus not attributed to the complexity of the programming language, but rather to the idea the program is trying to express / the problem it tries to solve.

2.3 Programming with Horn clauses

One way to obtain a logic better suited for logic programming is to restrict the set of formulae allowed in the program Γ. One such restriction is to the set of *Horn clauses*.

Definition 1. A formula is a *Horn clause* if it has the following form:

$$a_1 \wedge a_2 \wedge \ldots \wedge a_n \Rightarrow l$$

where $a_1, \ldots, a_n$ are atoms and l is either a literal or the logical constant $\bot$.

When talking about Horn clauses in some program Γ, we call them *rules*. The conclusion l is called the *head* of the rule; if the head if $\bot$, we call the rule a *constraint*. The conjunction $a_1 \wedge \ldots \wedge a_n$ is called the *body* of the rule; if the body is empty, we call the rule a *fact*.

It is customary to instead write

$$l \leftarrow a_1, a_2, \ldots, a_n,$$

that is, the conclusion is to the left, the premises to the right, the symbol $\leftarrow$ represents flipped implication, and we separate the premises by commas instead of the usual conjunction symbol $\wedge$. We use the two notations interchangably, with the former used mainly in logical contexts, and the latter in programming contexts.

2.3.1 Model construction

We are interested in information about the program as a whole, without reference to some query. That means finding models, performing cautious or brave reasoning and similar. This approach is best suited for finite domains, where it is expected that the models are finite. Examples are expert systems, databases, ...

Propositional case. In a propositional Horn clause, all atoms are required to be atomic propositions; similarly, all literals are required to be either atomic propositions or their negations.

The problem of deciding whether a set of propositional Horn clauses is satisfiable or not is called *Horn satisfiability*. There is an algorithm solving it in time $O(n)$, where n is the total size of the program (the total number of atoms and literals among all Horn clauses). In case the set is satisfiable, there exists a least model [29], and the algorithm outputs this model.

The algorithm proceeds bottom up, starting with facts. At any time, if the body of some rule becomes satisfied, the rule fires, forcing its head to become true. This is repeated until no new facts can be derived. Details on the algorithm can be found in the paper by Dowling and Galier. [9] Similar algorithms, where new information is derived from what is already known, are called *forward chaining* algorithms.

Extended Horn clauses. From a logical point of view, it makes sense to restrict the form of the clauses as much as possible while retaining their logical properties, because this makes analysis easier. However, from a practical (i.e. programming) point of view, we would like to express our ideas as easily as possible. Thus it makes sense to consider as unrestricted forms as possible, while retaining the logical properties.

Definition 2. A formula is an *extended Horn clause* if it matches the syntactic variable H. The syntactic variables H (heads of clauses) and B (bodies of clauses) are defined using the Backus-Naur form as follows:

$$B ::= \top \mid A \mid B \wedge B \mid B \vee B,$$
$$H ::= \bot \mid L \mid B \Rightarrow H \mid H \wedge H,$$

where A is the syntactic variable for atoms and L is the syntactic variable for literals.

In other words, the extension allows us to use disjunction in the bodies, and conjunctions and implication in the heads of Horn clauses. This is based on the following observations:

1. Each occurence of $B_1 \vee B_2$ in the bodies of clauses can be replaced by a new atom a if we augment the program by adding clauses $\{B_1 \Rightarrow a,\ B_2 \Rightarrow a\}$.
2. Each occurence of $H_1 \wedge H_2$ in the heads of clauses can be replaced with a new atom a if we augment the program with $\{a \Rightarrow H_1,\ a \Rightarrow H_2\}$.
3. Each occurence of $B \Rightarrow H$ in the heads of clauses can be replaced with a new atom a if we augment the program with $\{(a \wedge B) \Rightarrow H\}$.

These can be viewed as rewriting rules. It can be shown that the rules preserve the least model (if we ignore the new atom a in the models). Thus, by iteratively applying them to a program of extended Horn clauses, we eventually arrive at a program of basic Horn clauses that has the same least model. This transformation preserves the asymptotic size of the program, i.e. the resulting program of basic Horn clauses is at most a constant factor larger than the original program. Therefore, the least model can be computed in the same time as in the case of basic Horn clauses.

First-order case. The difference from the propositional case is the introduction of functions and predicates, and of the universal and existential quantifiers. There are restrictions as to which quantifiers can be where, and what types can they range over.

Definition 3. A *first-order (basic) Horn clause* is a formula that matches $B \Rightarrow H$. The syntactic variables H, B are defined using Backus-Naur form as follows:

$$B ::= \top \mid A \mid B \wedge B \mid \exists x.\, B,$$
$$H ::= \bot \mid L \mid \forall x.\, H,$$

where x ranges over non-function non-predicate types.

All free variables are implicitly universally quantified at the clause level. Since the clause matches the head variable H, this implicit quantification does not violate the definition.

Note that an existential quantification of a variable in the body of the clause is classically equivalent to universal quantification of the same variable over the entire clause. Symbolically,

$$(\exists x.\, B) \Rightarrow H \qquad \Longleftrightarrow \qquad \forall x'.\, (B[x := x'] \Rightarrow H),$$

where x' is a new variable (not occuring freely in B nor in H). This equivalence allows us to dispense of existential quantification in bodies. Thus, first-order Horn clauses are really just clauses of the form $B \Rightarrow H$ with no quantifiers, where all variables are implicitly universally quantified at the clause level.

To ensure that the model is finite, the rules are required to be *safe* according to the following definition:

Definition 4. A rule is *safe* if each variable in the head of the rule is present in the body of the rule as well. Otherwise, the rule is *unsafe*.

There are two ways to approach the least model problem. First, we can reduce the problem to the propositional case by *grounding* the program, transforming it into an (essentially) equivalent program with no variables in clauses. Second, we can instead solve the problem directly. We present the latter approach; the former is of interest mainly in the presence of negation, when a direct approach is non-trivial.

The idea of the algorithm is the same as in the propositional case. We maintain the set of so far derived truths, and in each step, we match them against all the rules to derive new truths. This is iterated until no new truths can be derived. Unlike in the propositional case, the presence of variables causes the matching process to be non-trivial, and it needs to be done efficiently in order for the algorithm to be efficient. An overview can be found in the technical report by Bancilhon and Ramakrishnan [4]; it is presented from the point of view of database theory. In what follows, we present an ad-hoc solution.

Ad-hoc solution. Each rule is represented as a right-nested implication $a_1 \Rightarrow a_2 \Rightarrow \ldots \Rightarrow a_n \Rightarrow l$. Whenever a new truth t is derived that matches a_1, a new rule instance $(a_2 \Rightarrow \ldots \Rightarrow a_n \Rightarrow l)[a_1 = t]$ is derived. Whenever a new rule instance is derived, if it is a fact, the corresponding truth is asserted. Otherwise, we consider candidate truths t that could match a_1 and for each of those that in fact do match it, a new rule instance is derived.

In other words, each rule sits on top of the first atom a_1 in its body, waiting for new matches t; each truth t waits for new rule instances that match it. Thus, the efficiency of this algorithm hinges on the ability to quickly look up all candidate rules (whenever a new truth is derived) and all candidate truths (whenever a new rule instance is derived). One option is to have *buckets* for all atoms encountered so far and their

generalizations. For example, if a truth $\texttt{parent}(\texttt{anna}, \texttt{boris})$ had been derived, there would be four buckets:

$$\texttt{parent}(\texttt{anna}, \texttt{boris}), \quad \texttt{parent}(*, \texttt{boris}), \quad \texttt{parent}(\texttt{anna}, *), \quad \texttt{parent}(*, *),$$

and the truth would fall into all four of them. In general, an n-ary predicate has 2^n buckets. When looking up candidate truths and candidate rule instances, it suffices to search the appropriate buckets.

2.3.2 Goal-directed proof search

In this approach, we pose a *query* to the system. The system then looks for proofs of the query, binding any free variables as necessary. For each found proof, the system returns the bindings for the free variables.

Goal-directed proof search is suitable in domains that are large or even infinite, such as abstract domains. Examples are mathematics (theorem provers, proof assistants, ...), general purpose programming, ...

To illustrate, suppose we are looking for all binary strings that are palindromes. Since lists can be of arbitrary length, the models are infinite, and explicitly constructing them is infeasible. However, in the presence of the above query, we may be able to lazily generate all such lists, even though there are infinitely many of them.

Normally, the rules of a formal system are read from top to bottom. For example, consider the rule of syllogism:

$$\frac{\Gamma \vdash A \Rightarrow B, \quad \Gamma \vdash B \Rightarrow C}{\Gamma \vdash A \Rightarrow C}.$$

The standard reading is that if the premises (at the top of the rule) are satisfied, then the conclusion follows. However, we are not interested in all the truths; we are interested only in the query, the goal. In this context, it makes sense to read the rules bottom to top. The above rule can be read as: "One way to prove $A \Rightarrow C$ is to find some B such that both $A \Rightarrow B$ and $B \Rightarrow C$ are provable."

For such an approach to theorem proving to be feasible, the number of options to consider must be sufficiently low. In the above example, this is clearly not the case, because we have to guess B and there are no obvious candidates for it. We see that arbitrary formal systems will *not* do as goal-directed logic programming languages. The problem of which systems *will* do has been discussed by Miller et. al. [21].

First-order extended Horn clauses. We adopt the notation used by Miller and Nadathur [20], where the sequents are denoted differently in the context of goal-directed proof search. The program is denoted $\mathcal{P}$ and the logical consequence is denoted $\longrightarrow$.

Definition 5. A formula is a *first-order extended Horn clause* if it matches D. The syntactic variables D (domain clauses) and G (goal clauses) are defined using Backus-Naur form as follows:

$$G ::= \top \mid A \mid G \wedge G \mid G \vee G \mid \exists x.\, G,$$
$$D ::= A \mid G \Rightarrow D \mid D \wedge D \mid \forall x.\, D,$$

where the variable x ranges over non-function non-predicate types.

In comparison to Horn clauses as they were defined in the part on model construction, the domain clauses correspond to heads of rules, and goal clauses correspond to bodies of rules. Note the following difference: no negation is allowed in domain clauses.

Let us now introduce the formal system. It consists of two parts: the first part deals with logical structure of goals, and the second part tells what should be done when the goals are atomic (i.e. there is no structure to decompose). We start with the first part.

$$\frac{\mathcal{P} \longrightarrow G_1 \qquad \mathcal{P} \longrightarrow G_2}{\mathcal{P} \longrightarrow G_1 \wedge G_2} \qquad \text{(decomposing on } \wedge\text{)}$$

$$\frac{\mathcal{P} \longrightarrow G_1}{\mathcal{P} \longrightarrow G_1 \vee G_2} \qquad \frac{\mathcal{P} \longrightarrow G_2}{\mathcal{P} \longrightarrow G_1 \vee G_2} \qquad \text{(decomposing on } \vee\text{)}$$

$$\frac{\mathcal{P} \longrightarrow G[x := X] \qquad \text{where } X \text{ is a new variable}}{\mathcal{P} \longrightarrow \exists x.\, G} \qquad \text{(decomposing on } \exists\text{)}$$

The first sequent states that to prove a conjunction $G_1 \wedge G_2$, it suffices to prove both G_1 and G_2. The second and third sequents state that to prove a disjunction $G_1 \vee G_2$, it suffices to either prove G_1 or to prove G_2. Finally, the fourth sequent states that to prove $\exists x.\, G$, it suffices to prove $G[x := X]$, where X is a new variable bound at the global level.

If none of the above sequents apply, then the goal must be an atomic formula, denote it A. Since there is no further propositional structure, the only way forward is to use the program clauses. We guess which of the clauses will be *most immediately useful* (i.e. the previous step in the proof) and pick it. Let us denote it D, we say that we are *backchaining* on D, and we call D the *intermediary*. Once we pick D, we can analyze the structure of the proof of $D \Rightarrow A$.

If $D = G \Rightarrow D'$, since it is useful (i.e. relevant), the next step in the proof is to first prove G, and then continue with proving $D' \Rightarrow A$. This brings us closer to the goal A, and we are now backchaining on D'.

If $D = D_1 \wedge D_2$, then we prove A using either D_1 or D_2. We have to use one of them (due to relevance), and we cannot use both (due to immediacy). Thus, we can backchain on either D_1 or D_2.

If $D = \forall x.\, D'$, then we prove A using some instantiation of D'. The most general instantiation is to use $D'[x := X]$, where X is a new variable at the global level; this variable will be instantiated in the future as necessary. Thus, we backchain on $D'[x := X]$.

Finally, if D is an atom, the only way to use it for proving A is to unify it with A (i.e. instantiate variables in D so that they are the same atom).

The rules for backchaining are summarized below. The intermediary is denoted above the arrow $\longrightarrow$; in a pure logical interpretation (top-down instead of bottom-up proof search), it can be read as being on the left side of the sequent.

$$\frac{\mathcal{P} \xrightarrow{D} A \qquad \text{where } D \in \mathcal{P}}{\mathcal{P} \longrightarrow A} \qquad \text{(choose intermediary)}$$

$$\frac{\text{(nothing)}}{\mathcal{P} \xrightarrow{A'} A \qquad A \text{ and } A' \text{ are unifiable}} \qquad \text{(backchaining base case)}$$

$$\frac{\mathcal{P} \xrightarrow{D} A \qquad \mathcal{P} \longrightarrow G}{\mathcal{P} \xrightarrow{G \Rightarrow D} A} \qquad \text{(backchaining on } \Rightarrow\text{)}$$

$$\frac{\mathcal{P} \xrightarrow{D_1} A}{\mathcal{P} \xrightarrow{D_1 \wedge D_2} A} \qquad \frac{\mathcal{P} \xrightarrow{D_2} A}{\mathcal{P} \xrightarrow{D_1 \wedge D_2} A} \qquad \text{(backchaining on } \wedge\text{)}$$

$$\frac{\mathcal{P} \xrightarrow{D[x:=X]} A \qquad \text{where } X \text{ is a new variable}}{\mathcal{P} \xrightarrow{\forall x.\, D} A} \qquad \text{(backchaining on } \forall\text{)}$$

The above is merely an intuitive motivation for the rules; it is by no means a formal proof of the completeness of the resulting proof search procedure. For this, consult the paper on uniform proofs by Miller et. al. [21], where it is shown that this proof search strategy is (in fact) complete for all first-order Horn clause programs, under both classical and intuitionistic logic.

In practice, a logic programming system must also make choices about the order in which subgoals should be proven, in what order should the program clauses be tried out, order in which disjunctive goals are tried out, ... The order can matter, in that different orders can lead to different running times of queries. Usually, to allow the programmer as much control over the program as possible (so that the system is "transparent"), the ordering is based on some simple rule, such as:

1. Conjunctive goals $G_1 \wedge G_2$ are tried from left to right.
2. The same for disjunctive goals.
3. Domain clauses are tried out based on the order in which they are listed in the source code, from top to bottom.

First-order hereditary Harrop formulae. The simple proof search strategy based on backchaining can be extended to a larger class of formulae, where both implication and universal quantification are allowed in goal clauses.

Definition 6. A formula is a *first-order hereditary Harrop formula* if it matches D. The syntactic variables G and D are defined using Backus-Naur as follows:

$$G ::= \top \mid A \mid G \wedge G \mid G \vee G \mid D \Rightarrow G \mid \exists x.\, G \mid \forall x.\, G,$$
$$D ::= A \mid G \Rightarrow D \mid D \wedge D \mid \forall x.\, D,$$

where the variable x ranged over non-function non-predicate types.

How useful are these additions? The power of implication in bodies is that it allows for hypothetical reasoning. The power of universal quantification in bodies, when coupled with implications, is that it allows us to consider "most general objects".

For example, consider the simple program

$$\{\texttt{red}(X) \Rightarrow \texttt{hasColor}(X), \quad \texttt{blue}(X) \Rightarrow \texttt{hasColor}(X)\}.$$

We may ask if everything that is red also has a color, by posing the query $\forall x.\, \texttt{red}(x) \Rightarrow \texttt{hasColor}(x)$. The system would correctly deduce yes, by assuming a new object c with $\texttt{red}(c)$ and then deriving $\texttt{hasColor}(c)$.

However, it would *not* derive the goal $\forall x.\, \texttt{hasColor}(x) \Rightarrow (\texttt{red}(x) \vee \texttt{blue}(x))$, because nothing can be derived purely from $\texttt{hasColor}(c)$, even though the above would be true in a closed world.

How does this extension affect the system? Backchaining is not affected, since the domain clauses are unaffected. However, there are further possibilities when breaking down the goal formula when it is not atomic. The new rules are:

$$\frac{\mathcal{P},\, D \longrightarrow G}{\mathcal{P} \longrightarrow D \Rightarrow G} \qquad \text{(decomposing on } \Rightarrow\text{)}$$

$$\frac{\mathcal{P} \longrightarrow G[x := c] \qquad \text{where } c \text{ is a new constant}}{\mathcal{P} \longrightarrow \forall x.\, G} \qquad \text{(decomposing on } \forall\text{)}$$

Note that both of these rules change the context in which we operate: the first rule augments the program with a new program clause, and the second rule introduces a new constant of which nothing is known, initially — but this can change with the augmentation of the program.

The resulting proof search procedure is no longer complete for classical logic; however, it is complete for intuitionistic logic [21]. The problem with allowing the new notions in the classical setting is illustrated by the following example:

Consider the empty program, and ask the query $p \vee (p \Rightarrow q)$. This is classically valid. However, the proof search breaks down the problem into either $\emptyset \longrightarrow p$, which

fails; or $p \longrightarrow q$, which also fails. The problem inherent in classical logic is the law of the excluded middle $p \vee \neg p$, which allows us to prove $p \vee (p \Rightarrow q)$ without proving either of the disjuncts. The negation is hidden in the implication.

2.4 Meaning of negation

So far, we have considered negation only in model construction methods, and only in heads of clauses. The meaning of negation there is clear: if we derive two conflicting literals or the false proposition $\bot$, there is no model. Other than that, there is no use for negated literals, since there are only positive literals in bodies of clauses.

The situation changes, however, when we allow negation in the bodies of the rules. A least model does not necessarily exist in such case; for example, consider the program $\{a \leftarrow \neg b,\ b \leftarrow \neg a\}$. In fact, allowing arbitrary negations in the bodies of the rules leaves us at the starting point, where we are asking about the satisfiability of an arbitrary (i.e. unrestricted) program. This is because under classical logic, Horn-like implications are related to disjunctions in the following way:

$$l \leftarrow a_1 \wedge \ldots \wedge a_n \qquad \Longleftrightarrow \qquad l \vee \neg a_1 \vee \ldots \vee \neg a_n$$

If we allow the a_i to be any literals, not just atoms, then we can express arbitrary disjunctions. The program is then a conjunction of disjunctions, and satisfiability of such a formula is in general NP-complete (SAT).

There are also considerations other than computational hardness, such as the concern for the meaning of the program. In standard logic, the presence or absence of propositions merely indicates information: if neither a nor $\neg a$ can be derived, then it is not reasonable to assume either (there are both models with a and models with $\neg a$). This is the called the *open world assumption*; there is no special treatment of negation.

On the other hand, if we operate in a closed environment, it may be reasonable to assume that everything that cannot be derived from the program is false. This is the *closed world assumption*, and its treatment of negation is called *default negation*.

To illustrate the difference, consider the program $\{a \leftarrow b\}$. Under the open world assumption, the program has three models: $\{\neg a, \neg b\}$, $\{a, \neg b\}$ and $\{a, b\}$. However, under the closed world assumption, only the first model makes sense; in the second model, there is no reason why a should hold, whereas in the third model, there is no reason why b should hold. As another example, consider $\{a \leftarrow \neg b\}$. There are again three models under the open world assumption. But under the closed world assumption, only $\{a, \neg b\}$ is a model.

The closed world assumption works just fine in the case of Horn clauses, due to the existence of the least model: everything else could be assumed false. However, in the presence of negation in the bodies of rules, it may become unclear what the intended

meaning of the program is. For example, consider the program $\{a \leftarrow \neg b,\ b \leftarrow \neg a\}$. There are two models under the open world assumption: $\{a, \neg b\}$ and $\{\neg a, b\}$. But which of them should also be models under the closed world assumption? Does it make sense to prefer either model over the other? How can we justify these models, intuitively?

In what follows, we present the common approaches towards negation. We view them in light of both concerns raised above: computational hardness and the meaning of the program (intuition).

2.4.1 Default negation

A proposition is assumed false unless proven otherwise (from the program). This can be interpreted both in the context of goal-directed proof search and in the context of model construction. Note that this meaning of negation is completely distinct from negation in the head of a clause. The former expresses that something is not known to be true (not provable from the program), while the latter expresses that something is known to be false. To distinguish them from each other, the former is denoted $\sim a$ and is called *default negation*; the latter retains the ordinary notation $\neg a$ and is called *explicit negation*. Explicit negation acts as a filter: models with both a and $\neg a$ are dropped due to being inconsistent; other than that, it has no special treatment.

In goal-directed proof search. In this context, the term *negation as failure* is instead of default negation. This comes from the procedural nature of goal-directed proof search: to prove $\sim a$, we try to prove the subgoal a. The original goal succeeds iff the subgoal fails, thus negation means failure to prove a certain goal.

Note that this has unintuitive behaviour when the formula a contains free variables: when the formula $\sim a$ succeeds, it does not bind any of the free variables. Then, consider the program $\{p(x), q(y)\}$ and the query $p(X) \wedge \sim q(X)$. If the subgoal $\sim q(X)$ is evaluated first, then no solution is reported; on the other hand, if the subgoal $p(X)$ is evaluated first, the solution $X = a$ is reported. Thus, negation breaks the logical purity and must be used carefully to avoid such side effects.

In model construction. For some programs, it is not clear what should be the representative model. Consider for example the program $\{a \leftarrow \sim b,\ b \leftarrow \sim a\}$. No proposition is initially provable, so it would seem that we should assume both $\sim a$ and $\sim b$. But if we do, then both a and b become provable, a contradiction.

On the other hand, there are some programs for which there *is* an intuitive representative model. Consider for example the program $\{a \leftarrow \sim b,\ b \leftarrow \sim c\}$. We see that c cannot be proved no matter what, so we can assume $\sim c$. With this, we can prove b;

then, $\sim b$ cannot be proved no matter what, so we conclude with $\sim a$.

The above example can be generalized to an entire class of well-behaved programs.

Definition 7. A program is *locally stratified* if the ground atoms can be assigned to strata so that: each atom depends positively only on atoms in the same or lower strata, and negatively only on atoms in lower strata.

Any locally stratified program has exactly one model [24] (or at most one model, if explicit negation is allowed). We present an ad-hoc algorithm for constructing it, assuming that we know the assignment of ground atoms to strata.

Ad-hoc algorithm. The model can be constructed by considering the strata from lowest to highest. When processing a particular stratum S, since all lower strata have already been processed, the truth values of these previous atoms are already known. Thus, we can substitute these values into the rules that produce atoms in S. The resulting set of rules contains only atoms from stratum S itself, and all dependencies are positive. Thus, there is a least model for atoms in S, which can be computed using standard methods (without negation). All atoms in this model are declared true, and the remaining atoms from S are declared false. Then, we continue with the next stratum, ...

Unfortunately, testing programs for local stratification is an undecidable problem [22]. Thus, tighter restrictions are considered.

Definition 8. A program is *stratified* if the predicates can be assigned to strata so that: each predicate depends positively only on predicates in the same or lower strata, and negatively only on lower strata.

Clearly, each stratified program is locally stratified as well. The assignment of predicates to strata can be calculated with topological sorting, in time linear in the size of the program.

In general, not all programs of interest are necessarily stratified. We are interested in extending the default negation approach to those programs as well.

Stable model semantics. A naive interpretation of default negation: prove what can be proved from the empty assumptions, and the rest is assumed false by default. We have seen that this does not work well on the program $\{a \leftarrow \sim b,\ b \leftarrow \sim a\}$. The problem is that we assumed all falsities at once. If we instead assume them one at a time (and after each assumption, derive new truths), we obtain the *stable model semantics* [12].

The actual semantics are defined slightly differently, making use of formal constructs such as fix points. However, the resulting mathematical object is the same.

The rest is taken in the context of propositional logic. This is the usual presentation of stable models; the first-order case is reduced to the propositional case by grounding the program. On a related note, there has been some work on computing stable models without explicitly grounding the program [10] [18].

Definition 9 (program reduct)**.** Given a set of atoms S representing a candidate model, the *program reduct* of program P with respect to S is the program $P|S$ constructed from P as follows: for each rule

$$l \leftarrow a_1,\ \ldots,\ a_n,\ {\sim}b_1,\ \ldots,\ {\sim}b_m,$$

if any of the negated atoms b_i are satisfied in S, we drop the rule. The rule is kept only if none of them are satisfied in S, in which case the negated atoms are dropped. That is, the following rule is in $P|S$:

$$l \leftarrow a_1, \ldots, a_n.$$

In other words, the program reduct incorporates all the negative information from S into the program P.

Definition 10 (stable model)**.** A set of atoms S is a *stable model* of P if the least model of the (positive only) program $P|S$ is exactly S.

In other words, stable models are exactly those models that can "reconstruct" themselves. Note that the least model of $P|S$ is monotonous when considered as a function of S: the larger the candidate S, the fewer negative information is there, and this in turn leads to fewer conclusions and smaller least model of $P|S$. From this, it is clear that no stable model is a subset of another stable model.

Algorithm for computing stable models. We can construct the complement of S, i.e. the set of assumed falsities, incrementally. We start with $\overline{S}_0 = \emptyset$, and derive the least model M_0 of $P|S_0$.

After each iteration i, we nondeterministically pick an atom $a \notin M_i$ to be assumed false (i.e. $\overline{S}_{i+1} = \overline{S}_i \cup \{a\}$), and then derive the new least model M_{i+1}. It makes sense to consider only those atoms a that actually modify the program reduct, i.e. ${\sim}a$ is present in the body of some rule in P. This is iterated until either all such atoms are exhausted, in which case the last model M is the stable model; or we find that M_{i+1} and $\overline{S}_{i+1}$ are inconsistent, in which case there is no solution in this branch.

Note that this is just a more detailed presentation of the idea from the beginning: assuming falsities one at a time, and deriving new truths after each step.

When comparing stable models to standard logic (with the open world assumption), the only difference is that in the latter, we are also allowed to assume propositions to

be true by default. The word "guessing" is more appropriate in that case. This goes with the intuition that if we have no information about something, it could be either true or false, whereas in the stable model semantics we always prefer the false route.

In case the program P is (locally) stratified, the stable model semantics matches the stratified negation semantics. The advantage is that stable models can be used for all programs, and can be used to compute the minimal model of a locally stratified program even if the assignment of atoms to strata is not known.

The stable model semantics is merely for intuition and ease of expression. In general, we are able to express any SAT instance by simply augmenting the program with $\{a \leftarrow \sim\neg a,\ \neg a \leftarrow \sim a\}$ for each atom a. This allows us to choose whether we want a to be true or false, thus we are no longer limited to assuming a only false.

2.5 Actual logic programming languages

In this section, we present a brief overview of representatives of the logic programming paradigm.

Prolog. Prolog is among the first logic programming languages. It is based on goal-directed proof search with first-order extended Horn clauses, with negation as failure. It is a mature language: there are many different systems implementing it, there is standardization, many libraries and extensions (such as definite clause grammars, constraint solving [28], ...).

It is untyped, thus it is up to the programmer to ensure that the predicates are used as intended. There are typed derivatives, such as Mercury [25], which is statically checked and allows the programmer to also specify *modes* of predicates: what binding patterns are allowed.

Lambda prolog. λProlog [20] is a *higher-order* logic programming language. It is based on goal-directed proof search with higher-order hereditary Harrop formulae, with negation as failure. Higher-order meaning that (limited) quantification over function and predicate types is allowed, and there is capability of solving (limited) unification problems involving higher-order variables.

This higher-order nature allows λProlog to work easily with variable bindings, substitutions, ... This makes it uniquely suited for representing formal systems making heavy use of such bindings (i.e. theorem provers, programming languages, ...).

Datalog. Datalog is a purely declarative language, sitting at the intersection of logic programming and databases. It is based on model construction with first-order (basic) Horn clauses, with safety restrictions, with stratified negation. Although based on

model construction, it has been optimized with *magic sets* [6] to handle specific queries faster than calculating the entire model.

To ensure the model is finite, complex terms (involving function symbols) are disallowed in arguments of predicates. This guarantees that a model can be found in finite time, thus a datalog query always terminates. On the other hand, this means that Datalog is *not* a Turing complete language.

There are many extensions to datalog. One extension relaxes the restriction on use of function symbols, allowing *some* controlled use of them so that the model is still guaranteed to be finite.

Answer set programming. ASP is a family of purely declarative languages. They are based on model construction with first-order (basic) Horn clauses, with safety restrictions, with strable model semantics for negation. This makes it uniquely suited for constraint satisfaction problems, such as various combinatorial puzzles.

The language is usually Datalog-like in syntax. Various language constructs are introduced for more concise representation, such as upper/lower bounds on the number of truths in a given set, integer ranges for asserting many propositions at once, disjunction in heads of rules, ...

Part II

Board games

Chapter 3

Analysis of board games

In this chapter, the goal is to identify common patterns in board game rules and discuss how they could be expressed formally. But first, we need a common understanding of the word "game". Then we identify two levels to a game, which are discussed further in sections 3.1 and 3.2. For illustrative purposes, we make heavy use of examples of games, which are described in section 3.3.

We understand games simply as multi-agent environments. In general, games usually define a measure of success for individual agents, which the agents are trying to maximize. However, we are not interested in finding optimal strategies for players; we are merely interested in modelling the environment. Thus, the goals are not necessary. In this minimalistic setting, the rules of the game dictate how the environment evolves, what do the individual agents see, how exactly can the agents interact with the environment and when the game ends.

An implementation of a game can be separated into two levels: the *environment* level and the *simulation* level; the latter determines how the game evolves in isolation, and the former determines the rest. We can view the simulation as the main program, and the environment as an oracle. Occasionally, the main program may request some information from the oracle in order to advance the game state. The main program concerns itself merely with the logic of the game; the oracle prescribes meaning to all the requests that could be possibly made by the main program.

Note that the distinction between the two is largely a matter of perspective. For example, one could view the entire game happening in the environment. Then, the simulation merely tells the players to "play the game and return the result". We draw the following line between the two: the simulation must be *deterministic* and should contain as much of the game logic as possible. The environment merely assigns meaning to primitives such as providing input from players, random choices, nondeterministic choices, ... At the intersection of the two, there must be some language which for specifying how exactly can the environment affect the simulation.

Of main interest to us is the simulation. Even so, we elaborate on the environment to obtain a better idea of what exactly it entails.

3.1 Environment level

In general, the environment may contain any elements from the outside world (in other words, the context in which the game is running). For example:

- New players may join a game while it is running.
- Each player has limited time for making decisions (e.g. chess clock).
- There may be timed events, moving the simulation forward at specific *real* times.

Any such feature must incorporate primitives that enable the simulation to work with it. For example, to implement feature 3.1, the system should have primitives for scheduling and unscheduling events. From the point of view of the simulation, the primitives are just meaningless tokens (in the sense that when a function is called, the call itself is meaningless). The meaning of these primitives is prescribed precisely by the environment.

We analyse the most basic (interactive) designs in games, and consider possible interfaces between the simulation and the environment in these contexts. With sufficiently general interfaces, the complexity of the designs can be shifted from the environment to the simulation itself.

Order of play. A common design is that the game proceeds in turns and in each turn, exactly one of the players performs an action. For example, consider chess. Generally though, the exact logic describing how control passes from one player to another may be more complicated. For example, it may be the privilege of the current player to decide the next player.

Another common design is that multiple players make their moves during the same turn. The players make their choices simultaneously, without knowledge of the others' choices. The overall combination then determines the effect on the environment. We mention rock-paper-scissors as an example.

In both of the above designs, the system specifically asks for player inputs. Thus, it suffices to have primitives for querying (possibly multiple) players, asking for their moves. It must be possible to specify which moves are valid for which player.

However, there are designs where the above primitives would not suffice. For example in dobble, any player may interact with the game at any time, as long as they respect the game rules. Here, the primitives should merely prescribe rules as to what are the allowed interactions and at what times.

Information hiding. In simple games, there is only one point of view: where we see everything about the game at any time (thus the entire history of the game as well). This is called the *game* perspective, because it is the perspective from which all simulation decisions are made. In general though, each player has their own perspective. Then, there should be primitives for notifying players of new information.

Note that when we take into account computational aspects, even if information is available, that does not necessarily mean that all its conclusions are. For example in dobble, given the descriptions of the top card and the target card, a computer could easily calculate the matching symbol and play at superhuman levels. The game is based precisely on the fact that this problem is *not* easy for humans. As a more theoretical example, consider a game where the goal is to find a hamiltonian path of some graph G (an NP-hard problem).

Therefore, the system should *not* merely present the players with a list of possible choices. Instead, a higher level representation should be used. One option is encoding the choices in a type; we are then asking the player for a term of that type.

Randomness. Random choice can be modelled as moves by a special player, called *nature*. Thus, it suffices to extend the player input mechanism.

In general, the implementation on the environment side is non-trivial, because making a random choice can be a difficult problem. Usually though, they *are* trivial (dice rolls, shuffling a deck of cards, ...).

3.2 Simulation level

In this level, all rules are interpreted in isolation, disregarding any meaning behind those rules that have special meaning in the environment. We present the common patterns occuring in board game rules, focusing on the constructions that can be easily stated in natural language and can be easily understood in the context of the game, but are difficult to formalize.

The ultimate goal is to design a representation langauge which would be flexible, but also precise. By flexibility, we mean that it can express arbitrary rules and can easily express high-level modifications to game rules. We illustrate the flexibility of natural language on some of the constructions, and show how this can pose a problem for the precision of the language.

The statements in a ruleset can be roughly divided into two categories: *declarative* statements and *imperative* statements. An important goal is to find a unified approach to them, which would treat both kinds of statements merely as different faces of the same thing.

3.2.1 Declarative statements

Declarative statements say *what* holds in a given game state. They say things about the past and the now. They form the language which allows us to interpret the state in higher-level notions, by defining relations among objects.

Example 1 (chess)**.** A king is under check if an enemy piece could reach it in one move (disregarding any restrictions on that move). A player is *checkmated* if their king is under check, and no move by the player can change this. The game reaches a *stalemate* if the turn player's king is *not* under check, but any move by the player endangers the king. The game ends either the turn player is checkmated or when the game reaches a stalemate.

Example 2 (sokoban)**.** The game ends when all of the boxes are on target cells. Alternatively, it ends when all of the target cells contain boxes. A cell is *passable* if it is not a wall. The player character can *reach* any of the four adjacent cells, provided they are passable. (Note that reaching is not the same as moving; the latter must take into account boxes as well.)

Example 3 (battleships)**.** The game ends when a player has lost all their ships. A ship is lost if one of the cells it occupies has been hit.

Declarative statements are naturally expressed in logic. Thus the fields of logic programming and deductive databases are relevant.

Game history. One option of representing the game is to represent only information that could be relevant in the future. That is, only the current state of the game is represented, not the entire history. However, this is problematic when the game rules are modified: new rules may refer to past information that is not accessible under the current game representation. That is, the representation is too narrow; it must be adjusted. This could be avoided, however, if the entire history of the game were part of the representation.

In fact, everything can be computed from the game history alone. For example, given the an entire game of chess (in some notation), it is possible to replay the entire game and calculate the position of some pawn at any point in time. Thus, an explicit representation of the current state is not necessary; it is merely a certain perspective of the game. However, it makes sense from a representation point of view: the current state aggregates most of the past information relevant to the future, and makes it easy to work with.

Example 4 (chess)**.** In chess, there is the rule of *threefold repetition*: if the same position occurs three time during a game (not necessarily in succession), then the

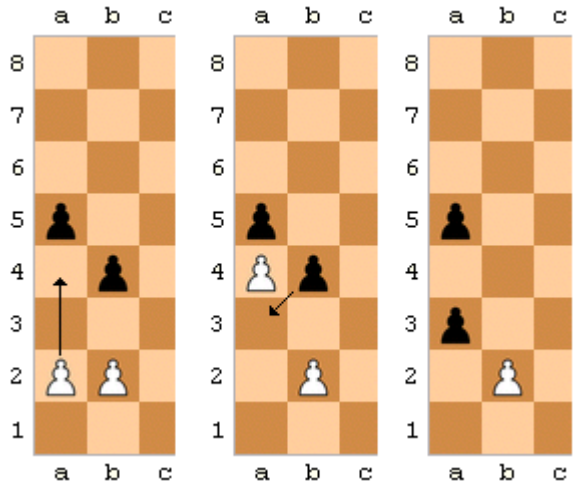


Figure 3.1: The *en passant* rule in chess. [1]

game is a draw. Another rule is *castling*, which moves the king and one of the rooks. It requires that neither the king nor the chosen rook have moved yet. There is also the *en passant* rule: immediately after a player moves a pawn two squares forward from its starting square, the pawn can be captured by an enemy pawn as if it had advanced only one square. The rule is illustrated in figure 3.1.

Hypothetical reasoning. Many rules are naturally expressed using hypothetical reasoning. One kind of such rules are rules that forbid moves that would lead to an invalid game state.

Example 5 (chess)**.** A player cannot make a move that would endanger their own king (i.e. put him in check). This applies to *all* moves, including special moves like castling and en passant.

Example 6 (sokoban)**.** The player *can* move to a cell with a box, which pushes (moves) the box one cell in that direction. However, a box cannot move to a wall nor a cell occupied by another box; this restricts the moves of the player (i.e. which boxes can be pushed). This is more natural than stating: "The player cannot move to a cell with a box if the next cell in that direction also contains a box."

Example 7 (battleships)**.** At the beginning of the game, each player places their ships secretly. An arrangement of ships is valid only if no two ships touch nor overlap each other; a player cannot place a ship where this would violate this restriction.

The formal semantics for hypothetical reasoning can be given by goal-directed proof search with hereditary Harrop formulae (see 2.3.2). However, the system could mix together distinct semantics for distinct notions (e.g. hypothetical reasoning with stable model semantics), and the semantics as a whole are non-trivial.

Exceptions. In natural language, it is easy to state *exceptions* to rules. These change how we interpret the game rules: the original rules still apply, but the exceptions are "on top of" them. Modifying the original rules for every little thing can be cumbersome, and exceptions provide a natural solution.

One way to look at it is that rules become *defeasible*, with the exceptions taking priority over the original rules. There can be exceptions to exceptions, ... in general, it should be possible for an arbitrary reasoning process to take place.

Example 8 (restrictions)**.** In general, any restriction (such as forbidden moves) can be viewed as an exception. For example, in chess, we may initially define valid moves without regard to the king's safety. Later, we make an exception: the moves are valid only if they do not leave the king in check.

Example 9 (magic chess++)**.** We can introduce exceptions stating that under specific cirumstances, the strength of the individual pieces is different:

1. If two pawns attack the king, then their total strength is 15 instead of 12.
2. If the player's health is exactly 1, then the player's pieces have double strength.
3. If the player's health is below 10, then the player's pieces have +1 strength.

However, too many exceptions can lead to unintended interactions and vague rules. In the above example what should be the combined effect of exceptions 2 and 3? Should one of them completely override the other and if so, which one should take priority? Or should there merely be some order in which they should apply? And if so, what should be the order?

We see that exceptions pose a challange to the precision of the language, in that it is not clear what should be the exact semantics for exceptions.

3.2.2 Imperative statements

Imperative statements say things about the future: what basic relations among objects should hold in the next state, creating new objects, ... Note that they can be embedded inside declarative statements (e.g. hypothetical reasoning); the result is still a declarative statement. The game progression (i.e. main loop) is naturally expressed imperatively.

Example 10 (battleships)**.** At the beginning of the game, each player secretly places their ships, respecting the game rules. Then, one of the players is randomly designated as the active player. Each turn, the active player shoots at one of the cells on the opponent's board, hitting that cell; the next turn, the other player becomes the active player.

Unlike purely imperative languages, board game rules are often *unfocused*. Even though there is usually some structure to the game progression, events can "pop-up" at any time due to circumstances in the game. These events modify the game state independently of the main progression. We categorize them into two types based on their effect: *triggers* and *replacement effects*.

Triggers. Triggers merely create new statements to be executed. They are similar to (normal) declarative statements, in that they have the form: "If some condition holds, then something."

Example 11 (chess++)**.** Whenever a pawn captures a piece, the pawn becomes that piece type. After each normal move, the turn player may move their king (usual rules apply to the move).

Example 12 (battleships++)**.** Whenever a player hits an opponent's ship, they may shoot again. At the beginning of each turn, the inactive player may move one of their (non-lost) ships forwards or backwards 1 cell, in the direction of the ship. The move must not violate the restrictions on the arrangement of the player's ships.

Replacement effects. The relation between replacement effects and imperative statements is similar to the relation between exceptions and declarative statements: both modify the statement, under certain conditions. They are also closely related to hypothetical statements.

Example 13 (sokoban++)**.** If an object would move to a cell with another object, instead the other object is moved 1 cell in the same direction. (This generalizes the "player pushing boxes" rule: now, boxes can push around other boxes as well. Thus, a player is able to move an entire row of boxes.)

Example 14 (chess++)**.** If a king would capture a queen, instead, the queen changes color to match the king's color. If a pawn would be captured, if the previous cell in the pawn's column is empty, the pawn moves there instead of being removed from the board. (The capturing piece moves normally.)

Similarly to exceptions, the exact semantics of replacement effects are not clear. They pose a challange to the precision of the language.

3.3 Examples

For illustrative purposes, we draw mostly from the following games. Note that these are not necessarily representative of modern board games, which usually have far more

complicated rules. Nevertheless, they suffice for illustrative purposes. The descriptions of the games are by no means unambiguous, perfectly accurate, nor complete; the emphasis is on conciseness and big picture.

Game 1 (chess)**.** The game is played on an 8×8 chessboard. There are 6 types of chess pieces: pawn, rook, knight, bishop, queen and king. Each piece is either white or black, corresponding to the player that controls it.

The players take turns. Each turn, the turn player moves one of their pieces. Each piece has its own rules governing what moves are legal and what moves are not. There is a mechanism for *capturing* enemy pieces. The first player to have their king captured unconditionally (i.e. no matter how that player moves) loses.

Game 2 (magic chess)**.** An (arbitrary) extension of chess. Each player has initially 20 health. The individual chess pieces have varying strength: pawns 6, knights and bishops 4, rooks 3, queen 2 and king 1. At the end of each turn, the turn player loses health equal to the total strength of all opponent's pieces that attack the king. If they reach 0 or less health, they lose.

Game 3 (sokoban)**.** The game is played in a level with dimensions $n \times n$. There is only a single player, who controls a character that can move in the four cardinal directions and can push around boxes scattered around the level. There are also walls and target cells; the goal of the player is to achieve a state where each target cell contains a box.

Game 4 (battleships)**.** Each of the two players has an $n \times n$ board and a set of ships. The player boards are kept secret from the opponent. Initially, each player (secretly) places their ships on their board. Afterwards, the players take turns. The turn player shoots at any cell on the opponent's board. The opponent that declares whether it was a hit or a miss. The first player to lose all their ships loses.

Game 5 (dobble)**.** There is a deck of 55 cards, each of them depicting 8 symbols. The deck is designed so that each two cards share exactly one symbol. There are n players, each of them is dealt a stack of k cards so that only the topmost card is visible. An initial *target* card is dealt from the deck.

At any time, a player may exclaim a symbol a place their topmost card x on top of the target card y. The exclaimed symbol must be precisely the one shared by the two cards. The card x then becomes the new target card. The first player to get rid of all cards in their stack wins.

Chapter 4

Related work

In this chapter, we compile literature that deals at least partially with the problem of game representation.

4.1 General Game Playing

General Game Playing [13] (GGP) is concerned with designing artificial intelligence that tackles the following task: given a description of a game, play the game as well as possible. This is in contrast to AI designed to play a specific game, where the designer of the AI system has much more information to work with; there, heuristics specifically designed for the game at hand can be implemented. On the other hand, in GGP, the heuristics must be derived without human help.

One of the main concerns of the GGP framework is the speed at which it is possible to evaluate the game states, calculate legal moves, update states, ... The faster this can be done, the larger the part of the game tree can be explored by the AI. Thus, languages for GGP strive to be fast, as general as possible to allow implementation of complex games and to provide a challenge, and some of the languages are specifically designed for derivation of heuristics.

4.1.1 Game Description Language

The first in a series of related languages is the *Game Description Language* [19]. It is able to express only discrete games with perfect information and no randomness.

In GDL, the state of the game is represented as a set of logical facts that are true. The rules of the game are also represented as either facts or rules of inference, which are, unlike the facts about the game state, always part of the state. The semantics are those of Datalog, with negation as failure, with functional symbols, and restricted recursion. This is the simulation level; some predicates and terms have special meaning (provided by the environment level):

1. Predicate `init` designates those facts that are true at the beginning of the game.
2. Predicate `role` denotes the players of the game.
3. Predicate `base` enumerates all facts that could possibly become true over the course of the game; this is for efficiency reasons. Similarly, `input` enumerates all actions that could possibly be legal, for individual players.
4. `true` represents what is true in the current state of the game. Similarly, `does` represents the last actions of the players. There is a special action `noop`, which means the player took no action (e.g. when it is not that player's turn).
5. `legal` represents the moves that are legal for the individual players, in the current state.
6. `next` represents what will be true in the next state.
7. The proposition `terminal` represents the end of the game. Predicate `goal` specifies for each player the payoff received.

GDL-II

GDL-II stands for *Game Description Language with Incomplete Information* [26]. It is an extension of GDL, capable of describing all discrete games with both randomness and imperfect information [27]. Towards this end, the predicates `percept`, `sees` and the term `random` are assigned special meanings:

1. `percept` determines the set of possible percepts that the player can receive from the game. It is similar in function to the relations `base` and `input`, in that it merely enumerates all possibilities.
2. The relation `sees` determines for each player what information is passed to them at the end of the turn.
3. The term `random` is a role that always chooses uniformly at random from its legal moves.

GDL-II is the currently supported extension of GDL. There is a large database of implemented games. Most of them are, however, abstract games or games with simple rules, compared to modern board games. For example, there are no constructs for executing multiple effects in a row (a very basic construct in imperative programming), no exceptions nor replacement effects, no access to game history, no hypothetical reasoning. Even so, the games that *can* be expressed provide enough of a challenge for GGP players.

Subsequent variants of GDL are not part of the GGP standard.

SDL

SDL stands for *System Definition Language* [2], an extension of GDL-II. The distinguishing point is that it remembers the entire history of the game.

There are only few differences from GDL-II. The `true`, `does`, `sees`, `legal`, `goal` and `terminal` predicates are expanded with an additional argument: the time step. There is the special unary predicate `step`, which denotes precisely the time steps. The chronological ordering of the steps is determined by the binary predicate `successor`. The `init` predicate becomes obsolete, as it becomes a special case of the `true` relation for the initial time step. The remaining predicate (`role`, `base`, `input` and `percept`) remain unchanged.

rtGDL

rtGDL stands for *Game Description Language for Real-time Games* [16], an extension of GDL. It is able to describe real-time games with perfect information and no randomness.

In rtGDL, `true` and `init` facts are extended by an additional real-valued argument: the duration for which the fact will be true, in seconds. The special duration `infinity` is introduced. The meaning is that a fact with an infinite duration does not expire naturally; though it may expire due to other effects, such as when its duration is set to another value in a state update.

The state changes either when a player performs an action, or when any of the facts expire due to time constraints. At each of those times, a new state is calculated as usual. The special predicate `expired` indicates which facts have just expired, and the durations of the `true` facts are updated to reflect the actual remaining time.

4.1.2 Toss

Another language for GGP, not in the family of GDL-like languages, is *Toss* [15]. It was designed so that it is easier to extract useful heuristics from the game description, resulting in stronger AI. The language is able to describe sequential games with perfect information and no randomness.

We describe the game representation only; for a complete syntax specification and examples of games, consult the Toss website [3]. The game is divided into two parts: the *control state* and the *board state*.

Control state. The control state dictates the high-level game progression. It is represented as a directed graph with labelled edges, where the states correspond to vertices (called *locations*) and possible moves correspond to outgoing edges. Each

location belongs to exactly one player, who is in control and decides which action should be taken. The labels of edges describe the effects of the action on the board state, and also conditions that must be satisfied in order for the action to be legal. One of the locations is designated as the starting location. The game ends when no legal action can be taken from the current location, at which point the players receive payoffs (which are determined by the location).

Board state. The board state is formally represented as a relational structure, which can be viewed simply as a set of objects together with relations on them. Modifications to the board state are based on *structure rewriting*: a rewriting rule $\mathfrak{L} \to \mathfrak{R}$ states what part of the board state can be rewritten (left side) and what is the replacement (right side).

Rewriting. A structure rewriting rule $\mathfrak{L} \to_s \mathfrak{R}$ is a pair of structures $\mathfrak{L}$, $\mathfrak{R}$ of the same signature, together with a partial function $s : \mathfrak{R} \to \mathfrak{L}$. The idea is that applying the rule "cuts out" some substructure that matches $\mathfrak{L}$ and replaces it with $\mathfrak{R}$. The function s tells which objects in $\mathfrak{R}$ correspond to which objects in $\mathfrak{L}$ so that when the replacement occurs, some of the original relations are preserved.

There is a mechanism for specifying how perfect must be the matching of $\mathfrak{L}$ to the current board state: we specify a set of relations τ that must match exactly (i.e. both $p(\ldots)$ and $\neg p(\ldots)$ must agree), and the remaining relations must match only positively. Formally, a τ-embedding of structure $\mathfrak{L}$ to structure $\mathfrak{A}$ is an injective function $\sigma : \mathfrak{L} \to \mathfrak{A}$ satisfying the following constraints:

$$p(a_1, \ldots, a_{n_p}) \text{ in } \mathfrak{L} \quad \Longrightarrow \quad p(\sigma(a_1), \ldots, \sigma(a_{n_p})) \text{ in } \mathfrak{A} \qquad \forall p \notin \tau, \tag{4.1}$$

$$p(a_1, \ldots, a_{n_p}) \text{ in } \mathfrak{L} \quad \Longleftrightarrow \quad p(\sigma(a_1), \ldots, \sigma(a_{n_p})) \text{ in } \mathfrak{A} \qquad \forall p \in \tau, \tag{4.2}$$

where n_p denotes the arity of relation p.

Actions. The effect is described by a structure rewriting rule and the set τ of relations to be embedded exactly. The legality of the action is determined by two logical formulas: *preconditions* and *postconditions*. The former specifies what must hold now, whereas the latter specifies what must hold in the resulting state (in order for the action to be legal).

Auxiliary relations. Other than serving as constraints for actions, the logical formulas can be used to define auxiliary relations. These relations can only be used on the left hand side of the rules, with the meaning that these relations must be exactly embedded by σ as well (i.e. they are added to τ).

4.1.3 Regular Boardgames

Regular Boardgames [17] (RBG) is can describe some discrete games with perfect information and no randomness. It is well suited for games where the main complexity is in the arrangement of pieces on the board and moving the pieces around, for example chess. On the other hand, it lacks the declarativeness of a logical language like GDL.

Similarly to Toss, the state of the game is divided into two parts: the control state and the board state.

Control state. RBG models all possible lines of play as a regular expression. A single line of play corresponds to a word h, whose symbols represent the operations that have been performed on the board state (the history of the game). Some operations change the board state (change the active player, move a piece, ...), others merely check that some property of the board state is satisfied and if not, the line of play is invalid.

At each point in time, exactly one player is in control. That player chooses the next operations to perform until another player becomes in control. The choices must always lead to a valid state (i.e. respecting the regular expression and respecting possible checks).

Board state. The board is a directed multigraph with labelled edges and labelled vertices. Edge labels are called *directions*. For each vertex and each direction, there may be at most one outgoing edge in that direction. The vertex labels are called *pieces* (e.g. empty square, black knight, white pawn, ...). Other than that, the board state also includes bounded integer variables that are initially 0. Exactly one vertex of the board is selected at all times. Most operations on the board are local, modifying or checking only the surroundings of the selected vertex.

Operations. The following operations can be performed on the board state.

Shift. Checks if there is an outgoing edge in the given direction from the selected vertex. If yes, moves the cursor in that direction.

On. Checks if the piece on the selected vertex is from a particular subset of pieces.

Off. Puts the specified piece on the selected vertex.

Assignment. Evaluates an arithmetic expression and assigns the value to a variable.

Comparison. Checks an inequality between two arithmetic expressions.

Switch. Passes control to the specified player, or *keeper*, which is a special player that makes arbitrary choices. The role of the keeper is to perform computation.

Pattern. Has the form $\{!M\}$ or $\{?M\}$, where M is a regular expression of operations that does not contain switches. The former checks if all words matching M are invalid, the latter checks if at least one word matching M is valid.

Part III

Our findings

Chapter 5

Causality graphs

In this chapter, we introduce *causality graphs*. We start with the motivation, followed by formal definitions in section 5.2. In section 5.3, we show that our definition of seeing has many intuitive properties. Then in section 5.4, we consider the computational problem of which pairs of objects see each other and related problems. The syntax and semantics are discussed in section 5.5. Finally, we discuss future work in section 5.7.

5.1 Motivation

It is natural to start from logic when designing a declarative language. However, there is an issue with this. From the point of view of logic, all objects of interest already exist, and we are merely talking about properties of these objects. There is no natural notion of *creating* new objects; its point of view is static. This is even the case in modal temporal logic, the only difference is that the properties we are talking about are in the context of time.

This is not to say that a notion of creating objects is impossible in logic, merely that it is unnatural and that it "does not exist at the highest level". We draw a parallel with classical and intuitionistic logic: even though there is an embedding of classical logic into intuitionistic logic, both of the logics are of interest (and of different interests).

A much more natural fit is to use the language of *type theory*. There, all objects are inductively constructed, starting from the most simple ones. For example, the natural numbers $\mathbb{N}$ are constructed as follows:

1. Zero is a natural number (i.e. $0 : \mathbb{N}$).
2. If x is a natural number, then so is $(\text{suc } x)$ (i.e. $\text{suc} : \mathbb{N} \to \mathbb{N}$).
3. Nothing else is a natural number (implicitly).

In most applications of type theory, the order in which the objects are constructed does not matter; the only thing that matters is whether an object exists (can be constructed

in a finite number of steps) or not. In our case, the objects are in the context of a game, thus time plays a crucial role. We say that y *sees* x if it was constructed either after x, or at the same time as x. We denote this $y \measuredangle x$. What should be the definition of seeing? What should be the structure of time?

Let us first illustrate why this is important. Suppose that we want to implement the following persistent effect: "At the end of each turn, for each coin, create a new coin." At first, it seems this can be easily implemented as the following function:

$$f : \text{endOfTurn} \to \text{coin} \to \text{coin}.$$

However, if x is the object representing the end of the turn and c is some coin, then the above yields infinitely many new coins:

$$f\,x\,c, \quad f\,(f\,x\,c), \quad \ldots, \quad f^n\,x\,c, \quad \ldots$$

The problem is that $f\,x$ considers *all* coins, not only coins visible from the end of turn object x. In order to solve this problem, two things need to be done: first, seeing must be incorporated into the (formal) language and second, there must be a reasonable definition of seeing. For now, we postpone the problem of language until later and use ad-hoc notation:

$$f' : (x : \text{endOfTurn}) \to (y : \text{coin}) \to \text{coin} \qquad \text{if } x \measuredangle y.$$

Let us focus on the definition of seeing instead.

Relevant seeing. The inductive method of constructing objects yields a very natural notion of time: one object x is constructed before another object y if the latter requires the former. That is, $y \measuredangle x$ if x is used for the construction of y. We call this definition of seeing *relevant seeing*, because each construction sees only those objects which are relevant for its construction. In a sense, it is the minimal reasonable definition of seeing, in that any seeing relation should be an extension of it.

Problem with innate properties. There are some issues with relevant seeing, as illustrated by the following example. Suppose that there is an apple a, and a rule that states an innate property of all apples: that they are green. The rule can be encoded simply as a function of the dependent type $(x : \text{apple}) \to \text{green}\ x$. Since the property is innate, we would expect that merely seeing the apple x would allow us to see that it is green. With the above definition of seeing, this is not necessarily the case: any object constructed from the apple without requiring the knowledge of its greenness is a counterexample.

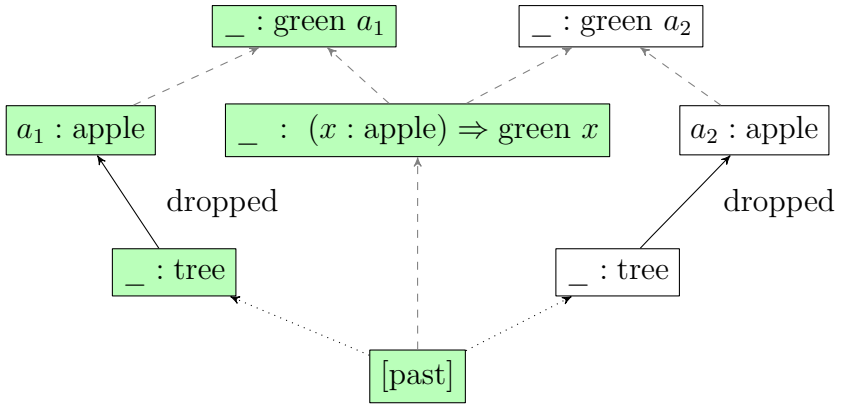


Figure 5.1: Example illustrating relevant seeing with immediate consequence. Dashed gray edges are the $\Rightarrow$ constructions, black edges are the $\to$ constructions. Dotted edges represent indirect construction (i.e. through possibly multiple edges). Objects visible from a_1 are highlighted.

Immediate consequence. To avoid the above problem, we amend the definition of seeing by adding a new type of arrow, denoted $\Rightarrow$, representing immediate (logical) consequence. Define seeing as follows: $y \measuredangle x$ if either x is an ancestor of y, or y is constructed from (some, not necessarily all) ancestors of x purely through the use of the immediate consequence arrow $\Rightarrow$.

This arrow can be used precisely for modelling innate properties, such as the greenness of an apple. In the above example, if we instead had $f : (x : \text{apple}) \Rightarrow \text{green } x$, then both a and $f\,a$ see each other. Thus, any object that sees either of them necessarily sees the other one. A related example is illustrated on figure 5.1.

Problem with aggregation. However, there are still some issues. Suppose that at the beginning of the game, n coins are produced as follows:

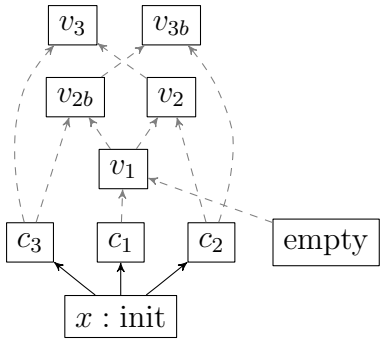
$$x : \text{init},$$
$$f : \text{init} \Rightarrow \mathbb{Z}_n \to \text{coin}.$$

The goal is to construct an object that can see all of the coins produced by init. The solution should work for all values of n. This is of interest because we may be interested in the number of these coins, their total monetary value, ... A necessary first step is to have some point of view (some object) from which we can see precisely those coins.

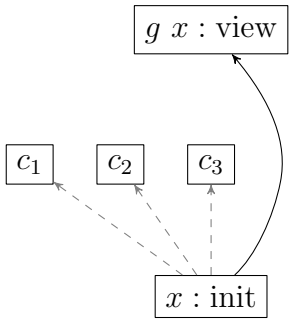
At first, it seems that the following suffices:

$$\text{empty} : \text{view},$$
$$\text{join} : (x : \text{view}) \Rightarrow (c : \text{coin}) \Rightarrow \text{view} \qquad \text{if } x \text{ does not see } c.$$

This does indeed yield some object that can see all coins, but there are many issues. First, there are too many ($n!$) maximal views. Second, if new coins were generated



(a) A proposed naive solution when the arrow $\rightarrow$ is used to produce the coins c_i. Only two (v_3 and v_{3b}) of the six maximal views are shown.



(b) A simple solution when the arrow $\Rightarrow$ is used to produce the coins c_i.

Figure 5.2: Relevant seeing with immediate consequence, aggregation example. For brevity, types of objects are mostly omitted; the same goes for functions (f, g and join).

sometime later during the game, the maximal views would change and the point of view would suddenly shift far into the future. There is no upper bound; this is due to the open-world nature of the game.

This is not a formal proof that such a solution does not exist, but it does illustrate the problems one can encounter when working towards a solution. If there is a solution, it is not as straightforward as this. However, if we adjust the problem only slightly, then a simple solution reveals itself. This shows us the direction in which it is worthwhile to generalize the seeing relation. The adjustment is as follows.

Suppose that the coins were produced instantly. That is, they were produced not using the function f, but using

$$f' : \text{init} \Rightarrow \mathbb{Z}_n \Rightarrow \text{coin}$$

instead. The only difference between the two functions is in the second arrow. With f', constructing the desired object is simple: just use the "higher" arrow $\rightarrow$.

$$g : \text{init} \rightarrow \text{view}.$$

It is because of the existence of an arrow "higher" than $\Rightarrow$ that there is a simple solution for $\Rightarrow$. If there were arrows "higher" than the usual $\rightarrow$, then the above solution could be transferred to the case of $\rightarrow$ as well. This is the direction in which we generalize relevant seeing, but before that, we consider one other option.

The examples are illustrated in figure 5.2.

Linear time. Another intuitive option is to have all time points arranged linearly, forming a total order. Then any two objects are comparable: either one of them was constructed before the other one, or they were constructed at the same time.

To avoid the problem with innate properties, we again use two types of arrows. The arrow $\Rightarrow$ constructs the object in the same time point, whereas the arrow $\rightarrow$ is guaranteed to construct it in some future time point. The exact semantics of $\rightarrow$ are not important, but for illustration, we mention two possible semantics:

1. The object is constructed in the next time step.
2. The object is constructed nondeterministically in *some* future time step, not necessarily the next one.

Problem with paralellism. The problem with linear time is that it does not allow for events happening in parallel (i.e. in isolation from one another). To illustrate, suppose that there is initially 1 coin in the game. Let there be two copies of the following (persistent) effect: "At the end of each turn, for each coin, create a new coin, then repeat once." This is a composite effect, meaning that it is composed of two (identical) components. These should be executed sequentially, with the later components waiting for the earlier components to finish before starting. This is so that the consequences of the earlier components are visible to the later components.

Consider the situation at the end of the first turn. Two events X, Y are triggered. If they are parallel to each other, then their consequences should not be visible to one another. Only some later event occuring after both of them, such as the beginning of the next turn, should see the total result. This means that each event produces $1 + 2 = 3$ new coins, yielding a total of 7 coins at the end.

However, if any two time points must be comparable, then this is impossible. Suppose that c_1 is the first coin produced, and without loss of generality let it be produced by the event X. But then, the first coin produced by Y, denoted c_2, is produced either later or at the same time. In either case, it can observe c_1. Since the second component of Y occurs after c_2, it too can observe c_1; thus it will produce additional coins.

Note that the argument applies to both semantics 1 and 2. The only assumption is that for any composite effect, its components occur one after another, with the later components waiting for the earlier components to finish.

Problem with relevance. Another problem of linear time is that too much is visible. An object sees not only their ancestors and some closely related entities, but it can potentially observe completely unrelated entities. This makes reasoning from the point of view of an object harder, because there are more uncertainties. (The parallelism problem can be viewed as one face of this problem.)

Due to these problems, the actual definition of seeing is based on relevant seeing instead of on linear time.

5.2 Definition of seeing

5.2.1 Graphs

In this subsection, we list all used notations and notions from graph theory. All of the following definitions are in the context of an arbitrary directed acyclic graph $G = (V, E)$, where V is the set of nodes and E is the set of (labelled) edges.

Definition 11 (nodes and edges)**.** The set of all edges coming from node u is denoted $E_0(u)$ and the set of all edges going to it is denoted $E_1(u)$. The source node of an edge $e = x \to y$ is denoted $\mathrm{src}(e) = x$ and the destination node is denoted $\mathrm{dest}(e) = y$. We say that x is a *parent* of y and that y is a *child* of x.

An *ancestor* of x is either x itself or any ancestor of one of its parents. If the node x itself is disallowed, we say that it is a *strict ancestor*. Similarly, the notions of *descendant* and *strict descendant* are defined, with children instead of parents in the definitions.

Definition 12 (paths)**.** The set of all paths is denoted $\mathcal{P}$. The functions $\mathcal{P}_0, \mathcal{P}_1 : V \to 2^{\mathcal{P}}$ assign to each node the set of paths starting in it and ending in it, respectively. The function $\mathcal{P}_{0,1} : V \to V \to 2^{\mathcal{P}}$ assigns to each pair of nodes the set of paths from the first node to the second node.

The starting node of a path p is denoted $\mathrm{start}(p)$; the ending node is denoted $\mathrm{end}(p)$. The set of all its edges is denoted $E(p)$; the set of all its nodes is denoted $V(p)$. The length of the path (i.e. number of its edges) is denoted $|p|$. The empty path starting and ending in the node v is denoted $\varepsilon(v)$.

Definition 13 (initial nodes)**.** A node u is *initial* if it has no incoming edge. A path p is *grounded* if it starts in an initial node. The set of all initial nodes that are ancestors of a node u is denoted $\mathrm{base}(u)$.

Definition 14 (subpaths)**.** If $u, v \in V(p)$ and u is before v, then we denote $p[u \dots v]$ the subpath of p that starts in u and ends in v. If u is the starting node, we omit it and write instead $p[\dots v]$. If v is the ending node, we omit it and write instead $p[u \dots]$; we call such a subpath a *tail* of p. The set of all tails of p is denoted $\mathrm{tails}(p)$.

Definition 15 (path operations)**.** If the final node $\mathrm{end}(p)$ is the same as the starting node of some path q, then we denote $p \cdot q$ the concatenation of the two paths.

If path q starts in some node of p, then we say that *splicing* q into p results in r if $r = p[\dots \mathrm{start}(q)] \cdot q$. We denote the operation of splicing $p[\dots / q]$.

Definition 16 (subgraphs)**.** Given a node u, the subgraph induced by nodes before u is called the *subgraph before* u, and is denoted $\mathrm{before}(u)$.

An edge e is *half-in* the set of nodes S if at least one of the nodes incident to e are in S. Given a node u, consider the subgraph before(u); add edges that are half-in before(u) and their respective destination nodes. The resulting graph is called the *subgraph half-before* u, and is denoted halfBefore(u).

The definitions are naturally translated to the case of sets of nodes. That is, before(U) is a subgraph of nodes that are before *at least one of* nodes $u \in U$, and halfBefore(U) extends this subgraph by including edges that are half-before U and their respective destination nodes.

5.2.2 Core

This subsection lists all the definitions leading up to the definition of seeing.

Definition 17 (causality graph)**.** A *causality graph* is a directed acyclic graph (V, E) with edges labelled with *durations*. The set of all durations is denoted $\mathcal{D}$. The nodes of the graph represent the objects and the labelled edges represent the causation (construction) relationships among them. The function $\delta : E \to \mathcal{D}$ assigns to each edge its label.

Definition 18 (durations)**.** The set of all durations is constructed from the set of *atomic* durations $\mathcal{D}_0$, as follows: $\mathcal{D} = 2^{\mathcal{D}_0}$.

Note that formally, atomic duration are *not* durations. This is because non-atomic durations are *sets* of atomic durations. There is a natural correspondence though: each atomic duration d corresponds to the non-atomic duration $\{d\}$.

To avoid confusion in contexts where the distinction between atomic and non-atomic durations is important, we call non-atomic durations *composite*. As a shortcut, we call atomic durations *atoms* and composite durations *composites*.

Definition 19 (ordering of durations)**.** The *domination* relation $\gtrsim \subseteq \mathcal{D}_0 \times \mathcal{D}_0$ is transitive, but not necessarily reflexive nor antisymmetric. This relation is naturally extended to composites, as follows: if D_1, D_2 are composites, then $D_1 \gtrsim D_2$ if for each $d_2 \in D_2$, there is some $d_1 \in D_1$ that dominates d_2.

Definition 20 (path duration)**.** Denote the set of all paths in the causality graph $\mathcal{P}$. With slight abuse of notation, the function $\delta : \mathcal{P} \to 2^{\mathcal{D}}$ assigns to each path in the graph its *duration*, defined as follows:

$$\delta(p) = \bigcup_{e \in E(p)} \delta(e),$$

In other words, it assigns the union of durations of edges on the path. Thus if we have a path

$$p \quad := \quad v_0 \xrightarrow{D_1} v_1 \xrightarrow{D_2} \ldots \xrightarrow{D_n} v_n$$

then $\delta(p) = D_1 \cup D_2 \cup \ldots \cup D_n$.

Definition 21 (path domination)**.** A path p dominates q, written (with slight abuse of notation) $p \gtrsim q$, if they start at the same node and $\delta(p)$ dominates $\delta(q)$. A path p *tail-dominates* path q if p dominates some tail of q.

Definition 22 (node seeing)**.** Node u *sees* node v, written $u \measuredangle v$, if for every grounded path $q \in \mathcal{P}_1(v)$, there exists a (not necessarily grounded) path $p \in \mathcal{P}_1(u)$ that tail-dominates q. The set of all nodes visible from u is denoted $\mathrm{prp}(u)$, and called the *perspective* of u.

The set of all paths leading to u represents the *construction time* of the node, in the following sense: each path provides a lower bound on the time when the node is constructed, and the actual construction time is the supremum of these times.

5.3 Properties

We now show some basic properties of the seeing relation. This is mainly to illustrate that the relation is not arbitrary, and that it has certain properties that one would intuitively expect from a "seeing" relation.

Intuitively, a node u should be able to see all its ancestors, because those are the nodes that were involved in u's construction. Thus it could be said that they are "in the past" of u.

Proposition 1. *If v is an ancestor of u, then $u \measuredangle v$.*

Proof. By definition, $u \measuredangle v$ iff for each grounded path $q \in \mathcal{P}_1(v)$, there exists a path $p \in \mathcal{P}_1(u)$ that tail-dominates q. Since v is an ancestor of u, the path q can be extended to a path $q \cdot q_\Delta$ leading to u. The path q_Δ then tail-dominates q, since $q = q \cdot \varepsilon(v)$ and any path dominates the empty path $\varepsilon(v)$. □

Corollary 1. *The relation $\measuredangle$ is reflexive.*

Proof. A special case of the above proposition where $u = v$. □

However, being able to see all ancestors is not enough. Suppose that we know that x is an apple, and we also know that all apples are green. Then, we should also know that x is green. It is an *immediate* consequence of the previous two constructions, thus it is impossible to observe the two without also seeing that x is green. If the two were among our ancestors, we would *have to* see that x is green, even if that construction were not among our ancestors.

One may have reservations about the use of "knowing" instead of "seeing". However, in some sense, knowing *is* seeing; thus properties expected of the former should be transferable to the latter.

The following theorem states that immediate consequence can be represented using edges with the empty duration.

Theorem 1. *Let v be a node and let $e_1 = x_1 \xrightarrow{D_1} v, \ldots, e_n = x_n \xrightarrow{D_n} v$ be all its incoming edges. Suppose that the first k of them have the empty duration: $D_1 = \ldots = D_k = \emptyset$. Then, we can characterize $u \trianglelefteq v$ as follows.*

Consider a a modified graph G' where the first k edges are missing. Then, $u \trianglelefteq v$ in the original graph G iff $u \trianglelefteq v$, $u \trianglelefteq x_1$, ... and $u \trianglelefteq x_k$ in the modified graph G'.

Proof. By definition, $u \trianglelefteq v$ iff each ground path q to v is tail-dominated by some (not necessarily the same) path to u. Consider cases based on the individual paths $q = q' \cdot e_i$.

If the path q passes through one of the edges with the empty duration, we have $\delta(q) = \delta(q') \cup \delta(e_i) = \delta(q')$. Thus the duration of the path is the same as the duration of the prefix q' that leads to the parent x_i. Therefore, a path p tail-dominates q iff it tail-dominates q'. We conclude that all such paths q are tail-dominated iff all paths to $x_1, \ldots, x_k$ are tail-dominated, which is equivalent to $u \trianglelefteq x_1$, ... and $u \trianglelefteq x_k$ (in either graph G or G').

Now for the cases where the path q passes through one of the remaining edges e_i (with a non-empty duration). Such paths are in a 1-to-1 correspondence with ground paths to v in the modified graph G'. We conclude that all such paths q are tail-dominated iff $u \trianglelefteq v$ in the modified graph. □

Corollary 2. *If a node v has all its incoming edges with the empty duration, then $u \trianglelefteq v$ iff u sees all of v's parents.*

Initial nodes represent the most basic building blocks, or in a logical interpretation, the most basic assumptions (axioms). It is then natural that if two constructions u, v each use some building block that the other one does not, then neither should be able to see the other.

Proposition 2. *If $u \trianglelefteq v$, then* $\mathrm{base}(u) \supseteq \mathrm{base}(v)$. *Alternatively, if x has some initial ancestor that is not a ancestor of u, then it cannot be the case that $u \trianglelefteq v$.*

Proof. If $x \in \mathrm{base}(v)$, then there is a (ground) path q from x to v. But from $u \trianglelefteq v$, we have that there is some path p that tail-dominates q; then, the path $q[\ldots/p]$ is a path from x to u. Thus $x \in \mathrm{base}(u)$. □

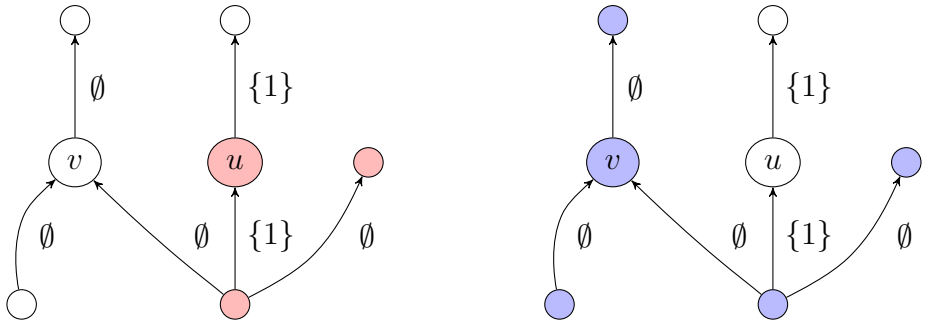
Corollary 3. *Only the descendants of an initial node x can see x.*

Proof. A special case of the above proposition where $v = x$ is initial. □

Corollary 4. *If all edges have the empty duration, then $u \trianglelefteq v$ iff* $\mathrm{base}(u) \supseteq \mathrm{base}(v)$.

Proof. Follows from corollary 3 and repeated application of corollary 2. □

All of the properties mentioned so far are illustrated in figure 5.3.



(a) The nodes visible from u.

(b) The nodes are visible from v.

Figure 5.3: Illustration of basic properties of causality graphs. The domination relation is defined so that the atom 1 does not dominate itself.

5.3.1 Transitivity

It is natural that seeing should be transitive. That is, if node u can see node v and node v can see node w, then node u should also be able to see node w. This is indeed true of our relation $\measuredangle$.

We start by introducing some auxiliary definitions, which ease the presentation of the proofs and provide a more natural language to speak in.

Definition 23 (matching)**.** The *matching* relation on atoms $\geq$ is the relfexive closure of $\gtrsim$. That is, $d_1 \geq d_2$ if either $d_1 \gtrsim d_2$ or $d_1 = d_2$. It is naturally lifted to composites and paths in the causal graph.

It is clear that if p dominates q, then it also matches q. The converse is not necessarily true.

Lemma 1. *The matching relation is reflexive and transitive. In addition to that, it composes with the domination relation as follows:*

1. *If $p \gtrsim q$ and $q \geq r$, then $p \gtrsim r$.*
2. *If $p \geq q$ and $q \gtrsim r$, then again $p \gtrsim r$.*

Proof. By definition, p matches q iff for every $d_1 \in \delta(q)$, there exists some $d_2 \in \delta(p)$ such that $d_2 \geq d_1$. From this, reflexivity is trivial, since for every element $d \in \delta(p)$, we have $d = d$ and thus $d \geq d$.

To prove transitivity, we need to show that for any $d_1 \in \delta(r)$, there is some $d_2 \in \delta(p)$ such that $d_2 \geq d_1$. Reason as follows: since $q \geq r$, there is some $d_{1.5} \in \delta(q)$ such that $d_{1.5} \geq d_1$; since $p \geq q$, there is some $d_2 \in \delta(p)$ such that $d_2 \geq d_{1.5}$. Chaining these two statements using transitivity of $\geq$ on atoms, we obtain the desired result.

The remaining two statements are proven analogously. □

The following lemma essentially states that splicing preserves the matching relation. This is a crucial technical tool for proving transitivity later in the section.

Lemma 2. *Let p be any path, and let $u \in V(p)$. If path r satisfies $r \geq p[u\ldots]$, then splicing r into p results in a path that matches the original path p, i.e. $p[\ldots/r] \geq p$. Similarly, if $r \leq p[u\ldots]$, then $p[\ldots/r] \leq p$.*

Proof. The property $r \geq p[u\ldots]$ of paths translates to $\delta(r) \geq \delta(p[u\ldots])$ on durations. Let us now take a look at the durations of $p[\ldots/r]$ and p:

$$\delta(p[\ldots/r]) = \delta(p[\ldots u]) \cup \delta(r)$$
$$\delta(p) = \delta(p[\ldots u]) \cup \delta(p[u\ldots])$$

For every element of $\delta(p[u\ldots])$, there is a matching (and in fact dominating) element of $\delta(r)$. For every element of $\delta(p[\ldots u])$, there is trivially a matching element of $\delta(p[\ldots u])$. The desired result follows.

The case where $r \leq p[u\ldots]$ is analogous. □

Corollary 5. *If q dominates some tail of p, then $p[\ldots/q] \geq p$. Similarly, if q is dominated by some tail of p, then $p[\ldots/q] \leq p$.*

We are now ready to tackle the actual problem.

Theorem 2. *The relation $\measuredangle$ is transitive.*

Proof. Suppose that $u \measuredangle v$ and $v \measuredangle w$. Let us take any grounded path $r \in \mathcal{P}_1(w)$; we construct some $p \in \mathcal{P}_1(u)$ such that p tail-dominates r.

Since $v \measuredangle w$, there exists some path $q_1 \in \mathcal{P}_1(v)$ that tail-dominates r. Consider the path $q = r[\ldots/q_1]$; it is grounded, starts in the same node as r, and leads to v. Since $u \measuredangle v$, there exists some path $p \in \mathcal{P}_1(u)$ that tail-dominates q. Consider cases based on the relative order of nodes $\mathrm{start}(p)$ and $\mathrm{start}(q_1)$ on the path q.

If $\mathrm{start}(p)$ is before $\mathrm{start}(q_1)$, splicing p into q cuts out the entire subpath q_1. From the definition of q_1, we know that $q_1 \gtrsim r[\mathrm{start}(q_1)\ldots]$; by splicing the latter into $q[\mathrm{start}(p)\ldots]$ and using lemma 2, we obtain

$$q[\mathrm{start}(p)\ldots] \quad \geq \quad q[\mathrm{start}(p)\ldots][\ldots/r[\mathrm{start}(q_1)\ldots]] \quad = \quad r[\mathrm{start}(p)\ldots].$$

From the definition of p, we know that $p \gtrsim q[\mathrm{start}(p)\ldots]$. Using lemma 1, we obtain $p \gtrsim r[\mathrm{start}(p)\ldots]$, thus we have constructed some p that tail-dominates r, as desired.

The case where $\mathrm{start}(p)$ is after $\mathrm{start}(q_1)$ is proven analogously, except that the other halves of the lemmas 1 and 2 are used. □

5.4 Computational aspects

Let us now turn to the computational problem of determining which pairs of nodes see each other, and its simplifications.

Definition 24 (simple causal graph)**.** A causal graph G is *simple* if each edge has a duration of size at most 1. In other words, the allowed edge durations are the empty set and the singleton sets.

We assume that the causal graph considered is a simple one. This is merely a technical detail, because any causal graph can be transformed into a simple causal graph as follows. Each edge with a composite duration of size $n > 0$ is broken down into a chain of n edges, each of which has a singleton duration. The resulting causal graph clearly has the same visibility properties, and is at most $k = |\mathcal{D}_0|$ times as large.

With this simplification in mind, we notationally identify the singleton durations with atoms, i.e. write d instead of $\{d\}$. The empty duration is still denoted $\emptyset$ (to emphasize that it does *not* correspond to an atom).

5.4.1 Computational model

Set operations. Assume the following time complexities for operations on *immutable* sets, with the size of the set denoted n:

1. Constructing an empty set in $O(1)$.
2. Testing for membership in $O(1)$.
3. Returning a new set with an added/removed element in $O(n)$.
4. Iterating over all elements of the set in $O(1)$ per element.

In the case of mutable sets, we assume the same complexities except for the addition/deletion of elements, where we assume complexity $O(1)$.

Causality graph representation. The causality graph is *not* given as a list of vertices and edges; instead, we assume an implicit representation (e.g. in memory, in a database, ...). The graph is lazily generated, starting from the initial nodes. Thus, given an arbitrary node u, it is guaranteed that all its ancestors are already computed — the complete set of its parents can be accessed in $O(1)$ time. The (not necessarily complete) set of its children can also be accessed in $O(1)$ time. However, there are no time guarantees as to when this set becomes complete.

5.4.2 Problems

Problem 1 ($\lambda q \lambda p$)**.** Given is a path p leading to u and a path q leading to v. Determine whether p tail-dominates q.

Theorem 3. *The $\lambda q \lambda p$ problem can be solved in time $O(|p| \cdot |q|)$.*

Proof. If $\mathrm{start}(p) \notin V(q)$, then p cannot tail-dominate q. This can be checked in time $O(|q|)$.

Otherwise, since the graph is acyclic, the node $\mathrm{start}(p)$ is exactly once on the path q. Thus there is only one tail $q[\mathrm{start}(p)\ldots]$ of q that can be dominated by p. It suffices to check that for each $d_1 \in \delta(q[\mathrm{start}(p)\ldots])$, there is some $d_2 \in \delta(p)$ such that $d_2 \gtrsim d_1$. This can be done in time $O(|p| \cdot |q|)$.

The total running time of the above procedure is $O(|q| + |p| \cdot |q|) = O(|p| \cdot |q|)$. □

Problem 2 ($\lambda q \exists p$)**.** Given is a node u and a path q leading to v. Determine whether there is a path $p \in \mathcal{P}_1(u)$ that tail-dominates q.

Theorem 4. *The $\lambda q \exists p$ problem is* NP*-complete.*

Proof. It is NP, since given a path $p \in \mathcal{P}_1(u)$, it is easy to check in polynomial time whether it tail-dominates q (the $\lambda q \lambda p$ problem).

Thus it remains to show NP-hardness; we do so by reduction of the SAT problem to the $\lambda q \exists p$ problem. Given a CNF formula $\phi = k_1 \wedge \ldots \wedge k_n$, where k_i are the individual clauses, we construct an instance of the $\lambda q \exists p$ problem such that the answer is "yes" iff ϕ is satisfiable.

The idea is that paths to u should correspond to all possible boolean assignments to variables, and the labels on each path should correspond to those clauses that are satisfied by the assignment. The path q and the domination relation $\gtrsim$ are set so that p tail-dominates q iff the labels on p are precisely $k_1, \ldots, k_n$ (in no particular order).

The exact construction follows. Let $\mathcal{D}_0 = \{k_1, \ldots, k_n\}$ and let the domination relation be so that the only atom d such that $d \gtrsim k_i$ is $d = k_i$ itself. Let the path q be

$$v_0 \xrightarrow{k_1} v_1 \xrightarrow{k_2} v_2 \xrightarrow{k_3} \ldots \xrightarrow{k_n} v_n = v.$$

We now construct the subgraph before u. Let $x_1, \ldots, x_m$ be all the variables in ϕ. For each variable x_i, we have a node u_i. Exactly two paths from u_i to u_{i+1} are constructed: the paths correspond to the assignments $x_i = 0$ and $x_i = 1$, and are labelled by those clauses that are satisfied by the assignment. After the last assignment, we go to $u_{m+1} = u$. Finally, we set $u_1 = v_0$, so that the paths to u and the path q have a common starting node (this being the *only* common node). □

The reduction is illustrated in figure 5.4.

Problem 3 ($\forall q \exists p$)**.** Given are nodes u and v. Determine whether $u \measuredangle v$, that is, whether for each grounded path $q \in \mathcal{P}_1(v)$ there is a path $p \in \mathcal{P}_1(u)$ that tail-dominates q.

Hypothesis 1. *The $\forall q \exists p$ problem is Π_2^{P} complete.*

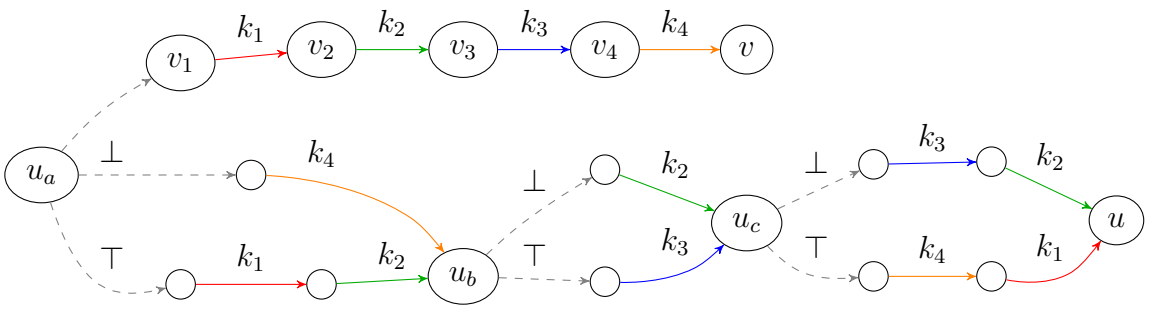


Figure 5.4: Reduction of a boolean SAT instance to an instance of the $\lambda q \exists p$ problem. The SAT instance is $\phi = (a \vee c) \wedge (a \vee \neg b \vee \neg c) \wedge (b \vee \neg c) \wedge (\neg a \vee c)$, where the clauses are denoted $k_1, \ldots, k_4$ from left to right. The dashed gray edges are for illustration only; the actual reduction is obtained by contracting these edges.

It is easy to see that the problem is Π_2^{P}, since given a grounded path q, the problem of deciding whether there is a satisfying path p or not is NP (theorem 4).

The difficulty is in the other direction. One may try a construction similar to the one in the proof of theorem 4: paths to u determine the existential variables and paths to v determine the universally quantified variables. This, however, does not map naturally to the $\forall\exists$-SAT problem; we illustrate the difficulty below.

We can view both problems as games with two players: first, the *universal* player chooses the universally quantified variables, and then the *existential* player chooses the existentially quantified ones. The outcome of the game is determined by the truth value of ϕ under the resulting boolean assignment: the existential player wins if it is true, otherwise the universal player wins.

In $\forall\exists$-SAT, each assignment to a universal variable eliminates a clause, in the following sense: if the assignment satisfies a clause, the existential player no longer needs to pay attention to it. Thus the universal player is trying to create as hard of a SAT instance as possible *by avoiding* the elimination of crucial clauses.

On the other hand, in the $\forall q \exists p$ problem, the universal player's choices directly create the problem instance for the existential player. Instead of avoiding the elimination of crucial clauses, the player chooses the crucial clauses directly.

For now, we leave the problem of Π_2^{P} completeness of the $\forall q \exists p$ problem open. However, it is clearly at least as hard as the $\lambda q \exists p$ problem. Thus if we are to base our programming language on the notion of seeing, we either have to consider some restricted version of the problem, or we have to show that it is tractable in common cases (i.e. fixed-parameter tractable and the parameter is usually small).

5.4.3 The $\forall q \exists p$ problem

Theorem 5. *The $\forall q \exists p$ problem can be solved in time*

$$O(k \cdot 2^k \cdot m_1 + k^2 \cdot 2^{2k} \cdot m_2),$$

where $k = |\mathcal{D}_0|$ and $m_1 = |\text{before}(u)|$, $m_2 = |\text{before}(v)|$ are the sizes of the subgraphs before u, v, respectively.

Definition 25 (remaining paths)**.** Given nodes u, v and node x that is an ancestor of v, the set $\text{rems}(u, v, x)$ is the set of all paths from x to v that are not tail-dominated by any path leading to u.

The above definition is crucial. First, it characterizes visibility: $u \measuredangle v$ iff for each $x \in \text{base}(v)$, the set of remaining paths $\text{rems}(u, v, x)$ is empty. Second, the values $\text{rems}(u, v, x)$ for all ancestors x of v can be computed by dynamic programming, using the following lemma.

Lemma 3. *If $x = v$, then*

$$\text{rems}(u, v, x) = \begin{cases} \{\varepsilon(v)\} & \text{if } v \text{ is not an ancestor of } u, \\ \emptyset & \text{otherwise.} \end{cases}$$

Otherwise, let $e_1 = x \to x_1, \ldots, e_n = x \to x_n$ be all the outgoing edges from x. Then,

$$\text{rems}(u, v, x) = \bigcup_{i=1,\ldots,n} \left\{ e_i \cdot q \;\middle|\; \begin{array}{c} q \in \text{rems}(u, v, x_i), \\ \forall p \in \mathcal{P}_{0,1}(x, u) : p \not\gtrsim e_i \cdot q \end{array} \right\}.$$

In other words, the lemma states that each path in $\text{rems}(u, v, x)$ can be constructed from some path in $\text{rems}(u, v, x_i)$ by prepending it with e_i. However, we must leave out those paths q for which the resulting $e_i \cdot q$ would be dominated by some path p from x to u.

Note that if we really want to calculate the values $\text{rems}(u, v, x)$, we must also know the values $\mathcal{P}_{0,1}(x, u)$. These too can be calculated using a dynamic programming approach: if $x = u$, then the only path is the empty path; otherwise, each path can be constructed by taking one step along an outgoing edge $x \xrightarrow{D_i} x_i$, and constructing a path from x_i to u.

Another concern is the sizes of the sets $\text{rems}(u, v, \_)$ and $\mathcal{P}_{0,1}(\_, u)$, since there can be potentially many paths. However, we do not need to know the entire information about a path. The only operations done on paths are:

1. constructing an empty path ($\varepsilon(v)$),
2. prepending an edge ($e_i \cdot q$),
3. checking whether one path is dominated by another path ($p \not\gtrsim e_i \cdot q$).

Instead of paths, any data structure $\mathcal{DS}$ supporting these operations suffices. The advantage is that the number of possible instances of the data structure can be much smaller than the number of paths.

The properties of the data structure can be formally stated as follows: there must be a mapping $f : \mathcal{P} \to \mathcal{DS}$ from paths to instances of the data structure, together with functions $\mathsf{empty} : V \to \mathcal{DS}$, $\mathsf{prepend} : E \to \mathcal{DS} \to \mathcal{DS}$ and $\mathsf{dominates} : \mathcal{DS} \to \mathcal{DS} \to \{0, 1\}$ satisfying the following conditions:

$$\begin{aligned} \mathsf{empty} &= f \circ \varepsilon \\ \mathsf{prepend}(e, f(p)) &= f(e \cdot p) \\ \mathsf{dominates}(f(p), f(q)) &= \begin{cases} 1 & \text{if } p \gtrsim q, \\ 0 & \text{otherwise.} \end{cases} \end{aligned}$$

In particular, we can choose $f = \delta$. This corresponds to considering only the durations of a path (instead of the entire path information).

1. The empty path has no labels on it, thus $\delta(\varepsilon(v)) = \emptyset$.
2. Prepending an edge merely adds the edge's duration: $\delta(e \cdot q) = \delta(e) \cup \delta(q)$.
3. Domination on paths translates (by definition) to domination on their durations: $p \gtrsim q \iff \delta(p) \gtrsim \delta(q)$.

In the most general setting, the algorithms for calculating $f(\mathrm{rems}(u, v, \_))$ and $f(\mathcal{P}_{0,1}(\_, u))$ are described in figures 1 and 2.

Time complexity. Let us now analyze the time complexity of these algorithms. Let t_0, t_{+1} and $t_\gtrsim$ be the time complexities of the functions empty, $\mathsf{prepend}$ and $\mathsf{dominates}$, respectively.

Before we run the actual algorithm, we do some precomputation. First, in order to ensure that $\mathsf{isAncestor}(\_, u)$ and $\mathsf{isAncestor}(\_, v)$ values can be retrieved in $O(1)$ time, we traverse through all ancestors of u (resp. v) and store them in a lookup table. Next, note that the only part of the graph relevant to fPaths are the ancestors of u. Thus, in a similar backwards traversal, we create a representation of the part of the graph before u. All queries posed during fPaths are posed to this representation instead of the actual graph. The benefit of this is that now, we may assume that the graph has size $O(m_1)$ during fPaths. Similarly for fRemPaths, the relevant part of the graph has size $O(m_2)$.

With these assumptions, let us analyze fPaths. There is one call to empty. For each of the $O(m_1)$ edges e in the graph, we consider up to $|\mathcal{DS}|$ values D; for each of them, we do one $\mathsf{prepend}$ operation. Thus this part of the algorithm takes time

$$O(t_0 + m_1 \cdot |\mathcal{DS}| \cdot t_{+1}).$$

```
1  Function fRemPaths(u, v, x)
2      if ¬isAncestor(x, v) then
3          return ∅
4      if x = v then
5          return (if isAncestor(v, u) then ∅ else {empty(v)})
6      else
7          result := MutableSet()
8          foreach e := x → y do
9              foreach D_1 ∈ fRemPaths(u, v, y) do
10                 D'_1 := prepend(e, D_1)
11                 dominated := 0
12                 foreach D_2 ∈ fPaths(x, u) do
13                     if dominates(D_2, D'_1) then
14                         dominated := 1
15                         break
16                 if ¬dominated then
17                     result.add(D'_1)
18         return freeze (result)
```

Algorithm 1: Algorithm for calculating values of $f(\text{rems}(u, v, \_))$. The entire function is implicitly memoized to avoid calculating the same value more than once.

```
1  Function fPaths(x, u)
2      if ¬isAncestor(x, u) then
3          return ∅
4      if x = u then
5          return {empty(u)}
6      else
7          result := MutableSet()
8          foreach e := x → y do
9              foreach D ∈ fPaths(y, u) do
10                 result.add(prepend(e, D))
11         return freeze (result)
```

Algorithm 2: Algorithm for calculating values of $f(\mathcal{P}_{0,1}(\_, u))$. The entire function is implicitly memoized.

The rest of the algorithm (i.e. calls to isAncestor, operations on result, ...) can be shown to take less time. Thus this is the time complexity of calculating all values of fPaths.

Assuming the values fPaths can be retrieved in $O(1)$ time (i.e. are already precomputed), fRemPaths can be analyzed as follows. There is one call to empty. For each of the $O(m_2)$ edges e in the graph, we consider up to $|\mathcal{DS}|$ values D_1; for each of them, we do one prepend operation and then consider up to $|\mathcal{DS}|$ values D_2; then, we compare them in time $t_{\gtrsim}$. Thus this part of the algorithm takes time

$$O(t_0 + m_2 \cdot (|\mathcal{DS}| \cdot t_{+1} + |\mathcal{DS}|^2 \cdot t_{\gtrsim})).$$

The rest of the algorithm (i.e. calls to isAncestor, updating the result variable, ...) can be shown to take less time. Thus this is the time complexity of calculating all values of fRemPaths (assuming fPaths is known).

Finally, to determine whether $u \measuredangle v$, we check whether for each $x \in \text{base}(v)$ the set $\text{fRemPaths}(u, v, x)$ is empty. Thus we obtain the following theorem.

Theorem 6. *The $\forall q \exists p$ problem can be solved in time*

$$O(t_0 + m_1 \cdot |\mathcal{DS}| \cdot t_{+1} + m_2 \cdot |\mathcal{DS}|^2 \cdot t_{\gtrsim}),$$

where $\mathcal{DS}$ is the data structure supporting the operations empty*,* prepend*,* dominates *with time complexities t_0, t_{+1} and $t_{\gtrsim}$, respectively.*

Consider the specialization $f = \delta$. The instances of the data structure are precisely the durations $\mathcal{D}$, and there are 2^k of them. The operations empty, prepend and dominates treat each duration as a (frozen) set (of atoms), and thus operate in times $t_0 = O(1)$, $t_{+1} = O(k)$ and $t_{\gtrsim} = O(k^2)$, respectively. This yields the time complexity

$$O(m_1 \cdot 2^k \cdot k + m_2 \cdot 2^{2k} \cdot k^2),$$

concluding the proof of theorem 5. □

Corollary 6. *The $\forall q \exists p$ problem is fixed-parameter tractable, with the parameter being the number of atoms k.*

Optimizations. A closer analysis of the algorithm shows that it is unnecessary to consider all durations. For one, if a duration contains atoms d_1, d_2 such that $d_1 \gtrsim d_2$, then the latter can be dropped. Thus it suffices to consider only those durations where each atom is maximal within that duration, i.e. it is not dominated by any other atom there (it can be dominated by itself, though). This allows us to obtain a better bound for certain domination relations $\gtrsim$.

Proposition 3. *If any two distinct atoms are comparable (i.e. either $d_1 \gtrsim d_2$ or $d_1 \lesssim d_2$), then the $\forall q \exists p$ problem can be solved in time $O(m_1 + m_2)$.*

Proof. Let $\mathcal{DS}$ be the set of maximal-only durations. Since any two atoms are comparable, any duration has a maximal element. Therefore, a maximal-only duration can contain at most one element. This allows for a faster prepend and dominates operations: $t_{+1} = O(1)$ and $t_{\gtrsim} = O(1)$. Moreover, there are only $O(k)$ sets of size at most 1, thus $|\mathcal{DS}| = O(k)$. Plugging into theorem 6, we obtain the bound

$$O(m_1 \cdot k + m_2 \cdot k^2).$$

Where do the coefficients k and k^2 come from? The fPaths and fRemPaths values are sets of durations, thus they have size $O(k)$. All of these values are considered at various places: the algorithm 1 considers them values on lines 9 and 12, and the algorithm 2 considers these values on line 9.

Note however, that not only are any two atoms comparable, but so are any two maximal-only durations (since these are exactly the empty set and the singleton sets). Thus, when calculating fPaths and fRemPaths values (which are sets of maximal-only durations), we can discard all durations except for the maximal one. Then, only $O(1)$ values are considered instead of $O(k)$ values. The desired bound follows. □

We can generalize the previous argument: instead of considering all possible durations, consider only maximal ones. This can be implemented by adjusting the algorithms 2 and 1 as follows: whenever the result variable is updated by adding a new duration D, we compare it to all other durations D' already present in result. Each D' that is dominated by D is discarded. On the other hand, if there is any D' that dominates D, we discard D.

The effect is smaller sets of durations, but at the cost of more costly updates to the variable result: insertion takes time $O(|\mathsf{result}|)$ instead of $O(1)$. By upper-bounding $|\mathsf{result}|$, we obtain the following theorem.

Theorem 7. *The $\forall q \exists p$ problem can be solved in time*

$$O(t_0 + m_1 \cdot (N \cdot t_{+1} + N^2 \cdot t_{\gtrsim}) + m_2 \cdot N^2 \cdot t_{\gtrsim}),$$

where N is the maximal number of elements of $\mathcal{DS}$ that are incomparable by dominates*, i.e. the size of the largest antichain. (Elements that dominate themselves are allowed, i.e. it is only required that any two* distinct *elements are incomparable.)*

5.4.4 The perspective problem

From a logic programming perspective, being able to check $u \measuredangle v$ for fixed nodes u, v is not enough. We are more interested in enumerating all nodes v that are visible from a given node u.

Problem 4 (perpective)**.** Given node u, find all vertices v such that $u \measuredangle v$.

The perspective problem is at least as hard as the $\forall q \exists p$ problem, since the latter (i.e. checking $u \measuredangle v$) can be solved by finding all nodes visible from u and then checking whether v is among them.

Computational assumptions. When designing algorithms for this problem, we have to take into consideration the lazy nature of the computational model: given a node u, only a not-necessarily-complete set of its children can be retrieved. To deal with this, we assume that whenever we ask for the perspective of the node u, all nodes visible from u are already computed, and thus can be reached in the graph with no issues.

Theorem 8. *In the restricted setting where $\mathcal{D}_0 = \{1\}$ and $\gtrsim\ = \emptyset$, the perspective problem can be solved in time $O(m)$, where $m = \text{halfBefore}(\text{prp}(u))$ is number of nodes and edges at least partially visible from u.*

Proof. Any node v before u is visible from u (proposition 1). Consider now node v that is not an ancestor of u, and let $x_1 \xrightarrow{D_1} v, \ldots, x_n \xrightarrow{D_n} v$ be all of v's incoming edges. Since atom 1 is not dominated by any atom, if it were among the labels D_i, it would not be the case that $u \measuredangle v$. On the other hand, if none of the labels is 1, then they must all be $\emptyset$. In this case, we can characterize visibility using corollary 2: $u \measuredangle v$ iff u sees all of v's parents. This yields the following forward chaining algorithm.

1. Start with $u \measuredangle v$ for all ancestors v of u.
2. If $u \measuredangle x_1, \ldots, u \measuredangle x_n$ and the nodes $x_1, \ldots, x_n$ have a common child v such that each of the edges $x_i \to v$ has the empty duration, then $u \measuredangle v$.

The step 2 is iterated until no new facts can be derived. This can be implemented with running time $O(m)$. □

We now turn to the perspective problem in general.

Definition 26 (ground-path covers)**.** Given a node u, a *u-ground-path cover* (u-gp-cover for short) of a ground path q is the set of all paths to u that tail-dominate q. It is denoted $\text{cover}_{(\mathcal{GP})}(u, q)$.

Definition 27 (node covers)**.** Given a node u, a *u-node cover* (u-n-cover for short) of a node v is the set of all u-gp-covers of ground paths to v. It is denoted $\text{cover}_{(V)}(u, v)$. Symbolically,

$$\text{cover}_{(V)}(u, v) = \{\text{cover}_{(\mathcal{GP})}(u, q) \mid q \text{ is a ground path to } v\}.$$

The above two definitions are crucial. First, vision can be characterized through n-covers as follows:

$$u \measuredangle v \qquad \text{iff} \qquad \emptyset \notin \text{cover}_{(V)}(u, v).$$

Second, both gp-covers and n-covers can be computed using dynamic programming. Using these two insights, we devise an inefficient algorithm for calculating the perspective of u. It is then made more efficient by using an appropriate data structure for representing relevant path information, in a similar way as in the $\forall q \exists p$ problem.

Definition 28 (path minus)**.** The operator $\ominus : 2^{\mathcal{P}} \to E \to 2^{\mathcal{P}}$ is defined as follows:

$$P \ominus e = \{p \mid p \in P,\ \delta(p) \gtrsim \delta(e)\}.$$

In other words, the operation $P \ominus e$ considers all paths in P and keeps only those whose duration dominates the duration of the edge e.

Lemma 4 (gp-cover extension)**.** *Let q be a ground path. If it is the empty path on node y, then*

$$\mathrm{cover}_{(\mathcal{GP})}(u, q) = \mathcal{P}_{0,1}(y, u).$$

Otherwise, let $q = q' \cdot e$, where $e = x \xrightarrow{D} y$. Then,

$$\mathrm{cover}_{(\mathcal{GP})}(u, q) = (\mathrm{cover}_{(\mathcal{GP})}(u, q') \ominus e) \cup \mathcal{P}_{0,1}(y, u).$$

Proof. By definition of gp-covers, we are looking for all paths p to u that tail-dominate the ground path q. Such a path p must start somewhere on q; consider cases based on this starting point.

If it starts in $\mathrm{end}(q) = y$, then it trivially tail-dominates q. Such paths p are precisely the paths in $\mathcal{P}_{0,1}(y, u)$.

Otherwise, the path $q = q' \cdot e$ must be non-empty, and p starts in the prefix q'. Such a path must necessarily tail-dominate the shorter ground path q', but it must also dominate the edge e used to extend it. This is precisely what is achieved with the $\ominus$ operation. □

Definition 29 (path-set minus)**.** The operator $\hat{\ominus} : 2^{2^{\mathcal{P}}} \to E \to 2^{2^{\mathcal{P}}}$ is defined as follows:

$$\hat{P} \,\hat{\ominus}\, e = \{P \ominus e \mid P \in \hat{P}\}.$$

If we interpret the set $\hat{P}$ as an n-cover of some node v, then the operation $\hat{P} \,\hat{\ominus}\, e$ calculates the n-cover of a hypothetical node v' with a single incoming edge $e' = v \xrightarrow{\delta(e)} v'$ and no outgoing edges.

This is because ground paths to v' are precisely the ground paths to v extended with e'. To obtain the set of gp-covers, we just need to extend each gp-cover of v with the edge e'. Since there are no outgoing edges from v', we have $\mathcal{P}_{0,1}(v', u) = \emptyset$.

Lemma 5 (n-cover extension)**.** *Let v be a node. If it is initial, then*

$$\mathrm{cover}_{(V)}(u, v) = \{\mathcal{P}_{0,1}(v, u)\}.$$

Otherwise, let $e_1 = x_1 \to v, \ldots, e_n = x_n \to v$ *be all the incoming edges to* v*. Then,*

$$\mathrm{cover}_{(V)}(u, v) = \bigcup_{i=1,\ldots,n} \{(P \ominus e_i) \cup \mathcal{P}_{0,1}(v, u) \mid P \in \mathrm{cover}_{(V)}(u, x_i)\}.$$

Proof. The case when v is initial is trivial: the only ground path is the empty path $\varepsilon(v)$, and the corresponding gp-cover is $\mathcal{P}_{0,1}(v, u)$.

Consider the second case. Ground paths to v are precisely the ground paths to some x_i extended with the edge e_i. Thus to obtain the set of gp-covers of v, it suffices to get the gp-covers of each x_i and extend each of them with e_i, respectively. □

The previous lemma yields an algorithm for calculating values $\mathrm{cover}_{(V)}(u, \_)$. Then, nodes v visible from u can be found using a simple forward chaining algorithm:

1. Start by marking all ancestors of u as visible.
2. Whenever all parents of a node x are marked visible, it is possible that x is visible as well. To find out, calculate $\mathrm{cover}_{(V)}(u, x)$ and check whether it contains the empty set. If yes, mark x as visible.

Step 2 is iterated until no more nodes can be marked.

All of this can be made more efficient by considering only relevant path information. This is done by using a data structure $\mathcal{DS}$ corresponding to a mapping $f : \mathcal{P} \to \mathcal{DS}$, with standard operations empty, prepend and dominates. That is, we calculate values

$$\mathrm{fNodeCover}(u, x) = \{\{f(p) \mid p \in P\} \mid P \in \mathrm{cover}_{(V)}(u, x)\}$$

instead of values $\mathrm{cover}_{(V)}(u, x)$. However, in the following time complexity analysis and optimizations, we still refer to data structure instances as paths, for clarity and ease of presentation.

The algorithm for computing fNodeCover values is described in figure 3.

Time complexity. Let $t_{\gtrsim e}$ be the cost of one operation of comparing a path with an edge. A naive implementation (line 11) yields $t_{\gtrsim e} = t_0 + t_{+1} + t_{\gtrsim}$, where t_0, t_{+1} and $t_{\gtrsim}$ are the costs of the operations empty, prepend and dominates, respectively.

We assume that all values of $\mathrm{fPaths}(\_, u)$ have been precomputed and can be retrieved in $O(1)$ time. Since only fPaths for ancestors of u are computed, the precomputation takes time

$$O(t_0 + |\mathrm{before}(u)| \cdot |\mathcal{DS}| \cdot t_{+1}).$$

First, let us consider the size of the relevant subgraph, i.e. the number of nodes and edges considered by the forward chaining algorithm. A node is considered either if it is visible from u, or it is adjacent to a visible node. Similarly, an edge is considered if it is incident to a visible node. Thus the relevant subgraph is precisely $\mathrm{halfBefore}(\mathrm{prp}(u))$.

```
1  Function fNodeCover(u, v)
2      fPathsVU := fPaths(v, u)
3      if isInitial(v) then
4          return {fPathsVU}
5      else
6          result := MutableSet()
7          foreach e := x → v do
8              foreach D̂ ∈ fNodeCover(u, x) do
9                  D̂′ := MutableSet()
10                 foreach D ∈ D̂ do
11                     if dominates(D, prepend(e, empty)) then
12                         D̂′.add(D)
13                 foreach D ∈ fPathsVU do
14                     D̂′.add(D)
15                 result.add(freeze(D̂′))
16         return freeze(result)
```

Algorithm 3: Algorithm for calculating values of fNodeCover(u, _). The entire function is implicitly memoized.

Next, let us analyze the total cost of calls to fNodeCover. For each of the $O(m)$ edges $e := x \to y$, we consider the gp-covers $\hat{D} \in 2^{\mathcal{DS}}$; there are $O(2^{|\mathcal{DS}|})$ possible gp-covers. For each of those, we filter out those paths D that do not dominate the edge e; there are $O(|\mathcal{DS}|)$ distinct paths to consider. Each individual path D requires one t_0 and one t_{+1} operation (to create a path corresponding to the edge e), and one $t_{\gtrsim}$ operation (to compare with D). The rest can be shown to be insignificant, which yields the time complexity

$$O(m \cdot 2^{|\mathcal{DS}|} \cdot |\mathcal{DS}| \cdot t_{\gtrsim e}).$$

This is clearly more than the time spent by precomputing fPath values. In total, we obtain the following theorem.

Theorem 9. *The perspective problem can be solved in time*

$$O(m \cdot 2^{|\mathcal{DS}|} \cdot |\mathcal{DS}| \cdot t_{\gtrsim e}).$$

Corollary 7. *If $k = |\mathcal{DS}_0|$ is the number of distinct atoms, then the perspective problem can be solved in time*

$$O(m \cdot 2^{2^k} \cdot 2^k \cdot t_{\gtrsim e}).$$

Optimizations. *Gp-covers.* First, suppose that there are two paths D_1, D_2 in a gp-cover such that $\mathsf{dominates}(D_1, D_2)$. Then, whenever the latter path can be extended by an edge s, so can the former; thus p_2 need not be considered at all in the gp-cover. We can thus restrict the set of valid gp-covers to *maximal-only* ones, i.e. ones where each path p is maximal.

In the algorithm, this can be achieved by the following adjustment: whenever a new element D is to be added to the set $\hat{D}'$ in line 12, we discard those already present elements D' that are dominated by D. On the other hand, if any of those elements dominates D, then we discard D. This leads to smaller gp-covers, but at the cost of longer construction times. The time complexity is

$$O(m \cdot 2^N \cdot (N^2 + N \cdot t_{\gtrsim e})),$$

where N is the size of the greatest antichain on paths.

N-covers. Second, note that an n-cover is only as good as its "weakest" gp-cover. Suppose that sets of paths $\hat{D}_1, \hat{D}_2$ satisfy the following: for each path $D_1 \in \hat{D}_1$, there exists a path $D_2 \in \hat{D}_2$ such that $\mathsf{dominates}(D_2, D_1)$. Then, when we decide to extend boths sets $\hat{D}_1, \hat{D}_2$ with an edge e (in the sense of the operator $\ominus$), the property is preserved. But that means that in a series of such extensions, the empty set always appears first in the set $\hat{D}_1$. Thus if both $\hat{D}_1, \hat{D}_2$ were to appear in an n-cover, we can discard the latter, and it suffices to consider only minimal-only n-covers.

In the algorithm, this can be achieved by the following adjustment: whenever a new gp-cover $\hat{D}'$ is added to the result variable, we discard those already present gp-covers $\hat{D}$ that are at least as good as $\hat{D}'$ (in the sense described above). On the other hand, if any of these gp-covers is worse than $\hat{D}'$, we discard $\hat{D}'$. This leads to smaller n-covers, but at the cost of longer construction times. The time complexity is

$$O(m \cdot (M^2 + M \cdot (N^2 + N \cdot t_{\gtrsim e})),$$

where M is the size of the largest antichain on maximal-only sets of paths.

Theorem 10. *The perspective problem can be solved in time*

$$O(m \cdot (M^2 + M \cdot (N^2 + N \cdot t_{\gtrsim e})),$$

where N is the size of the largest antichain on paths, and M is the size of the largest antichain on maximal-only sets of paths.

Corollary 8. *In the restricted setting where any two distinct atoms are comparable, the perspective problem can be solved in time $O(m \cdot t_{\gtrsim e})$.*

Proof. Since any two distinct atoms are comparable, the size of the largest antichain on paths is 1, thus $N = 1$. But then, any two sets of maximal-only paths are comparable as well, thus $M = 1$ as well. □

5.5 Syntax and semantics

In previous sections, causality graphs were viewed as static objects. This was for the purposes of analyzing the seeing relation. But in the greater scheme of things, the graph represents the history of the game, which is a dynamic object: further and further events (i.e. objects) take place. The static properties of the graph are interesting only as aids to this graph generation process. We now take a closer look at this process.

Without constraints in rules. The game is given as a set of rules, where each rule has the form of a typed function. The meaning of a function $f : A \to B$ without constraints is that for every object a of type A, the object $f\,a$ of type B should be constructed; naturally, the incoming edges to this result are precisely from f and a. What should be the durations of these edges? At first glance, any of the following seems reasonable:

1. The arrow is labelled with the duration δ of the edge from a to $f\,a$. That is, we would write $f : A \xrightarrow{\delta} B$. The edge from f is of fixed duration δ_F.
2. Both of the durations can be parameterized, e.g. $f : A \xrightarrow{\delta_f,\ \delta_a} B$.

The option 2 is more general. However, it is more verbose, and there should be some reasonable default which suffices for most cases. This is exactly option 1, the default being δ_F. What should be the default value?

Consider the case when the function takes multiple arguments in succession, i.e. $f : A_1 \xrightarrow{\delta_1} \ldots \xrightarrow{\delta_n} A_n \to B$, and suppose there are objects $a_1, \ldots, a_n$ of the respective argument types. Then, the result is $r = f\,a_1\,\ldots\,a_n$; let us look at the edges between this result and the arguments a_i (figure 5.5). The path from f to r consists of n *application edges*, and each of the arguments connects to this path via one *argument edge*. Note that the last argument is special, in that its path to the result r does not contain any application edges. Thus, if all the arguments are to be treated the same (which is intuitive), then the only reasonable option for the default δ_F is the empty duration $\emptyset$. Another reason for why $\emptyset$ should be the default duration is that it is the universal duration.

The rules of a game form a whole, i.e. there is some context ("the game") unifying all of the rules. Therefore there should be some special node g_0 that immediately produces all of the rules (i.e. there is an edge with the empty duration to each of the rules). This ensures that the rules see each other. Which in turn makes it impossible for some node a to not see some b merely because the latter used a rule which the former did not.

Constraints. Visibility allows us to consider the perspectives of the individual objects. Each perspective is a closed world, thus it makes sense to use the language of logic

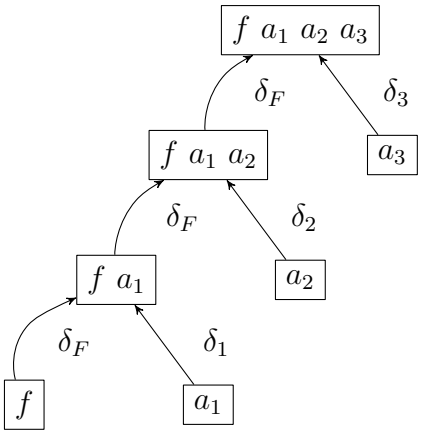


Figure 5.5: Function application in the causality graph.

for describing constraints from this perspective. For example: "From the perspective of x, there must be some green object, and all green objects must be fruits."

There may be some rules such that: they place constraints on the perspective of node x, but they also produce objects that can be seen by x. If the logic is monotonous, this is not an issue; however, the presence of negation complicates things. A newly produced object may invalidate past (negative) assumptions. Thus, either some form of stratification is required to avoid such scenarios, or some form of stable models is required to invalidate those histories that undermine themselves.

5.6 Suitability for board games

Finally, let us see the suitability of the language for representing board games. In the second part, we have analyzed board games and identified common patterns. Let us go over them one by one, and see if they can be expressed in our language:

Declarative statements. Yes, with the expressivity depending on the logic for describing constraints.

Game history. Yes.

Hypothetical reasoning. Depends on the logic for describing constraints. However, stable model semantics for negation are desirable as well, and combining the two is non-trivial.

Exceptions. The conditional activation can be handled, but the override requires some notion of mutation. This would have to be user defined (and likely computationally inefficient).

Imperative statements. Yes (simply use a non-empty duration).

Triggers. Yes (same as declarative statements).

Replacement effects. Same as exceptions.

The language treats declarative statements as the same thing as imperative statements; the only difference is in the duration used. Roughly, declarative statements use the empty duration whereas imperative statements use a non-empty duration. However, many levels of operation can be defined with durations.

5.7 Future work

In this section, we sketch some directions for future work.

Alternative construction of durations. We have used a powerset construction of the durations. This is flexible: new atomic durations can be added with arbitrary relations to other atomic durations, and the framework adapts. However, this generality comes at the cost of computational hardness. This could be solved by requiring the atomic durations to be totally ordered (except for reflexivity). However, this may introduce unintended side effects, as the user specified order of atomic durations would have to be extended to some total order, and it is not clear which one it should be.

Alternative constructions to the durations could be considered. Also, from a mathematical point of view, it is of interest to separate all the relevant properties of the construction into a more general abstract object. To illustrate, the powerset operations results in a structure similar to join-semilattices, except that not every element can see itself and there can be equivalent elements that are not equal; the abstract object (or the alternative constructions) could go in this direction.

Representation of state. The framework represents only the history of the game. From this point of view, the game is monotonous: objects can only appear, they cannot be deleted. Mutable objects do not exist at this level.

Instead, they exist as an *interpretation of the history*. For example, the deletion of an object x could manifest as a new object saying "x has been deleted". To actually see whether x is or is not there, one has to view all the past events to see whether there is a deletion event or not. Essentially, one *calculates the current state from the history*. In a naive implementation, this would take time at least linear in the length of the history, making it expensive.

Conclusion

In this thesis, we have made the first steps towards a programming language for board games. The foundation for our language is type theory, which we extend with time. However, if we are to found a programming language based on these concepts, then we must show that they are intuitive and that the relevant computational problems are tractable.

To reason about type theory with time, we have defined *causality graphs*, with the central notions being *durations* and the *domination relation*.

We then defined the *seeing* relation. The intended meaning of seeing is that a node (representing an object) sees precisely those other nodes which were constructed either before or at the same time. We have shown that our definition satisfies many intuitive properties; we specially mention reflexivity, transitivity, and the property that each node x can see all its ancestors (i.e. nodes that participated in the x's construction).

We then defined the computational problems related to seeing. The most general among them is the decision problem called the $\forall\exists$ problem. We have shown that this problem is NP-hard. We hypothesize that the problem is in fact Π_2^{P}-hard.

Afterwards, we devised a general algorithm for the $\forall\exists$ problem. An analysis of the running time revealed that it is in fact fixed-parameter tractable, with the parameter being the number of atomic durations. We then showed that when the domination relation is restricted to near total orders, the algorithm can be adjusted to run in linear time in the size of the graph.

We then moved on to the more general *perspective problem*, and devised an algorithm for it. Analysis of its running time again showed that the problem is fixed-parameter tractable, again with the parameter being the number of atomic durations. And again, for the restricted case of the domination relation being a near total order, there is an algorithm running in time linear in the size of the graph.

The above has resolved the questions about the intuitiveness and the computational tractability. So far, the view of the causality graph was static. But the actual graph represents the history of the game and is thus dynamic — the question of semantics of the language arises. This can be succinctly characterized as a type theory, with constraints for restricting applications of functions. Constraints are interpreted from the perspective of a certain node, and the language of logic is used to talk about this

perspective.

Finally, we have evaluated the language with respect to the board game patterns described in chapter 3. Specially, we mention that the language treats declarative statements the same as imperative statements, i.e. they are merely different faces of the same thing.

There are several directions for future work. One is devising alternative constructions for durations in the causal graph, which is interesting from a pure mathematical perspective (to isolate the important properties) but also from a programming language perspective (there could be a flexible construction that yields faster algorithms than the powerset construction). Another direction is the problem of state representation, where the goal is to extend the current "history only" framework with fast operations on "current states". Last but not least, the language could be implemented, which would also require defining a formal syntax for the language.

Bibliography

[1] En passant image. https://commons.wikimedia.org/wiki/File:Ajedrez_captura_al_paso_del_peon.png, 2020 (accessed August 1, 2020).

[2] System definition language. http://ggp.stanford.edu/notes/chapter_18.html, 2020 (accessed August 1, 2020).

[3] Toss. http://toss.sourceforge.net/, 2020 (accessed July 31, 2020).

[4] Francois Bancilhon and Raghu Ramakrishnan. An amateur's introduction to recursive query processing strategies. In *Readings in Artificial Intelligence and Databases*, pages 376–430. Elsevier, 1989.

[5] Henk P Barendregt. Introduction to generalized type systems. 1991.

[6] Catriel Beeri and Raghu Ramakrishnan. On the power of magic. *The journal of logic programming*, 10(3-4):255–299, 1991.

[7] Ana Bove, Peter Dybjer, and Ulf Norell. A brief overview of agda–a functional language with dependent types. In *International Conference on Theorem Proving in Higher Order Logics*, pages 73–78. Springer, 2009.

[8] Cameron Browne and Frederic Maire. Evolutionary game design. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(1):1–16, 2010.

[9] William F Dowling and Jean H Gallier. Linear-time algorithms for testing the satisfiability of propositional horn formulae. *The Journal of Logic Programming*, 1(3):267–284, 1984.

[10] Thomas Eiter, James Lu, and VS Subrahmanian. Computing non-ground representations of stable models. In *International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 198–217. Springer, 1997.

[11] Jose M Font, Tobias Mahlmann, Daniel Manrique, and Julian Togelius. A card game description language. In *European Conference on the Applications of Evolutionary Computation*, pages 254–263. Springer, 2013.

[12] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *ICLP/SLP*, volume 88, pages 1070–1080, 1988.

[13] Michael Genesereth, Nathaniel Love, and Barney Pell. General game playing: Overview of the aaai competition. *AI magazine*, 26(2):62, 2005.

[14] Herman Geuvers. Introduction to type theory. In *International LerNet ALFA Summer School on Language Engineering and Rigorous Software Development*, pages 1–56. Springer, 2008.

[15] Lukasz Kaiser and Lukasz Stafiniak. First-order logic with counting for general game playing. In *AAAI*, 2011.

[16] J. Kowalski and A. Kisielewicz. Game Description Language for Real-time Games. In *IJCAI Workshop on General Intelligence in Game-Playing Agents (GIGA'15)*, pages 23–30, 2015.

[17] J. Kowalski, M. Mika, J. Sutowicz, and M. Szykuła. Regular Boardgames. In *AAAI Conference on Artificial Intelligence*, 2019. (to appear).

[18] Claire Lefèvre, Christopher Béatrix, Igor Stéphan, and Laurent Garcia. Asperix, a first order forward chaining approach for answer set computing. *arXiv preprint arXiv:1503.07717*, 2015.

[19] Nathaniel Love, Timothy Hinrichs, David Haley, Eric Schkufza, and Michael Genesereth. General game playing: Game description language specification. 2008.

[20] Dale Miller and Gopalan Nadathur. *Programming with higher-order logic*. Cambridge University Press, 2012.

[21] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied logic*, 51(1-2):125–157, 1991.

[22] Luigi Palopoli. Testing logic programs for local stratification. *Theoretical Computer Science*, 103(2):205–234, 1992.

[23] Diego Perez-Liebana, Spyridon Samothrakis, Julian Togelius, Tom Schaul, Simon M Lucas, Adrien Couëtoux, Jerry Lee, Chong-U Lim, and Tommy Thompson. The 2014 general video game playing competition. *IEEE Transactions on Computational Intelligence and AI in Games*, 8(3):229–243, 2015.

[24] Teodor C Przymusinski. On the declarative semantics of deductive databases and logic programs. In *Foundations of deductive databases and logic programming*, pages 193–216. Elsevier, 1988.

[25] Zoltan Somogyi, Fergus J Henderson, and Thomas Charles Conway. Mercury, an efficient purely declarative logic programming language. *Australian Computer Science Communications*, 17:499–512, 1995.

[26] Michael Thielscher. A general game description language for incomplete information games. In *AAAI*, volume 10, pages 994–999, 2010.

[27] Michael Thielscher. The general game playing description language is universal. In *IJCAI Proceedings-International Joint Conference on Artificial Intelligence*, volume 22, page 1107, 2011.

[28] Markus Triska. The finite domain constraint solver of swi-prolog. In *International Symposium on Functional and Logic Programming*, pages 307–316. Springer, 2012.

[29] Maarten H Van Emden and Robert A Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM (JACM)*, 23(4):733–742, 1976.