ELSEVIER

Annals of Pure and Applied Logic 123 (2003) 1–99

ANNALS OF PURE AND APPLIED LOGIC

www.elsevier.com/locate/apal

Introduction to computability logic☆

Giorgi Japaridze

Department of Computing Sciences, Villanova University, 800 Lancaster Avenue, Villanova, PA 19085, USA

Received 1 July 2002; received in revised form 22 March 2003; accepted 22 April 2003
Communicated by S.N. Artemov

Abstract

This work is an attempt to lay foundations for a theory of interactive computation and bring logic and theory of computing closer together. It semantically introduces a logic of computability and sets a program for studying various aspects of that logic. The intuitive notion of (interactive) computational problems is formalized as a certain new, procedural-rule-free sort of games (called *static games*) between the machine and the environment, and computability is understood as existence of an interactive Turing machine that wins the game against any possible environment. The formalism used as a specification language for computational problems, called the *universal language*, is a non-disjoint union of the formalisms of classical, intuitionistic and linear logics, with logical operators interpreted as certain—most basic and natural—operations on problems. Validity of a formula is understood as being "always computable", and the set of all valid formulas is called the *universal logic*. The name "universal" is related to the potential of this logic to integrate, on the basis of one semantics, classical, intuitionistic and linear logics, with their seemingly unrelated or even antagonistic philosophies. In particular, the classical notion of truth turns out to be nothing but computability restricted to the formulas of the classical fragment of the universal language, which makes classical logic a natural syntactic fragment of the universal logic. The same appears to be the case for intuitionistic and linear logics (understood in a broad sense and not necessarily identified with the particular known axiomatic systems). Unlike classical logic, these two do not have a good concept of truth, and the notion of computability restricted to the corresponding two fragments of the universal language, based on the intuitions that it formalizes, can well qualify as "intuitionistic truth" and "linear-logic truth". The paper also provides illustrations of potential applications of the universal logic in knowledgebase, resourcebase and planning systems, as well as constructive applied theories. The author has tried to make this article easy to read. It is fully self-contained and can

☆Supported by NSF Grant CCR-0208816.
E-mail address: giorgi.japaridze@villanova.edu (G. Japaridze).
URL: http://www.csc.vill.edu/~japaridz

0168-0072/03/$ - see front matter © 2003 Elsevier B.V. All rights reserved.
doi:10.1016/S0168-0072(03)00023-X

be understood without any specialized knowledge of any particular subfield of logic or computer science.

© 2003 Elsevier B.V. All rights reserved.

MSC: primary 03B47; secondary 03F50; 03B70; 68Q10; 68T27; 68T30; 91A05

Keywords: Computability logic; Interactive computation; Game semantics; Linear logic; Intuitionistic logic; Knowledgebase

Contents

1. Introduction

Computability, i.e. algorithmic solvability, is certainly one of the most interesting concepts in mathematics and computer science, and it would be more than natural to ask the question about what logic it induces. The goal of this paper is to introduce a basic logic of computability and provide a solid formal ground for future, more advanced, studies of this exciting subject.

Computability is a property of *computational problems*, or *problems* for short, sometimes also referred to as *computational tasks*. They can be thought of as dialogues, or games between two agents. The most standard types of problems are short, two-step dialogues, consisting of asking a question (input) and giving an answer (output). However, there is no call for restricting the length of dialogues to only two steps: any more or less advanced and systematic study of computational problems sooner or later will face the necessity to consider problems of higher degrees or interactivity. After all, most of the tasks that real computers and computer networks perform are truly interactive! In this paper by a computational problem we always mean an interactive problem that can be an arbitrarily (including infinitely) long dialogue; moreover, as this will be seen shortly, we do not exclude "dialogues" of length less than 2, either.

The following few sections introduce a concept of games that presents a formalization of our intuitive notion of (interactive) computational problems, thus providing us with a clear formal basis for studying their algorithmic solvability. After defining a natural set of basic game operations and elaborating the corresponding logic, we can obtain a tool for deeper and more systematic understanding of computational problems and relations between them, such as, say, the relation of reducibility of one problem to another, or the property of decidability, both generalized to complex, interactive problems. What follows below in this section is an informal explanation of our approach.

We understand computational problems as games between two players: the *machine* (*agent*) and the *user* (*environment*). While the game-playing behavior of the latter can be arbitrary, the machine, as its name suggests, only can follow algorithmic strategies—strategies implementable as computer programs. As a simple example, consider the problem of computing the value of function f. In the formalism that we are going to employ, this problem would be expressed by the formula

$$\sqcap x \sqcup y(y = f(x)).$$

It stands for a two-move-deep game, where the first move—selecting a particular value m for x—must be made by the user, and the second move—selecting a value n for y—by the machine. The game is then considered won by the machine, i.e., the problem solved, if n really equals $f(m)$. Obviously, computability of f means nothing but existence of a machine that wins the game $\sqcap x \sqcup y(y = f(x))$ against any possible (behavior of the) user.

Generally, $\sqcap x A(x)$ is a game where the user has to make the first move by selecting a particular value m for x, after which the play continues—and the winner is determined—according to the rules of $A(m)$; if the user fails to make an initial move,

the game is considered won by the machine as there was no particular (sub)problem specified by its adversary that it failed to solve. $\sqcup xA(x)$ is defined in the same way, only here it is the machine who makes an initial move/choice. This interpretation makes $\sqcup$ a constructive version of existential quantifier, while $\sqcap$ is a constructive version of universal quantifier.

As for atomic formulas, such as $n = f(m)$, they can be understood as games without any moves. We call this sort of games *elementary*. An elementary game is automatically won or lost by the machine depending on whether the formula representing it is true or false (true = won, false = lost). This interpretation makes the classical concept of predicates a special case of games, and classical logic a special case of computability logic.

The meanings of the propositional counterparts $\sqcup$ and $\sqcap$ of $\sqcup$ and $\sqcap$ must be easy to guess. They, too, signify a choice by the corresponding player. The only difference is that while in the case of $\sqcup$ and $\sqcap$ the choice is made among the objects of the universe of discourse, $\sqcup$ and $\sqcap$ mean a choice between *left* and *right*. For example, the problem of deciding language L could be expressed by

$$\sqcap x(x \in L \sqcup x \notin L),$$

denoting the game where the user has to select a string s as a value of x, to which the machine should reply by one of the moves *left* or *right*; the game will be considered won by the machine if $s \in L$ and the move *left* was made or $s \notin L$ and the choice was *right*, so that decidability of L means nothing but existence of a machine that always wins the game $\sqcap x(x \in L \sqcup x \notin L)$.

By iterating available game operators one can express an infinite variety of computational problems, of an arbitrary degree of complexity/interactivity, only few of which may have special names established in the literature. For example, $\sqcap x(x \in L1 \sqcup x \in L2)$, "the problem of naming one of the two (not necessarily disjoint) sets $L1, L2$ to which a given object belongs", or the four-move-deep game $\sqcap x \sqcup y \sqcap z(P(x, y, z) \sqcup \neg P(x, y, z))$, "the problem of computing, for any given value m, a value n for which the (unary) predicate $P(m, n, z)$ will then be decided", etc.

In the above example we used classical negation $\neg$. The other classical operators will also be allowed in our language, and they all acquire a new, natural game interpretation. The reason why we can still call them "classical" is that, when applied to elementary games—the sort of games that are nothing but classical predicates—they act exactly in the classical way, and the non-standard behavior of these operators is only observed when their scope is extended beyond elementary games. Here is a brief informal explanation of how the "classical" operators are understood as game operations:

The game $\neg A$ is nothing but A with the roles of the two players switched: the machine's moves or wins become the user's moves or wins, and vice versa. For example, if *Chess* is the game of chess from the point of view of the white player, then $\neg$ *Chess* would be the same game from the point of view of the black player. (Here we rule out the possibility of draw outcomes in *Chess* by, say, understanding them as wins for the black player; of course, we also exclude the clock from our considerations.)

The operations $\wedge$ and $\vee$ combine games in a way that corresponds to the intuition of parallel computations. Playing $A \wedge B$ or $A \vee B$ means playing, in parallel, the two games A and B. In $A \wedge B$ the machine is considered the winner if it wins in both of the components, while in $A \vee B$ it is sufficient to win in one of the components. Thus we have two sorts of conjunction: $\sqcap, \wedge$ and two sorts of disjunction: $\sqcup, \vee$. To appreciate the difference, let us compare the games $Chess \vee \neg Chess$ and $Chess \sqcup \neg Chess$. The former is, in fact, a simultaneous play on two boards, where on the left board the agent plays white, and on the right board plays black. There is a simple strategy for the agent that guarantees success in this game even when it is played against Kasparov himself. All that the agent needs to do is to mimic, in *Chess*, the moves made by Kasparov in $\neg Chess$, and vice versa. On the other hand, to win the game $Chess \sqcup \neg Chess$ is not easy: here, at the very beginning, the agent has to choose between *Chess* and $\neg Chess$ and then win the chosen one-board game. Generally, the principle $A \vee \neg A$ is valid in the sense that the corresponding problem is always solvable by a machine, whereas this is not so for $A \sqcup \neg A$.

While all the classical tautologies automatically hold when classical operators are applied to elementary games, in the general (non-elementary) case the class of valid principles shrinks. For example, $\neg A \vee (A \wedge A)$ is no longer valid. The above "mimicking strategy" would obviously fail to meet the purpose in the three-board game

$$\neg Chess \vee (Chess \wedge Chess),$$

for here the best the agent can do is to pair $\neg Chess$ with one of the two conjuncts of $Chess \wedge Chess$. It is possible that then $\neg Chess$ and the unmatched *Chess* are both lost, in which case the whole game will be lost.

The class of valid principles with $\wedge$, $\vee$ and $\neg$ forms a logic that resembles linear logic [10] with $\neg$ understood as linear negation and $\wedge, \vee$ as multiplicatives. This resemblance extends to the class of principles that also involve $\sqcap, \sqcup, \bigsqcap, \bigsqcup$, when these operators are understood as the corresponding additive operators of linear logic. In view of this, to the classical operators $\wedge, \vee$ we can also refer as "multiplicatives", and call $\sqcap, \sqcup, \bigsqcap, \bigsqcup$ "additives". The names that we may prefer to use, however, are "*parallel operations*" for $\wedge, \vee$ and "*choice operations*" for $\sqcap, \sqcup, \bigsqcap, \bigsqcup$.

The multiplicative/parallel/classical implication $\rightarrow$, as a game operation, is perhaps most interesting from the computability-theoretic point of view. Formally $A \rightarrow B$ can be defined as $\neg A \vee B$. The intuitive meaning of $A \rightarrow B$ is the problem of *reducing* problem B to problem A. Putting it in other words, solving $A \rightarrow B$ means solving B having A as an (external) *computational resource*. "Computational resource" is symmetric to "computational problem": what is a problem (task) for the machine, is a resource for the environment, and vice versa.

To get a feel of $\rightarrow$ as a problem reduction operator, let us consider a reduction of the acceptance problem to the halting problem. The halting problem can be expressed by

$$\bigsqcap x \bigsqcap y(Halts(x, y) \sqcup \neg Halts(x, y)),$$

where $Halts(x, y)$ is the predicate (elementary game) "Turing machine x halts on input y". And the acceptance problem can be expressed by

$$\sqcap x \sqcap y(Accepts(x, y) \sqcup \neg Accepts(x, y)),$$

with $Accepts(x, y)$ meaning "Turing machine x accepts input y". While the acceptance problem is not decidable, it is algorithmically reducible to the halting problem. In particular, there is a machine that always wins the game

$$\sqcap x \sqcap y(Halts(x, y) \sqcup \neg Halts(x, y)) \rightarrow \sqcap x \sqcap y(Accepts(x, y) \sqcup \neg Accepts(x, y)).$$

A strategy for solving this problem is to wait till the environment specifies values m and n for x and y in the consequent, then select the same values m and n for x and y in the antecedent (where the roles of the machine and the environment are switched), and see whether the environment responds by *left* or *right* there. If the response is *left*, simulate machine m on input n until it halts and then select, in the consequent, *left* or *right* depending on whether the simulation accepted or rejected. And if the environment's response in the antecedent was *right*, then select *right* in the consequent.

We can see that what the machine did in the above strategy was nothing but reducing the acceptance problem to the halting problem by querying the oracle (environment) for the halting problem—asking it whether machine m halts on input n. The usage of the oracle—that acts as an external computational resource for the machine—is, however, limited here. In particular, every legal question can be asked only once, and it cannot be re-asked later. For example, in the above example, after querying the oracle for the halting problem regarding machine m on input n, the agent would not be able to repeat the same query with different parameters m' and n', for that would require having two "copies" of the resource $\sqcap x \sqcap y(Halts(x, y) \sqcup \neg Halts(x, y))$ (which could be expressed by their $\wedge$-conjunction) rather than one. Taking into account this strict control over resource consumption, we call the sort of reduction captured by $\rightarrow$ *strong*, or *linear*, *reduction*. As we just saw, the acceptance problem is strongly reducible to the halting problem.

While linear reducibility is an interesting concept in its own rights, it does not capture our intuitive notion of algorithmic reducibility in the most general way, as it can be shown to be strictly stronger than Turing reducibility which is considered an adequate formalization of algorithmic reducibility of one simple (non-interactive) problem to another. What does capture the intuitive concept of reduction in full generality is another game operation that we call the *weak reduction* operation, denoted by $\Rightarrow$. Formally, $\Rightarrow$ can be defined in terms of $\rightarrow$ and (yet another natural) game operation $!$ by $A \Rightarrow B = !A \rightarrow B$. Here $!A$ is the game where the agent has to successfully play, in parallel, as many "copies" of A as the environment requests. The behavior of $!$ strongly resembles that of what is called *storage* in linear logic.

Back to weak reduction, the difference between $A \rightarrow B$ and $A \Rightarrow B$ is that, while in the former every act of resource consumption is strictly accounted for, the latter allows uncontrolled usage of computational resources (resource A). Indeed, having $!A$ in the antecedent of a $\rightarrow$-implication, where the roles of the players are switched,

obviously means having an unlimited capability to ask and re-ask questions regarding A. To get a feel of this, consider the Kolmogorov complexity problem. It can be expressed by $\sqcap t \sqcup z KC(z,t)$, where $KC(z,t)$ is the predicate "z is the smallest (code of a) Turing machine that returns t on input 0"—one of the equivalent ways to say that z is the Kolmogorov complexity of t. Having no algorithmic solution, the Kolmogorov complexity problem, however, is algorithmically reducible to the halting problem. In our terms, this means nothing but that there is a machine that always wins the game

$$\sqcap x \sqcap y(Halts(x,y) \sqcup \neg Halts(x,y)) \Rightarrow \sqcap t \sqcup z KC(z,t).$$

Here is a strategy for such a machine: Wait till the environment selects a particular value m for t in the consequent. Then, starting from $i=0$, do the following: create a new copy of the (original) antecedent, and make a move in it by specifying x and y as i and 0, respectively. If the environment responds by *right*, increment i by one and repeat the step; if the environment responds by *left*, simulate machine i on input 0 until it halts; if you see that machine i returned m, make a move in the consequent by specifying z as i; otherwise, increment i by one and repeat the step.

We have just demonstrated an example of a weak reduction to the halting problem, which cannot be replaced with a linear reduction. A convincing case can be made in favor of the thesis that the semantics of $\Rightarrow$ adequately captures our weakest possible intuitive notion of reducing one interactive computational problem to another. Once we accept this thesis, algorithmic reducibility of problem B to problem A means nothing but existence of a machine that always wins the game $A \Rightarrow B$.

What are the valid principles of $\Rightarrow$? There are reasons to expect that the logic of weak reduction is exactly the implicative fragment of Heyting's intuitionistic calculus; furthermore, this conjecture extends to the logic of weak reducibility combined with choice operations, where intuitionistic implication, conjunction, disjunction, universal quantifier and existential quantifier are understood as $\Rightarrow$, $\sqcap, \sqcup, \sqcap$ and $\sqcup$, respectively.

Along with additive quantifiers we also have a different sort of quantifiers—$\forall$ and its dual $\exists$ $(=\neg\forall\neg)$—that can be called "blind quantifiers", with no counterparts in linear logic. The meaning of $\forall x A(x)$ is similar to that of $\sqcap x A(x)$, with the difference that the particular value of x that the user "selects" is invisible to the machine, so that it has to play blindly in a way that guarantees success no matter what that value is. This way, $\forall$ and $\exists$ produce games with *imperfect information*.

Compare the problems $\sqcap x(Even(x) \sqcup Odd(x))$ and $\forall x(Even(x) \sqcup Odd(x))$. Both of them are about telling whether a given number is even or odd; the difference is only in whether that "given number" is communicated to the machine or not. The first problem is an easy-to-win, two-move-deep game of a structure that we have already seen. The second game, on the other hand, is one-move deep with only by the machine to make a move—select the "true" disjunct, which is hardly possible to do as the value of x remains unspecified.

As an example of a solvable non-elementary $\forall$-problem, look at

$$\forall x(Even(x) \sqcup Odd(x) \rightarrow \sqcap y(Even(x+y) \sqcup Odd(x+y))),$$

solving which means solving what follows "$\forall x$" without knowing the value of x. Unlike $\forall x(Even(x) \sqcup Odd(x))$, this game is certainly winnable: The machine waits till the environment selects a value n for y in the consequent and also selects one of the $\sqcup$-disjuncts in the antecedent (if either selection is never made, the machine automatically wins). Then: If n is even, in the consequent the machine makes the same move *left* or *right* as the environment made in the antecedent, and otherwise, if n is odd, it reverses the environment's move.

When applied to elementary games only, the blind quantifiers, just like $\neg$ or the multiplicative operations, behave exactly in the classical way, which allows us to refer to them as "classical operators" along with $\neg, \wedge, \vee$ and $\rightarrow$. $\forall$ and $\exists$, however, cannot be considered "big brothers" of $\wedge$ and $\vee$, as they are game operations of a different type, different from what can be called *multiplicative quantifiers*. Multiplicative quantifiers technically would be easy to define in the same style as $\wedge$ and $\vee$, but at this point we refrain from including them in the language of computability logic for the reasons of simplicity, on one hand, and for the absence of strong computability-theoretic motivation on the other hand. Out of similar considerations, we are not attempting to introduce blind versions of conjunction and disjunction.

In most computability theory books or papers, the term "computational problem" means a simple problem, usually of one of the two types: $\sqcap x \sqcup y(f(x) = y)$ or $\sqcap x(P(x) \sqcup \neg P(x))$. As the above examples demonstrated, however, what can be considered an adequate formal equivalent of our broader intuition of computational problems, goes far beyond this simple sort of problems. Computational problems of a higher degree of interactivity and complexity of their structure emerge naturally and, as we noted, have to be addressed in any more or less advanced study in computability theory. So far this has been mostly done in an ad hoc manner as there has been no universal language for describing complex problems.[1] The logical formalism introduced in the present paper offers a convenient language for specifying generalized computational problems and studying them in a systematic way. Succeeding in finding an axiomatization of the corresponding logic or, at least, some reasonably rich fragments of it, would have not only theoretical, but also high practical significance. Among the applications would be the possibility to build computability logic into a machine and then use such a machine for systematically finding solutions to new problems. In particular, if the principle $A_1 \wedge \cdots \wedge A_n \rightarrow B$ is provable in our logic and solutions for $A_1, \ldots, A_n$

[1] Beginning from [7], interactive/alternating type computations and similar concepts such as interactive proof systems, have been extensively studied in theoretical computer science. But those studies do not offer a comprehensive specification language for interactive problems, for they are, in fact, only about new, *interactive models of computation* for old, essentially non-interactive types of problems, and make sense only in the context of computational complexity. Our approach, which is concerned with computability rather than complexity, at this point is only very remotely related to that line of research, and the similarity is more terminological than conceptual.

are known, then, using a proof of $A_1 \wedge \cdots \wedge A_n \rightarrow B$ and querying the programs that solve $A_1, \ldots, A_n$, the machine could be able to automatically produce a solution for B.

Part I—Computational problems, i.e. games

We are getting to formal definitions of some key concepts on games, including those informally introduced in the previous section. Having said enough, in Section 1, about the intuitive meaning of games as interactive computational problems, this part will be written mostly in game-theoretic terms. The players that we called the user and the machine will now be renamed into the more technical $\bot$ and $\top$, respectively.

2. Formal definition of games

We fix three infinite sets of expressions:

- **Moves**, whose elements are called *moves*. **Moves** is the set of all finite strings over some fixed finite alphabet called **Move alphabet**. We assume that **Move alphabet** contains each of the following 13 symbols: $0, \ldots, 9, ., :, \spadesuit$ and does not contain any of the three symbols $\top$, $\bot$, $-$.
- **Constants**, whose elements are called *constants*. Constants are all possible (names of the) elements of the universe of discourse. Without loss of generality, we assume that $\textbf{Constants} = \{0, 1, 2, \ldots\}$.
- **Variables**, whose elements are called *variables*. Variables will range over constants. We assume that $\textbf{Variables} = \{v_0, v_1, \ldots\}$.

The set of *terms* is defined as the union of **Variables** and **Constants**.

A move prefixed with $\bot$ or $\top$ will be called a ($\bot$- or $\top$-) *labeled move*, or *labmove* for short, where the label $\bot$ or $\top$ indicates which player has made the move. Sequences of labeled moves we call *runs*, and finite runs call *positions*. Runs and positions will often be delimited with "$\langle$", "$\rangle$", as in $\langle \top 0, \bot 1.1, \bot 0 \rangle$. $\langle\rangle$ will thus stand for the *empty position*.

A *valuation* is a function e of the type $\textbf{Variables} \rightarrow \textbf{Constants}$.

For convenience of future references, let us fix the following names: **Runs** for the set of all runs, **Valuations** for the set of all valuations and **Players** for the set $\{\bot, \top\}$.

Throughout the paper we will be using certain letters—with or without indices—as dedicated metavariables (unless specified otherwise) for certain classes of objects. These metavariables will usually be assumed universally quantified in the context unless otherwise specified. In particular:

- $\wp$ will range over players. $\neg\wp$ will stand for the "adversary" of player $\wp$. That is, if $\wp = \bot$, then $\neg\wp = \top$, and if $\wp = \top$, then $\neg\wp = \bot$.
- α, β will range over moves.
- λ will range over labeled moves.
- Φ, Ψ, Θ (can also be written as $\langle\Phi\rangle$, $\langle\Psi\rangle$, $\langle\Theta\rangle$) will range over positions.
- Γ, Δ (or $\langle\Gamma\rangle$, $\langle\Delta\rangle$) will range over runs. $\langle\Phi, \Gamma\rangle$ will mean the concatenation of position Φ and run Γ, $\langle\lambda, \Gamma\rangle$ will mean the concatenation of position $\langle\lambda\rangle$ and run Γ, etc.

- x, y, z will range over variables, and $\vec{x}, \vec{y}, \vec{z}$ over tuples, sequences or sets of variables.
- c will range over constants, and $\vec{c}$ over tuples, sequences or sets of constants.
- e will range over valuations.
- A, B, C, D will range over games (in the sense of Definition 2.1 below).

Remembering this convention is strongly recommended, as we will often be using the above metavariables without specifying/reminding for what types of objects they stand. The following definition is the first example of doing so, where it is only implicitly assumed that $\wp$ is a player, e is a valuation, Φ is a position and Γ is a run.

Definition 2.1. A *game* A is a pair $(\mathbf{Lr}^A, \mathbf{Wn}^A)$ where:

1. $\mathbf{Lr}^A$ is a function that maps valuations to subsets of **Runs** such that, writing $\mathbf{Lr}^A_e$ for $\mathbf{Lr}^A(e)$, the following conditions are satisfied:
   (a) A finite or infinite run Γ is in $\mathbf{Lr}^A_e$ iff all of its non-empty finite (not necessarily proper) initial segments are in $\mathbf{Lr}^A_e$.
   (b) $\langle \Phi, \wp\spadesuit \rangle \notin \mathbf{Lr}^A_e$.
   Elements of $\mathbf{Lr}^A_e$ are called *legal runs of A with respect to* (w.r.t.) e, and all the other runs called *illegal runs of A* w.r.t. e. In particular, if the last move of the shortest illegal initial segment of Γ is $\wp$-labeled, then Γ is said to be a *$\wp$-illegal run of A* w.r.t. e.
2. $\mathbf{Wn}^A$ is a function of the type $\mathbf{Valuations} \times \mathbf{Runs} \to \mathbf{Players}$ such that, writing $\mathbf{Wn}^A_e\langle\Gamma\rangle$ for $\mathbf{Wn}^A(e, \Gamma)$, the following condition is satisfied:
   (a) If Γ is a $\wp$-illegal run of A w.r.t. e, then $\mathbf{Wn}^A_e\langle\Gamma\rangle = \neg\wp$.

Along with $\mathbf{Lr}^A_e$ we will also be using the notation $\mathbf{Lm}\wp^A_e\langle\Phi\rangle$ defined by

$$\mathbf{Lm}\wp^A_e\langle\Phi\rangle = \{\alpha \mid \langle \Phi, \wp\alpha\rangle \in \mathbf{Lr}^A_e\}.$$

Here **Lm** abbreviates "**L**egal **m**oves" vs. **Lr** which abbreviates "**L**egal **r**uns". The meaning of $\mathbf{Lm}\wp^A_e\langle\Phi\rangle$ is that it tells us what moves are *legal* for player $\wp$ in position Φ when A is played, where a move α is considered legal for $\wp$ in a given position iff adding $\wp\alpha$ to that position yields a legal position. Whether a given position is legal or not may depend on what particular values of variables we have in mind, so both $\mathbf{Lr}^A_e$ and $\mathbf{Lm}\wp^A_e$ take valuation e as an (additional) argument. Some runs or moves of the game may be legal w.r.t. every valuation. We call such moves or runs *unilegal*. We will be using the notation $\mathbf{LR}^A$ to denote the set of unilegal runs of A. That is,

$$\mathbf{LR}^A = \bigcap_e \mathbf{Lr}^A_e.$$

$\mathbf{Lr}^A_e$ can be defined in terms of $\mathbf{Lm}\wp^A_e$ as follows:

$$\mathbf{Lr}^A_e = \{\Gamma \mid \text{for every initial segment } \langle \Phi, \wp\alpha\rangle \text{ of } \Gamma, \alpha \in \mathbf{Lm}\wp^A_e\langle\Phi\rangle\}.$$

Since **Lr** and **Lm** are interdefinable, we can use either of them when defining a given game. Intuitively, however, the primary between these two is **Lm** rather than **Lr**: a run is legal iff no illegal moves have been made in it. This automatically implies condition 1(a) that had to be explicitly stated in Definition 2.1 because it uses **Lr** instead of

Lm. Note that the empty position $\langle\rangle$ is a unilegal position of any game. This formally follows from the above condition 1(a). Intuitively, $\langle\rangle$ is legal because no moves and hence no illegal moves have been made in it.

The meaning of $\mathbf{Wn}_e^A\langle\Gamma\rangle$ is that it tells us which of the players has won game A. Who the winner is, of course, depends on how the game was played, that is, what run of this play has been generated, so $\mathbf{Wn}_e^A$ takes a run Γ as an argument. And, just as in the case of **Lr**, a valuation e is taken as an additional argument.

According to the definition, a $\wp$-illegal run is an illegal run where $\wp$ has made the first illegal move. When modeling real-life situations, such as computer–user, robot–environment, server–client interactions etc., where moves represent actions, illegal moves can be considered actions that can, will or should never be done. However, not ruling out illegal runs from considerations is a very useful and convenient mathematical abstraction, and in our approach we do not formally assume that illegal moves will never be made. Whether we rule out or not the possibility of making illegal moves makes no difference whatsoever for the players and the chances of their strategies to be successful: a player, whose only goal is to win the game, can always safely assume that its adversary will never make an illegal move for, if it does, the player will automatically become the winner and its goal will be accomplished. This is so according to condition 2(a) of Definition 2.1. The set **Moves** is selected very generously, and it is natural to expect that there will always be some elements of this set that are not among legal moves. However, we may have different illegal moves for different players, games, positions or valuations. Again, a technically very convenient assumption is that there is at least one move that is always illegal. According to condition 1(b) of Definition 2.1, ♠ is such a move. Making this move can be thought of as a standard way for a player to surrender.

We call the **Lr** component of a game its *structure* and the **Wn** component its *contents*.

Remark 2.2. In order to define the contents of a game, it is sufficient to define it only for legal runs for, as we noted, an illegal run is always lost by the player who made the first illegal move. Similarly, in view of condition 1(a) of Definition 2.1, in order to define the structure of a game, it is sufficient to define what non-empty finite runs, i.e. positions, are legal. Many of the definitions of particular games found in this paper will rely on this fact without explicitly mentioning it. Some of those definitions will define the structure of a game in terms of **Lm** rather than **Lr**. In such cases, it is sufficient to define the values of **Lm** only for legal positions. There would be no circularity in this, because we know that the empty position is always legal, and even though the question whether a non-empty position Φ is legal depends on the **Lm** function, it only depends on the values of that function for positions shorter than Φ itself.

Let us agree on some jargon some of which in fact we have already started using in the above paragraphs. When $\langle\wp\alpha\rangle\in\mathbf{Lr}_e^A$, we say that $\wp\alpha$ is a *legal initial labmove of* A w.r.t. e, or that α is $\wp$'s *legal initial move in* A w.r.t. e; of course, the word "illegal" will be used here instead of "legal" if $\langle\wp\alpha\rangle\notin\mathbf{Lr}_e^A$. When $\langle\wp\alpha\rangle\in\mathbf{LR}^A$, we say that $\wp\alpha$ is a *unilegal initial labmove of* A, or that α is $\wp$'s *unilegal initial move in* A. When

$\mathbf{Wn}_e^A\langle\Gamma\rangle = \wp$, we may say that Γ is a *$\wp$-won run of A* w.r.t. e, or a *run of A won by $\wp$* w.r.t. e. Of course, if $\mathbf{Wn}_e^A\langle\Gamma\rangle \neq \wp$, we would use the word "lost" instead of "won". Simply saying "won" or "lost" without specifying the player will usually mean $\top$-won or $\top$-lost. When e is irrelevant or fixed in the context, we will often omit the words "w.r.t. e", or omit the subscript e in $\mathbf{Lr}_e^A$, $\mathbf{Lm}\wp_e^A$, $\mathbf{Wn}_e^A$. Finally, if both A and e are fixed in the context or A is fixed and e is irrelevant, we will simply say "Γ is won (legal, etc.)" instead of "Γ is a won (legal, etc.) run of A w.r.t. e".

3. Structural game properties

The game properties defined below[2] can be called *structural properties*, because they only depend on the structure of a game and not on its contents.

Definition 3.1. We say that game A is:

1. *Strict* iff, for every e and Φ, either $\mathbf{Lm}\bot_e^A\langle\Phi\rangle = \emptyset$ or $\mathbf{Lm}\top_e^A\langle\Phi\rangle = \emptyset$.
2. *Elementary* iff, for every e, $\mathbf{Lr}_e^A = \{\langle\rangle\}$.
3. *Finite-depth* iff there is an integer n such that, for every e, the length of each element of $\mathbf{Lr}_e^A$ is $\leqslant n$.
4. *Perifinite-depth* iff, for every e, the length of each element of $\mathbf{Lr}_e^A$ is finite.
5. *Unistructural* iff, for every e, $\mathbf{Lr}_e^A = \mathbf{LR}^A$.
6. *x-unistructural*, or *unistructural in x*, iff, for any two valuations e_1 and e_2 that agree on all variables except x, $\mathbf{Lr}_{e_1}^A = \mathbf{Lr}_{e_2}^A$.

Thus, in a strict game, at most one of the players can have legal moves in a given position. Most of the authors who studied game semantics in the past, including the author of this paper [12], only considered strict games. In the versions of games by some other authors [8,1] there is no formal requirement that only one of the players can have legal moves in a given position, but this is only a terminological issue, as those games are still "essentially strict" (henceforth referred to simply as "strict") in the sense that in every position of every particular play, it is still the case that only one of the players can make a move, even if the other player also has legal moves in the same position. This is a consequence of having strict *procedural rules*—rules that govern who and when can move, the most standard procedural rule being the rule according which the players should move (take turns) in strictly alternating order.

In contrast, in our present approach, we do not impose the requirement of strictness as this can be seen from Definition 2.1, nor are we going to have any procedural rules whatsoever. How games are played up will be formally defined in Part II. At this point we can just adopt an informal but yet perfectly accurate explanation according to which either player can make any move at any time. This makes our games what can be called *free*. As a consequence, our games are most general—and hence most natural, stable and immune to possible future revisions and tampering—of all two-player,

[2] The author is grateful to Scott Weinstein for his help in selecting some terminology used in this definition.

two-outcome games, and they are also most powerful and universal as a modeling tool for various sorts of situations of interactive nature.

Situations that make free games an at least more direct and adequate—if not the only possible—modeling tool emerge even in very simple cases such as parallel plays of chess discussed in Section 1. Imagine an agent who plays white, over the Internet, a parallel game (whether it be a $\wedge$- or $\vee$-combination) on two chessboards with two independent adversaries that, together, form the (one) environment for the agent. At the beginning certainly only the agent has legal moves. But after the agent makes his first move, say, on board #1, the situation changes. Now both the agent and the environment have legal moves: the agent can make another opening move on board #2, while the environment can make a reply move on board #1. As the two adversaries of the agent are independent, one cannot expect any coordination between their actions, such as adversary #1 waiting for the agent to make a move on board #2 before he (adversary #1) replies to the agent's opening move on board #1. So, whether the second move in this parallel play will be made by the agent or the environment, may simply depend on who can or wants to act sooner.

Most of the interactive problems we deal with in everyday life are free rather than strict. An agent whose goal is to be a good family man and, at the same time, a good worker [3], thus playing the $\wedge$-combination of these two subgoals/subtasks against the environment that consists of two independent "adversaries" of the agent—his wife and his boss—is playing a free game, for the boss is unlikely to wait with his request to start working on the new project till the agent satisfies his wife's request to fix the malfunctioning washing machine in the basement.

And in the world of computers that is the main target of applications for our concepts, hardly many communication or interaction protocols are really strict. A parallel combination—whether it be a $\wedge$-, $\vee$- or some other type of combination—of tasks/dialogues/processes usually means real parallelism, with little or no coordination between different subprocesses, which makes such a combination free even if each individual component of it is strict.

A substantial difference between strict and free games, including the subclass of free games that we are going to call "static games" in the next section—a difference that may essentially affect players' strategies and their chances to succeed—is that, in free games, players may have to decide not only *which* of the available moves to make, but sometimes also *when* to make moves, i.e. *whether* to move in a given situation at all or wait to see how the adversary acts; moreover, while they are thinking on their decision, they can, simultaneously, keep watching what the adversary is doing, hoping to get some additional information that may help them to come up with a good decision. The following example demonstrates an essential use of this kind of "think and watch" strategy. (Even though the game operations used in it were informally explained in Section 1, the reader may want to postpone looking into this example until later when (s)he feels more comfortable after having seen all the relevant formal definitions.)

Example 3.2. Let $A(x,z)$ be a decidable arithmetical predicate such that the predicate $\forall z A(x,z)$ is undecidable, and let $B(x,y)$ be an undecidable arithmetical predicate.

Consider the following computational problem:

$$\sqcap x\left(\sqcap y(\forall zA(x,z)\wedge B(x,y))\sqcap\sqcap zA(x,z)\rightarrow\forall zA(x,z)\wedge\sqcap yB(x,y)\right).$$

After $\bot$ specifies a value m for x, $\top$ will seemingly have to decide what to do: to watch or to think. The 'watch' choice is to wait till $\bot$ specifies a value k for y in the consequent, after which $\top$ can select the $\sqcap$-conjunct $\sqcap y(\forall zA(m,z)\wedge B(m,y))$ in the antecedent and specify y as k in it, thus bringing the play down to the always-won elementary game $\forall zA(m,z)\wedge B(m,k)\rightarrow\forall zA(m,z)\wedge B(m,k)$. While being successful if $\forall zA(m,z)$ is true, the watch strategy is a bad choice when $\forall zA(m,z)$ is false, for there is no guarantee that $\bot$ will make a move in $\sqcap yB(m,y)$, and if not, the game will be lost. When $\forall zA(m,z)$ is false, the following 'think' strategy is successful: Start looking for a number n for which $A(m,n)$ is false. This can be done by testing $A(m,z)$, in turn, for $z=0$, $z=1,\ldots$. After you find n, select the $\sqcap$-conjunct $\sqcap zA(m,z)$ in the antecedent, specify z as n in it, and celebrate victory. The trouble is that if $\forall zA(m,z)$ is true, such a number n will never be found. Thus, which of the above two choices (watch or think) would be successful depends on whether $\forall zA(m,z)$ is true or false, and since $\forall zA(x,z)$ is undecidable, $\top$ has no way to make the right choice. Fortunately, there is no need to choose. Rather, these two strategies can be pursued simultaneously: $\top$ starts looking for a number n which makes $A(m,n)$ false and, at the same time, periodically checks if $\bot$ has made a move in $\sqcap yB(m,y)$. If the number n is found before $\bot$ makes such a move, $\top$ continues as prescribed by the think strategy; if vice versa, $\top$ continues as prescribed by the watch strategy; finally, if none of these two events ever occur, which, note, is only possible when $\forall zA(m,z)$ is true (for otherwise a number n falsifying $A(m,n)$ would have been found), again $\top$ will be the winner because, just as in the corresponding scenario of the watch strategy, it will have won both of the conjuncts of the consequent.

This is an example of a computational problem the essence of which is impossible to capture within the framework of strict games in the style of [12]. The strictness set serious limitations to that framework,[3] precluding it from maturing into a comprehensive logic of computability, and bringing the whole approach into a dead end. The new line taken in the present paper signifies a decisive breakthrough from that dead end.

Back to the other structural game properties, it is obvious that elementary games enjoy all of the other five structural properties, and that all finite-depth games are also perifinite-depth, but not vice versa. The games that are not perifinite-depth we can call *infinite-depth*. The concept of the depth of a game will be formally defined later in Section 8. Very roughly, it can be thought of as the maximum possible length of a legal run of the game.

Intuitively, game A is x-unistructural if the structure of A does not depend on x—more precisely, whether a given run is a legal run of A w.r.t. a given valuation does not depend on the value returned for x by that valuation. For (simply) unistructural

[3] Among such limitations were the necessity to insist that elementary games only be decidable predicates, and the impossibility to introduce anything similar to our present blind quantifiers or our present (Section 26) concept of uniform validity.

games, the above question does not depend on the valuation at all. Hence the valuation parameter can always be omitted when talking about what runs or moves of a unistructural game are legal. Virtually all examples discussed in this paper deal with unistructural games. However, natural examples of non-unistructural (*multistructural*) games do exist. Let the reader try to find some.

Definitions of any game properties, such as the above structural properties, can be naturally extended to game operations by stipulating that a game operation has a given property (is strict, elementary, etc.) if and only if it preserves that property—that is, whenever all the game arguments of the operation have the given property and the operation is defined for those arguments, the game returned by the operation also has that property. That a certain game property P_1 is stronger than a game property P_2 generally does not imply that the same 'stronger-than' relation will hold for P_1 and P_2 understood as game operation properties. For example, the multiplicative operations defined in Section 12 are elementary but not strict.

4. Static games

The game property that we call static property is most important in our study and, as some discussions below may suggest, is likely to find use beyond our semantics as well.[4]

Our free games are obviously general and powerful enough to expect that they can model everything that anyone would ever call a (two-outcome) interactive computational problem/task. In other words, our concept of games is complete as a formal counterpart of the intuitive notion of interactive problems. However, it would be harder to argue that it is sound as well. There are some free games that may not represent any meaningful or well-defined computational problems in some or most people's perception.

Consider the game F that only has two non-empty legal runs: $\langle\top 0\rangle$, won by $\top$, and $\langle\bot 0\rangle$, won by $\bot$. Whoever moves first thus wins. This is a contest of speed rather than intellect. If communication happens by exchanging messages through a (typically) asynchronous network, that often has some unpredictable delays, this can also be a contest of luck: assuming that the arbitration is done by $\neg\wp$ or a third party who is recording the order of moves, it is possible that $\wp$ makes a move earlier than $\neg\wp$ does, but the message carrying that move is delivered to the arbiter only after $\neg\wp$'s move arrives, and the arbiter will be unable to see that it was $\wp$ who moved first. An engineering-style attempt to solve this problem could be to assume that moves always carry timestamps. But this would not save the case: even though timestamps would correctly show the real order of moves by each particular player, they could not be used to compare two moves by two different players, for the clocks of the two players would never be perfectly synchronized.

[4] One of such "unintended" applications, the possibility of which was pointed out to the author by Andreas Blass, can be giving an intuitive game-interpretation to Joyal's [17] free bicomplete categories.

Another attempt to deal with this problem could be to assign to each player, in a strictly alternating order, a constant-length time slot during which the player has exclusive access to the communication medium. Let alone that this could introduce some unfair asymmetry in favor of the player who gets the first slot, the essence of the problem would still not be taken care of—in particular, the fact that some games would still essentially depend on the relative speeds of the two players, even if (arguably) no longer on the speed of the network. This kind of games might be perceived as meaningful interactive problems/tasks by an engineer, but less likely so by a theoretical scientist, to whom it would not be clear what particular speed would be natural to assume for $\bot$ who represents the environment with its indetermined and unformalizable behavior.

We want to consider only what can be called "*pure*" problems: interactive tasks that remain meaningful and well defined without any particular assumptions regarding the speeds of the interacting agents, the exact timing behavior of the communication hardware and similar annoying details—tasks where, for being successful, only matters *what* to do rather than *how fast* to do. As long as we agree that only this kind of tasks are what we perceive as (meaningful, well-defined) interactive computational problems, our formal concept of games becomes certainly too general to be a sound formal counterpart of our intuitive notion of interactive problems. What has all the chances to fit the bill, however, is the formal concept of static games that we are now going to introduce.

Definition 4.1. We say that run Δ is a $\wp$-*delay* of run Γ iff

1. for each $\wp' \in$ **Players**, the subsequence of $\wp'$-labeled moves of Δ is the same as that of Γ, and
2. for any $n, m \geqslant 1$, if the nth $\wp$-labeled move is made later than (is to the right of) the mth $\neg\wp$-labeled move in Γ, then so is it in Δ.

Intuitively, "Δ is a $\wp$-delay of Γ" means that in Δ each player has made the same sequence of moves as in Γ, only, in Δ, $\wp$ might have been acting with some delay. Technically, Δ can be thought of as the result of repetitively swapping, in Γ, some (possibly all, possibly none) $\wp$-labeled moves with their $\neg\wp$-labeled right-hand neighbors. In other words, it is the result of "shifting" some (maybe all, maybe none) $\wp$-labeled moves to the right; some of such labmoves can be shifted farther than others, but their order should be preserved, i.e. a $\wp$-labeled move can never jump over another $\wp$-labeled move.

Definition 4.2. A game A is said to be *static* iff, for all $\wp$, Γ, Δ and e, where Δ is a $\wp$-delay of Γ, we have: if Γ is a $\wp$-won run of A w.r.t. e, then so is Δ.

Example 4.3. To see why static games are really what we are willing to call "pure" computational problems, imagine a play over the delay-prone Internet. If there is a central arbiter (that can be located either in one of the players' computer or somewhere on a third, neutral territory) recording the order of moves, then the players have full access to information about the official version of the run that is being generated,

even though they—or, at least, one of them—could suffer from their moves being delivered/acknowledged with delays. But let us make the situation even more dramatic: assume, as this is a typical case in distributed systems, that there is no central arbiter. Rather, each players' machine records moves in the order it receives them, so that we have what is called distributed arbitration. Every time a player makes a move, the move is appended to the player's internal record of the run and, at the same time, mailed to the adversary. And every time a message from the adversary arrives, the move contained in it is appended to the player's record of the run. The game starts. Seeing no messages from $\bot$, $\top$ decides to go ahead and make an opening move α_1. As it happens, $\bot$ also decides to make an "opening" move β_1. The messages carrying $\top\alpha_1$ and $\bot\beta_1$ cross. So, after they are both delivered, $\top$'s internal records show the position $\langle\top\alpha_1, \bot\beta_1\rangle$, while $\bot$ thinks that the current position is $\langle\bot\beta_1, \top\alpha_1\rangle$. Both of the players decide to make two consecutive new moves: $\langle\alpha_2, \alpha_3\rangle$ and $\langle\beta_2, \beta_3\rangle$, respectively, and the two pairs of messages, again, cross. After making their second series of moves and receiving a second series of "replies" from their adversaries, both players decide to make no further moves. The game thus ends. According to $\top$'s records, the run was $\langle\top\alpha_1, \bot\beta_1, \top\alpha_2, \top\alpha_3, \bot\beta_2, \bot\beta_3\rangle$,[5] while $\bot$ thinks that the run was $\langle\bot\beta_1, \top\alpha_1, \bot\beta_2, \bot\beta_3, \top\alpha_2, \top\alpha_3\rangle$. As for the "real run", i.e. the real order in which these six moves were made (if this concept makes sense at all), it can be yet something different, such as, say, $\langle\bot\beta_1, \top\alpha_1, \top\alpha_2, \bot\beta_2, \bot\beta_3, \top\alpha_3\rangle$. A little thought can convince us that in any case the real run—as well as the version of the run seen by player $\neg\wp$—will be a $\wp$-delay of the version of the run seen by player $\wp$. Hence, provided that the game is static, $\wp$ can fully trust his own version of the run (position) and only care about making good moves for this version, because, no matter if it shows the true or a distorted picture of the real run, the latter is guaranteed to be successful as long as the former is. Moreover, for similar reasons, the player will remain equally successful if, instead of immediately appending the adversary's moves to his version of the run, he simply queues those moves in a buffer as if they had not arrived yet, and fetches them only later at a more convenient time—after, perhaps, making and appending to his records some of his own moves first. The effect will amount to having full control over the speed of the adversary, thus allowing the player to select his own pace for the play and worry only about what moves to make rather than how fast to make them. This fact, formulated in more precise terms, will be formally proven later in Part II (Theorem 17.2).

Thus, static games allow players to make a full abstraction from any specific assumptions regarding the type of arbitration (central or distributed), the speed of the communication network and the speed of the adversary: whatever strategy they follow, it is always safe to assume or pretend that the arbitration is fair and unique (central), the network is perfectly accurate (zero delays) and the adversary is "slow enough".

[5] The order of β_2 and β_3 would not be confused, as we may assume that either the communication medium does not allow a later message from the same source to jump over an earlier message, or that those messages carry ordinal stamps, so that even if β_3 arrives earlier than β_2, $\top$ will still be able to figure out that β_3 is a later move.

On the other hand, with some additional thought, we can see that if a game is not static, there will always be situations when no particular one of the above three sorts of abstractions can be made. Specifically, such situations will emerge every time when $\wp$'s strategy generates a $\wp$-won run that has some $\wp$-lost $\wp$-delays. To summarize, there are all reasons to accept the following thesis stating equivalence between static games what we tried to characterize as "pure" computational problems:

Thesis 4.4. *The concept of static games is an adequate formal counterpart of our intuitive notion of well-defined, speed-independent, "pure" computational problems.*

Let us now look at some simple technical facts related to static games.

Lemma 4.5. *Suppose Δ is a $\wp$-delay of Γ and $\neg\wp\alpha$ is the first labmove of Γ. Then $\neg\wp\alpha$ is the first labmove of Δ as well.*

Proof. Indeed, the first move of Δ cannot be $\wp$-labeled, because this would violate condition 2 of Definition 4.1. And it cannot be any other $\neg\wp$-labeled move but $\neg\wp\alpha$, because this would violate condition 1. □

Lemma 4.6. *If Δ is a $\wp$-delay of Γ, then Γ is a $\neg\wp$-delay of Δ.*

Proof. Suppose Δ is a $\wp$-delay of Γ. Condition 1 of Definition 4.1 is symmetric, so it will continue to be satisfied with $\Gamma, \Delta, \neg\wp$ in the roles of $\Delta, \Gamma, \wp$, respectively. As for condition 2, assume that, in Δ, the nth $\neg\wp$-labeled move is made later than the mth $\wp$-labeled move. We need to show that then, in Γ, we also have that the nth $\neg\wp$-labeled move is made later than the mth $\wp$-labeled move. Suppose this was not the case. That is, the mth $\wp$-labeled move is made later than the nth $\neg\wp$-labeled move in Γ. Since Δ is a $\wp$-delay of Γ, we then have (with the roles of m and n interchanged) that the mth $\wp$-labeled move is made later than the nth $\neg\wp$-labeled move in Δ, which is a contradiction. □

Lemma 4.7. *Assume A is a static game, e is any valuation and Δ is a $\wp$-delay of Γ. Then we have:*
1. *If Δ is a $\wp$-illegal run of A w.r.t. e, then so is Γ.*
2. *If Γ is a $\neg\wp$-illegal run of A w.r.t. e, then so is Δ.*

Proof. Let us fix throughout the context (and no longer explicitly mention) an arbitrary static game and an arbitrary valuation. Assume Δ is a $\wp$-delay of Γ.

Clause 1: Assume Δ is $\wp$-illegal. As a $\wp$-illegal run, Δ should have the shortest illegal initial segment that looks like $\langle \Psi, \wp\alpha\rangle$. Assume this position has m $\wp$-labeled and n $\neg\wp$-labeled moves. Thus, $\wp\alpha$ is the mth $\wp$-labeled move of Δ. Consider the result of deleting, in Γ, all the labmoves except its first m $\wp$-labeled moves and first n $\neg\wp$-labeled moves. It will look like $\langle \Phi, \wp\alpha, \Theta\rangle$, where Θ only consists of $\neg\wp$-labeled moves. It is obvious that

$$\langle \Psi, \wp\alpha\rangle \text{ is a } \wp\text{-delay of } \langle \Phi, \wp\alpha, \Theta\rangle. \tag{1}$$

Suppose that $\langle \Phi, \wp\alpha, \Theta\rangle$ is not $\wp$-illegal. Then, since $\spadesuit$ is an always-illegal move, $\langle \Phi, \wp\alpha, \Theta, \neg\wp\spadesuit\rangle$ is $\neg\wp$-illegal and hence won by $\wp$. In view of (1), obviously $\langle \Psi, \wp\alpha, \neg\wp\spadesuit\rangle$ is a $\wp$-delay of $\langle \Phi, \wp\alpha, \Theta, \neg\wp\spadesuit\rangle$ and hence also won by $\wp$. But this is impossible because $\langle \Psi, \wp\alpha\rangle$ and hence $\langle \Psi, \wp\alpha, \neg\wp\spadesuit\rangle$ is $\wp$-illegal.

From this contradiction we conclude that $\langle \Phi, \wp\alpha, \Theta\rangle$ must be $\wp$-illegal. Taking into account that Θ does not contain any $\wp$-labeled moves, we must have that $\langle \Phi, \wp\alpha\rangle$ is also $\wp$-illegal. But notice that $\langle \Phi, \wp\alpha\rangle$ (even though not necessarily $\langle \Phi, \wp\alpha, \Theta\rangle$) is an initial segment of Γ, and hence Γ is $\wp$-illegal.

Clause 2: Assume Γ is $\neg\wp$-illegal. By Lemma 4.6, Γ is a $\neg\wp$-delay of Δ which, by the already proven clause 1, implies that Δ is $\neg\wp$-illegal. □

Obviously elementary games are static: their only legal run $\langle\,\rangle$ does not have any proper $\wp$-delays, and if we consider a non-empty run Γ won by $\wp$, we must have that the adversary of $\wp$ has made the first (illegal) move in Γ. Then, by Lemma 4.5, any $\wp$-delay of Γ will have the same first labmove and hence will also be won by $\wp$ for the same reason as Γ was won. Moreover, we have:

Proposition 4.8. *Every strict game is static.*

Proof. Let us fix an arbitrary strict game and an arbitrary valuation. Assume Γ is $\wp$-won and Δ is a $\wp$-delay of Γ. Our goal is to show that Δ, too, is $\wp$-won. The case $\Delta = \Gamma$ is trivial, so assume that Δ is a proper $\wp$-delay of Γ. Then we must have $\Gamma = \langle \Phi, \wp\alpha, \Gamma'\rangle$ and $\Delta = \langle \Phi, \neg\wp\beta, \Delta'\rangle$ for some $\Phi, \alpha, \beta, \Gamma', \Delta'$, where Φ is the (possibly empty) longest common initial segment of Γ and Δ, and $\wp\alpha$ is the leftmost $\wp$-labeled move of Γ that has been shifted to the right ("delayed") when obtaining Δ from Γ. If Φ is illegal, then it must be $\neg\wp$-illegal (otherwise Γ would not be won by $\wp$), which makes Δ also $\neg\wp$-illegal and hence won by $\wp$. Suppose now Φ is legal. Then α must be a legal move of $\wp$ in position Φ (otherwise, again, Γ would be $\wp$-illegal and hence lost by $\wp$). Therefore, since the game is strict, $\neg\wp$ has no legal moves in position Φ, which makes $\langle \Phi, \neg\wp\beta\rangle$ and hence Δ $\neg\wp$-illegal and hence $\wp$-won. □

While all strict games are static, obviously not all static games are strict. The parallel plays of chess that we have seen are examples as a little thought can convince us.

5. The arity of a game

We start this section by introducing some notation. Let e_1 and e_2 be valuations, A a game and $\vec{x}$ a (possibly empty or infinite) set of variables.

- We write $e_1[A] = e_2[A]$ to mean that $\mathbf{Lr}^A_{e_1} = \mathbf{Lr}^A_{e_2}$ and, for every run Γ, $\mathbf{Wn}^A_{e_1}\langle\Gamma\rangle = \mathbf{Wn}^A_{e_2}\langle\Gamma\rangle$ (the meaning of the notation $e[A]$ will be officially defined only in Section 7; right now we do not need to know it).
- We write $e_1 \equiv_{\vec{x}} e_2$ iff $e_1(y) = e_2(y)$ for every variable y with $y \notin \vec{x}$. That is, $e_1 \equiv_{\vec{x}} e_2$ means that e_1 and e_2 agree on all variables except, perhaps, some variables from $\vec{x}$. When $\vec{x}$ consists of one single variable x, we can write $e_1 \equiv_x e_2$ instead of $e_1 \equiv_{\vec{x}} e_2$.

Definition 5.1. Let A be a game.

1. For a variable x, we say that A *does not depend on* x iff $e_1[A]=e_2[A]$ for any two valuations e_1 and e_2 with $e_1 \equiv_x e_2$; otherwise we say that A *depends on* x.
2. Similarly, for a set $\vec{x}$ of variables, we say that A *does not depend on* $\vec{x}$ iff $e_1[A]=e_2[A]$ for any two valuations e_1 and e_2 with $e_1 \equiv_{\vec{x}} e_2$; otherwise we say that A *depends on* $\vec{x}$. This terminology also extends to sequences or tuples of variables, identifying them with the sets of their elements.
3. A is said to be *finitary* iff there is a finite subset $\vec{x}$ of **Variables** such that A does not depend on (**Variables** $-\vec{x}$). When $\vec{x}$ is smallest among such finite subsets, the cardinality n of $\vec{x}$ is called the *arity* of A and A is said to be *n-ary*. Games that are not finitary are said to be *infinitary* and their arity is defined to be ∞ (*infinite*)—the greatest of all arities.
4. A is said to be a *constant game* iff it is 0-ary, i.e. does not depend on **Variables**.

The intuitive meaning of "A does not depend on $\vec{x}$" is that what the values of the variables from $\vec{x}$ are, is totally irrelevant and those values can be fully ignored when talking about what runs of A are legal or won. A finitary game is a game where there is only a finite number of variables whose values matter.

Lemma 5.2. *For any game A and any sets $\vec{x}$ and $\vec{y}$ of variables, if A does not depend on $\vec{x}$ and does not depend on $\vec{y}$, then A does not depend on $\vec{x}\cup\vec{y}$.*

Proof. Suppose A does not depend on $\vec{x}$ and does not depend on $\vec{y}$. Consider any two valuations e_1 and e_2 with $e_1 \equiv_{\vec{x}\cup\vec{y}} e_2$. We want to show that $e_1[A]=e_2[A]$. Let e be a valuation with $e_1 \equiv_{\vec{x}\cup\vec{y}} e \equiv_{\vec{x}\cup\vec{y}} e_2$ that agrees with e_1 on all variables from $\vec{x}-\vec{y}$ and agrees with e_2 on all variables from $\vec{y}-\vec{x}$. Clearly then we have $e \equiv_{\vec{y}} e_1$ and $e \equiv_{\vec{x}} e_2$. Consequently $e[A]=e_1[A]$ and $e[A]=e_2[A]$, whence $e_1[A]=e_2[A]$. □

The above lemma almost immediately implies that if $\vec{x}$ is a finite set of variables, then A does not depend on $\vec{x}$ iff it does not depend on any of the variables from $\vec{x}$. This, however, is not necessarily the case when $\vec{x}$ is infinite. For example, if A is the elementary game with $\mathbf{Wn}_e^A\langle\,\rangle=\top$ iff the set of variables for which e returns 0 is finite, then A does not depend on any particular variable or any finite set of variables, but it depends on **Variables** as well as any cofinite subset of it.

Based on Lemma 5.2, it would also be easy to show that for an n-ary ($n\neq\infty$) game A, a smallest set $\vec{x}$ such that A does not depend on (**Variables** $-\vec{x}$) is unique and its n elements are exactly the variables on which A (individually) depends.

6. Predicates as elementary games

Let us agree to understand classical predicates—in particular, predicates on **Constants**, as subsets of **Valuations** (at which the given predicate is true) rather than, as more commonly done, sets of tuples of constants. Say, "$x>y$" is the predicate

that is true at valuation e iff $e(x)>e(y)$. This way, $P(x)$ and $P(y)$ are understood as different predicates rather than two naming forms of the same predicate. Furthermore, this approach does not restrict predicates, as more commonly done, to ones whose arity is finite: what we call a predicate may depend on the values of infinitely many variables.

The main advantage of this understanding of predicates over the "traditional" understanding is not only that it is more general as includes ∞-ary predicates, but also that it significantly simplifies definitions of propositional connectives and quantifiers as operations on predicates. For example, the classical conjunction and disjunction of predicates P and Q can be defined simply as their intersection and union, respectively. Try to formally define classical disjunction as an operation on predicates understood as sets of tuples of constants rather than sets of valuations, and then try to explain the difference between $P(x)\vee Q(x)$ and $P(x)\vee Q(y)$ to appreciate the difference!

So, henceforth by a *predicate* we will mean a subset of **Valuations**. Equivalently, a predicate P can be understood as a function of the type **Valuations** $\longrightarrow$ **Players**, with $P(e)=\top$ meaning that P is true at e (i.e. $e\in P$), and $P(e)=\bot$ meaning that P is false at e (i.e. $e\notin P$).

Now let us look at elementary games. They all have the same structure: there are no non-empty legal runs. So, the structure component can be ignored, and an elementary game identified with its contents (the **Wn** component). As $\langle\rangle$ is the only legal run of an elementary game, the **Wn** function of such a game is fully defined by giving its values (at different valuations) for $\langle\rangle$. Thus, by fixing the only relevant value $\langle\rangle$ for the run argument, the **Wn** function of an elementary game and, hence, the whole game, becomes a function from **Valuations** to **Players**; in other words, an elementary game becomes a predicate.

With this fact in mind, from now on we will be using the terms "predicate" and "elementary game" as synonyms, identifying an elementary game $(\{\langle\rangle\},\mathbf{Wn}^A)$ with the predicate P that is true at exactly those valuations e for which we have $\mathbf{Wn}^A_e\langle\rangle=\top$.

7. Substitution of variables; instantiation

The standard operation of substitution of variables by terms in predicates naturally extends to games as generalized predicates:

Definition 7.1. Let $\vec{x}=x_1,x_2,\ldots$ be a (finite or infinite) sequence of distinct variables, and $\vec{t}=t_1,t_2,\ldots$ a sequence of (not necessarily distinct) terms of the same length as $\vec{x}$. Then *the result of substituting $\vec{x}$ by $\vec{t}$ in game A*, denoted $A[x_1/t_1,x_2/t_2,\ldots]$ or $A[\vec{x}/\vec{t}]$, is defined as follows. For all e and Γ, $\mathbf{Lr}^{A[\vec{x}/\vec{t}]}_e=\mathbf{Lr}^A_{e'}$ and $\mathbf{Wn}^{A[\vec{x}/\vec{t}]}_e\langle\Gamma\rangle=\mathbf{Wn}^A_{e'}\langle\Gamma\rangle$, where e' is the (unique) valuation with $e\equiv_{\vec{x}}e'$ such that, for every x_i from $\vec{x}$, we have:

- if t_i is a constant, then $e'(x_i)=t_i$;
- if t_i is a variable, then $e'(x_i)=e(t_i)$.

For the following definition, remember that $\mathbf{Variables}=\{v_0,v_1,\ldots\}$.

Definition 7.2. For a valuation e, the *e-instantiation* of game A, denoted $e[A]$, is the game $A[v_0/e(v_0), v_1/e(v_1), \ldots]$. When $B = e[A]$ for some e, we say that B is (simply) an *instantiation*, or *instance* of A.

Instantiation turns every game into a constant game. When we deal with constant games, valuations become irrelevant and, according to our convention, "e" can be omitted in the expressions $\mathbf{Lr}_e^A$, $\mathbf{Lm}\wp_e^A$ and $\mathbf{Wn}_e^A$, where $\mathbf{Lr}_e^A$ can also be written as $\mathbf{LR}^A$. Observe that for every $A, e, \wp$ we then have

$$\mathbf{Lr}_e^A = \mathbf{Lr}^{e[A]}; \quad \mathbf{Lm}\wp_e^A = \mathbf{Lm}\wp^{e[A]}; \quad \mathbf{Wn}_e^A = \mathbf{Wn}^{e[A]}.$$

Hence, instead of phrases such as "legal (won) run of A w.r.t. e", etc., from now on we may just say "legal (won) run of $e[A]$", etc.

One of the possible views of games is to consider only constant games as entities in their own right, and understand non-constant games as functions from valuations to constant games. In particular, a (non-constant) game A can be defined as the function that sends each valuation e to the game $e[A]$.

Convention 7.3. Sometimes it is convenient to fix a certain tuple $\vec{x} = x_1, \ldots, x_n$ of distinct variables for a game A throughout a context. We will refer to $x_1, \ldots, x_n$ as the *attached tuple* of A. In such a case, following a similar notational practice commonly used in the literature for predicates, we can write A as $A(x_1, \ldots, x_n)$ or $A(\vec{x})$. It is important to note here that when doing so, by no means do we mean that $A(x_1, \ldots, x_n)$ is an n-ary game, or that $x_1, \ldots, x_n$ are all the variables on which A may depend, or that A depends on each of those variables. Once A is given with an attached tuple $x_1, \ldots, x_n$, we will write $A(t_1, \ldots, t_n)$ or $A(\vec{t})$ to mean the same as the more clumsy expressions $A[x_1/t_1, \ldots, x_n/t_n]$ or $A[\vec{x}/\vec{t}]$.

Definition 7.4. In a context where A comes with an attached tuple $(x_1, \ldots, x_n)$, a *tuple-instantiation* of $A(x_1, \ldots, x_n)$ is the game $A(c_1, \ldots, c_n)$ for some arbitrary constants $c_1, \ldots, c_n$.

Note that unlike instantiations, tuple-instantiations are not necessarily constant games.

It is not hard to verify that the operation of substitution of variables preserves the strict, elementary, finite-depth, perifinite-depth and finitary properties of games, as well as the unistructural property. It does not however preserve the x-unistructural property. For example, let $F(y)$ be a unary game that is not y-unistructural and that does not depend on x. Since $F(y)$ does not depend on x, it is x-unistructural. But the game $F(x)$, which is nothing but the result of renaming y into x in $F(y)$, is obviously no longer unistructural in x. Finally, we have:

Proposition 7.5. *The operation of substitution of variables is static.*

Proof. Assume A is a static game. Fix a valuation e, and let $\vec{x}$, $\vec{t}$ and e' be exactly as in Definition 7.1. Assume $\mathbf{Wn}_e^{A[\vec{x}/\vec{t}]}\langle\Gamma\rangle = \wp$ and Δ is a $\wp$-delay of Γ. We need to show that then $\mathbf{Wn}_e^{A[\vec{x}/\vec{t}]}\langle\Delta\rangle = \wp$.

By Definition 7.1, we have ($\mathbf{Lr}_e^{A[\vec{x}/\vec{t}]} = \mathbf{Lr}_{e'}^{A}$ and) $\mathbf{Wn}_e^{A[\vec{x}/\vec{t}]}\langle\Gamma\rangle = \mathbf{Wn}_{e'}^{A}\langle\Gamma\rangle$. Consequently, as A is static and Δ is a $\wp$-delay of Γ, we have $\mathbf{Wn}_{e'}^{A}\langle\Delta\rangle = \wp$. This, again by Definition 7.1 with Δ in the role of Γ implies that $\mathbf{Wn}_e^{A[\vec{x}/\vec{t}]}\langle\Delta\rangle = \wp$. □

8. Prefixation; the depth of a game

Each unilegal initial labmove λ of game A can be thought of as an operation that turns A into the game playing which means the same as playing A after λ has been made as an opening move. The following definition formalizes this idea:

Definition 8.1. Let $\langle\lambda\rangle \in \mathbf{LR}^A$. The *$\lambda$-prefixation* $\langle\lambda\rangle A$ of A is defined as follows:
- $\mathbf{Lr}_e^{\langle\lambda\rangle A} = \{\Gamma \mid \langle\lambda,\Gamma\rangle \in \mathbf{Lr}_e^A\}$.
- $\mathbf{Wn}_e^{\langle\lambda\rangle A}\langle\Gamma\rangle = \mathbf{Wn}_e^A\langle\lambda,\Gamma\rangle$.

Obviously if A does not depend on $\vec{x}$, neither does $\langle\lambda\rangle A$, so that this operation preserves the finitary property of games.

The operation of prefixation can be generalized from unilegal initial labmoves, i.e. one-move-long unilegal positions, to all unilegal positions of A by writing $\langle\lambda_1,\ldots,\lambda_n\rangle A$ to mean the same as $\langle\lambda_n\rangle\ldots\langle\lambda_1\rangle A$, identifying $\langle\,\rangle A$ with A. Note the reversal of the order of the λ_i's!

Thus, each Φ that is a unilegal position of game A can be understood as (representing) a new game—in particular, the game $\langle\Phi\rangle A$, playing which means the same as playing A beginning from position Φ. We can refer to $\langle\Phi\rangle A$ as the game *corresponding to* the position Φ of A.

Prefixation will be very handy in visualizing unilegal runs of games as it allows us to think of such a run as a sequence of games (corresponding to the initial segments of the run) rather than a sequence of moves. In particular, where $\Gamma = \langle\lambda_1,\lambda_2,\ldots\rangle$ is a unilegal run of A, the *game-sequence representation* of Γ is the sequence $\langle A_0, A_1, A_2,\ldots\rangle$ of games such that:
- $A_0 = A$ $(= \langle\,\rangle A)$;
- $A_{n+1} = \langle\lambda_{n+1}\rangle A_n$ $(= \langle\lambda_1,\ldots,\lambda_{n+1}\rangle A)$.

Note that $\langle\lambda\rangle A$ is only defined when λ is a unilegal initial labmove of A. This trivially makes the operation of prefixation an elementary operation: since it is undefined for elementary games that have no legal initial labmoves, it cannot violate the elementary property of games.

When defined, obviously prefixation can decrease but never increase the lengths of legal runs, so that this operation is finite-depth and perifinite-depth. It is also a straightforward job to verify that it is strict, unistructural and x-unistructural (any x). Showing that it is static is not hard either:

Proposition 8.2. *The operation of prefixation is static.*

Proof. Assume A is a static game, λ is a unilegal initial labmove of A, and Δ is a $\wp$-delay of Γ. Suppose $\mathbf{Wn}_e^{\langle\lambda\rangle A}\langle\Gamma\rangle = \wp$. This means that $\mathbf{Wn}_e^A\langle\lambda,\Gamma\rangle = \wp$. $\langle\lambda,\Delta\rangle$

is a $\wp$-delay of $\langle \lambda, \Gamma \rangle$ and, since A is static, we have $\mathbf{Wn}_e^A\langle \lambda, \Delta \rangle = \wp$, whence $\mathbf{Wn}_e^{\langle \lambda \rangle A}\langle \Gamma \rangle = \wp$. □

The operation of prefixation allows us to define the following useful relation on games: We say that A is a *descendant* of B—symbolically $A \prec B$—iff there is a non-empty unilegal position Φ of B such that $A = \langle \Phi \rangle B$. $B \succ A$ will mean $A \prec B$.

$\prec$ forms what is called a *well-founded* relation on the set of (all descendants of) perifinite-depth games: as long as A_0 is perifinite-depth, there is no infinite descending chain $A_0 \succ A_1 \succ A_2 \succ \cdots$. This allows us to use transfinite induction in definitions and proofs concerning perifinite-depth games. That is, if the assumption that a certain statement (resp. concept) S is true (resp. defined) for all descendants of a perifinite-depth game A implies that S is also true (resp. defined) for A, we can conclude that S is true (resp. defined) for all perifinite-depth games.

In view of the above remark, we define the *depth* $|A|$ of a constant perifinite-depth game A as the smallest ordinal number that is greater than each of the ordinal numbers from the set $\{|B| \mid B \prec A\}$. The concept of depth extends to all (not-necessarily-constant) perifinite-depth games A by defining $|A|$ as the smallest ordinal number that is greater or equal to each of the ordinal numbers from the set $\{|e[A]| \mid e \in \mathbf{Valuations}\}$. We use the symbol ∞ to denote the depth of infinite- (non-perifinite-) depth games. ∞ is the only depth that is not an ordinal number. For convenience we extend the standard 'greater than' relation $>$ on ordinal numbers to ∞ by stipulating that ∞ is greater than any other depth, and extend the standard successor function to ∞ by stipulating that $\infty + 1 = \infty$.

Thus, the depth of an elementary game is 0, the depth of a non-elementary finite-depth game is among $\{1, 2, \ldots\}$, the depth of a perifinite-depth game that is not finite-depth is a transfinite ordinal number, and every other game has the non-ordinal depth ∞.

9. Negation; trivial games

In Section 2 we agreed to use the symbol $\neg$ to denote the "switching" operation on **Players**. In this section we establish two more meanings of the same symbol: it will as well be used as an operation on runs and an operation on games. In particular, the *negation* $\neg\Gamma$ *of run* Γ will be understood as the result of interchanging the two labels in Γ, i.e., replacing each labmove $\wp\alpha$ by $\neg\wp\alpha$. Note that $\neg\neg\Gamma = \Gamma$ because $\neg\neg\wp = \wp$.

Definition 9.1. The *negation* $\neg A$ of game A is defined as follows:
- $\mathbf{Lr}_e^{\neg A} = \{\Gamma \mid \neg\Gamma \in \mathbf{Lr}_e^A\}$.
- $\mathbf{Wn}_e^{\neg A}\langle \Gamma \rangle = \neg\mathbf{Wn}_e^A\langle \neg\Gamma \rangle$.

Thus, as this was claimed in Section 1, $\neg A$ is the result of switching the roles (moves and wins) of the two players in A. Notice that A and $\neg A$ will always have

the same sets of variables on which they depend, so that this operation preserves the finitary property of games.

Proposition 9.2. $\neg\neg A = A$ (*any game* A).

Proof. $\Gamma \in \mathbf{Lr}_e^{\neg\neg A}$ iff $\neg\Gamma \in \mathbf{Lr}_e^{\neg A}$ iff $\neg\neg\Gamma \in \mathbf{Lr}_e^{A}$ iff $\Gamma \in \mathbf{Lr}_e^{A}$.
Next, $\mathbf{Wn}_e^{\neg\neg A}\langle\Gamma\rangle = \neg\mathbf{Wn}_e^{\neg A}\langle\neg\Gamma\rangle = \neg\neg\mathbf{Wn}_e^{A}\langle\neg\neg\Gamma\rangle = \mathbf{Wn}_e^{A}\langle\Gamma\rangle$. □

The following observations are straightforward but important:

$$\begin{aligned}\mathbf{Lm}\wp_e^{\neg A}\langle\,\rangle &= \mathbf{Lm}\neg\wp_e^{A}\langle\,\rangle,\\ \mathbf{Wn}_e^{\neg A}\langle\,\rangle &= \neg\mathbf{Wn}_e^{A}\langle\,\rangle.\end{aligned} \tag{2}$$

$$\textit{Where } \langle\wp\alpha\rangle \in \mathbf{LR}^{\neg A},\ \textit{we have } \langle\wp\alpha\rangle\neg A = \neg(\langle\neg\wp\alpha\rangle A). \tag{3}$$

It is pretty obvious that the operation of negation preserves all of our structural game properties. We also have:

Proposition 9.3. *The game operation* $\neg$ *is static.*

Proof. Consider a static game A and runs Γ and Δ where Δ is a $\wp$-delay of Γ. Suppose $\mathbf{Wn}_e^{\neg A}\langle\Gamma\rangle = \wp$, i.e. $\mathbf{Wn}_e^{A}\langle\neg\Gamma\rangle = \neg\wp$. Notice that $\neg\Delta$ is a $\neg\wp$-delay of $\neg\Gamma$. Hence, as A is static, we have $\mathbf{Wn}_e^{A}\langle\neg\Delta\rangle = \neg\wp$. Consequently, $\mathbf{Wn}_e^{\neg A}\langle\Delta\rangle = \wp$. □

We have seen three different meanings of the same symbol $\neg$ so far. Hopefully, the reader will not be confused by such abuse of notation. The following definition extends this kind of abuse to the symbols $\bot$ and $\top$, which will be used to denote, besides the two players, two special games as well:

Definition 9.4. *The games* $\top$ and $\bot$ are defined by: $\mathbf{Lr}_e^{\top} = \mathbf{Lr}_e^{\bot} = \{\langle\,\rangle\}$; $\mathbf{Wn}_e^{\top}\langle\,\rangle = \top$; $\mathbf{Wn}_e^{\bot}\langle\,\rangle = \bot$.

Notice that there are only two constant elementary games, and these are exactly the games $\top$ and $\bot$. We call these two games *trivial games*. Of course, $\top = \neg\bot$. As elementary games, $\top$ and $\bot$, that can be considered 0-ary operations on games, are strict, finite-depth, perifinite-depth, x-unistructural, unistructural and static.

10. Choice (additive) operations

Definition 10.1. The *choice* (*additive*) *conjunction* $A_1 \sqcap \cdots \sqcap A_n$ of games $A_1,\ldots,A_n$ ($n \geqslant 2$) is defined as follows:

- $\mathbf{Lr}_e^{A_1\sqcap\cdots\sqcap A_n} = \{\langle\,\rangle\} \cup \{\langle\bot i,\Gamma\rangle \mid i \in \{1,\ldots,n\},\ \Gamma \in \mathbf{Lr}_e^{A_i}\}$.
- $\mathbf{Wn}_e^{A_1\sqcap\cdots\sqcap A_n}\langle\,\rangle = \top$; $\mathbf{Wn}_e^{A_1\sqcap\cdots\sqcap A_n}\langle\bot i,\Gamma\rangle = \mathbf{Wn}_e^{A_i}\langle\Gamma\rangle$ $(i \in \{1,\ldots,n\})$.

Thus, in the initial position of $A_1 \sqcap \cdots \sqcap A_n$, only $\bot$ has legal moves—in particular, n legal (and unilegal) moves: $1,\ldots,n$, corresponding to the choice of one of the conjuncts. After making such a choice i, the game continues according to the rules of A_i. In other words,

$$\langle \bot i\rangle(A_1 \sqcap \cdots \sqcap A_n) = A_i \quad (i \in \{1,\ldots,n\}). \tag{4}$$

The initial choice is not only a privilege, but also an obligation of $\bot$, and if it fails to make such a choice, the game is considered lost by $\bot$. Intuitively, as noted in Section 1, this is so because there was no particular subproblem (A_i) specified by $\bot$ that $\top$ failed to solve.

The above Eq. (4), together with the following equations:

$$\begin{aligned} &\mathbf{Lm}\top_e^{A_1\sqcap\cdots\sqcap A_n}\langle\,\rangle = \emptyset,\\ &\mathbf{Lm}\bot_e^{A_1\sqcap\cdots\sqcap A_n}\langle\,\rangle = \{1,\ldots,n\},\\ &\mathbf{Wn}_e^{A_1\sqcap\cdots\sqcap A_n}\langle\,\rangle = \top \end{aligned} \tag{5}$$

forms what we call an *inductive characterization* of $\sqcap$. In particular, (5) is called the *initial conditions* of the inductive characterization, stating what the legal moves are and who the winner is in the initial (empty) position of a given instance of the game, and condition (4) is called the *inductive clause*, stating to what game $A_1 \sqcap \cdots \sqcap A_n$ evolves after an initial unilegal move is made. Eqs. (2) and (3) from the previous section are another example of an inductive characterization—that of the operation $\neg$.

While any number $n \geqslant 2$ of conjuncts is officially allowed,[6] in order to keep things simpler, we will sometimes pretend that the only version of $\sqcap$ we have is binary. The same applies to the operations $\sqcup, \wedge, \vee$ defined later. Everything important that we say or prove for the binary version of the operation, usually silently extends to the n-ary version for any $n \geqslant 2$. It should be noted however that, strictly speaking, an n-ary conjunction generally cannot be directly expressed in terms of the binary conjunction. For example, even though the two games $A_1 \sqcap A_2 \sqcap A_3$ and $(A_1 \sqcap A_2) \sqcap A_3$ are *equivalent* (in the sense explained in Part II), they are not *equal* for, after all, the depth of the latter is at least 2, while the former can be a game of depth 1. Hence, having n instead of 2 in the official definition of $\sqcap$ is not quite redundant.

The *choice* (*additive*) *disjunction* $A_1 \sqcup \cdots \sqcup A_n$ of games $A_1,\ldots,A_n$ ($n \geqslant 2$) is defined by

$$A_1 \sqcup \cdots \sqcup A_n =_{\text{def}} \neg(\neg A_1 \sqcap \cdots \sqcap \neg A_n).$$

A verification of the following fact is straightforward:

[6] Of course, the requirement $n \geqslant 2$ could be painlessly relaxed to $n \geqslant 0$; the only reason why we still prefer to officially require that $n \geqslant 2$ is purely notational convenience: with $n < 2$, the expression $A_1 \sqcap \cdots \sqcap A_n$ would look a little funny, and we are reluctant to switch to some more flexible but less habitual notation such as, say, $\sqcap(A_1,\ldots,A_n)$ instead of $A_1 \sqcap \cdots \sqcap A_n$.

Proposition 10.2. *For any games* $A_1,\ldots,A_n$ $(n\geqslant 2)$ *we have*:
- $\mathbf{Lr}_e^{A_1\sqcup\cdots\sqcup A_n}=\{\langle\,\rangle\}\cup\{\langle\top i,\Gamma\rangle \mid i\in\{1,\ldots,n\},\ \Gamma\in\mathbf{Lr}_e^{A_i}\}$.
- $\mathbf{Wn}_e^{A_1\sqcup\cdots\sqcup A_n}\langle\,\rangle=\bot$; $\mathbf{Wn}_e^{A_1\sqcup\cdots\sqcup A_n}\langle\top i,\Gamma\rangle=\mathbf{Wn}_e^{A_i}\langle\Gamma\rangle$ $(i\in\{1,\ldots,n\})$.

Thus, in a choice disjunction it is $\top$ rather than $\bot$ who makes the initial choice of a component. We could have used the statement of the above proposition as a direct definition of $\sqcup$. As we see, $\sqcup$ can be defined in a way fully symmetric to $\sqcap$, by just interchanging "$\top$" and "$\bot$" in the body of the definition. Alternatively, we could have introduced $\sqcup$ as the basic choice operator and then defined $\sqcap$ by

$$A_1\sqcap\cdots\sqcap A_n=\neg(\neg A_1\sqcup\cdots\sqcup\neg A_n).$$

As an immediate corollary of Proposition 9.2, we have

$$\neg(A_1\sqcup\cdots\sqcup A_n)=\neg A_1\sqcap\cdots\sqcap\neg A_n,$$
$$\neg(A_1\sqcap\cdots\sqcap A_n)=\neg A_1\sqcup\cdots\sqcup\neg A_n.$$

Next, just as in classical logic where conjunction and disjunction have their "big brothers"—universal quantifier and existential quantifier, here, too, we have quantifier-style versions of choice operations:

Definition 10.3. The *choice* (*additive*) *universal quantification* $\bigsqcap xA(x)$ of game $A(x)$ (remember Convention 7.3) is defined as follows:
- $\mathbf{Lr}_e^{\bigsqcap xA(x)}=\{\langle\,\rangle\}\cup\{\langle\bot c,\Gamma\rangle \mid c\in\mathbf{Constants},\ \Gamma\in\mathbf{Lr}_e^{A(c)}\}$.
- $\mathbf{Wn}_e^{\bigsqcap xA(x)}\langle\,\rangle=\top$; $\mathbf{Wn}_e^{\bigsqcap xA(x)}\langle\bot c,\Gamma\rangle=\mathbf{Wn}_e^{A(c)}\langle\Gamma\rangle$.

Thus, the first (uni)legal move in $\bigsqcap xA(x)$ consists in choosing, by $\bot$, one of the elements c of **Constants**, after which the play continues according to the rules of $A(c)$, which means nothing but that

$$\langle\bot c\rangle\bigsqcap xA(x)=A(c). \tag{6}$$

If this first move was not made, then $\top$ is considered to be the winner.

The *choice* (*additive*) *existential quantification* $\bigsqcup xA(x)$ of game $A(x)$, officially defined by

$$\bigsqcup xA(x)=_{\text{def}}\neg\bigsqcap x\neg A(x),$$

is similar to $\bigsqcap xA(x)$, with the difference that there it is $\top$ rather than $\bot$ who can and has to make the initial choice of c.

Just as in the case of $\sqcup$ (Proposition 10.2), we can obtain a direct definition of $\bigsqcup xA(x)$ from the definition of $\bigsqcap xA(x)$ by simply interchanging $\top$ and $\bot$:

Proposition 10.4. *For any game* $A(x)$, *we have*
- $\mathbf{Lr}_e^{\bigsqcup xA(x)}=\{\langle\,\rangle\}\cup\{\langle\top c,\Gamma\rangle \mid c\in\mathbf{Constants},\ \Gamma\in\mathbf{Lr}_e^{A(c)}\}$.
- $\mathbf{Wn}_e^{\bigsqcup xA(x)}\langle\,\rangle=\bot$; $\mathbf{Wn}_e^{\bigsqcup xA(x)}\langle\top c,\Gamma\rangle=\mathbf{Wn}_e^{A(c)}\langle\Gamma\rangle$.

As we may guess, we always have

$$\neg\bigsqcup xA(x) = \bigsqcap x\neg A(x),$$
$$\neg\bigsqcap xA(x) = \bigsqcup x\neg A(x).$$

Equations in the style of (4) and (6) are very handy in visualizing unilegal runs through game-sequence representation (see Section 8), as demonstrated below:

Example 10.5. Consider the halting problem $H = \bigsqcap x \bigsqcap y(Halts(x, y) \sqcup \neg Halts(x, y))$ discussed in Section 1, and the unilegal run $\langle \bot 99, \bot 0, \top 1\rangle$ of it. This run has four initial segments, and correspondingly it can be represented as the following sequence of four games:

0. $\bigsqcap x \bigsqcap y(Halts(x, y) \sqcup \neg Halts(x, y))$ i.e. $\langle\rangle H$.
1. $\bigsqcap y(Halts(99, y) \sqcup \neg Halts(99, y))$ i.e. $\langle \bot 99\rangle H$.
2. $Halts(99, 0) \sqcup \neg Halts(99, 0)$ i.e. $\langle \bot 99, \bot 0\rangle H$.
3. $Halts(99, 0)$ i.e. $\langle \bot 99, \bot 0, \top 1\rangle H$.

Then the run is won by $\top$ iff $Halts(99, 0)$ is true. Generally, when a finite unilegal run is represented as a sequence $A_0, \ldots, A_n$ of games in the above style, the run is won (w.r.t. e) by the very player $\wp$ for which we have $\mathbf{Wn}_e^{A_n}\langle\rangle = \wp$.

It is easy to see that if A does not depend on $\vec{x}$ and B does not depend on $\vec{y}$, then $A \sqcap B$ and $A \sqcup B$ do not depend on $\vec{x} \cap \vec{y}$. Taking into account that the intersection of two cofinite sets is always cofinite, it is obvious that $\sqcap$ and $\sqcup$ preserve the finitary property of games. As for the additive quantifiers, one can see that if $A(x)$ does not depend on $\vec{y}$, then $\bigsqcap xA(x)$ and $\bigsqcup xA(x)$ do not depend on $\vec{y} \cup \{x\}$, so that $\bigsqcap$ and $\bigsqcup$ are finitary operations as well.

All choice operations are strict, as in the empty position only one of the players has legal moves, and in any other legal position the set of legal moves by either player is the same as that of the corresponding position of the corresponding (sub)game. The choice operations are also finite- and perifinite-depth, as they can only increase the depth of their (deepest) argument by 1. One can also verify that these operations are x-unistructural (any variable x) and unistructural. The choice operations are not elementary though: in fact, their application to whatever games obviously always produces a non-elementary game. Finally, we have:

Proposition 10.6. *The operations* $\sqcap, \sqcup, \bigsqcap, \bigsqcup$ *are static.*

Proof. It would be sufficient to demonstrate how to prove this for (the binary) $\sqcap$, the other cases being similar. Assume A_1 and A_2 are static games, $\mathbf{Wn}_e^{A_1 \sqcap A_2}\langle\Gamma\rangle = \wp$ and Δ is a $\wp$-delay of Γ. We need to show that $\mathbf{Wn}_e^{A_1 \sqcap A_2}\langle\Delta\rangle = \wp$.

If $\Gamma = \langle\rangle$, then $\Delta = \Gamma$ and we are done. So, assume $\Gamma \neq \langle\rangle$.

If the first labmove of Γ is an illegal initial labmove of $e[A_1 \sqcap A_2]$, this move should be $\neg\wp$-labeled (otherwise Γ would not be won by $\wp$). Then, by Lemma 4.5, Δ has the same first labmove, which makes it a $\neg\wp$-illegal and hence $\wp$-won run of $e[A_1 \sqcap A_2]$.

Now assume that the first labmove of Γ is a legal initial labmove of $e[A_1 \sqcap A_2]$. This means that Γ looks like $\langle \bot i, \Gamma' \rangle$ with $i \in \{1,2\}$, and we have

$$\mathbf{Wn}_e^{A_i}\langle \Gamma' \rangle = \wp. \tag{7}$$

There are two cases to consider:

Case 1: $\wp = \top$. Then, in view of Lemma 4.5, Δ looks like $\langle \bot i, \Delta' \rangle$. This implies that Δ' is a $\wp$-delay of Γ'. Then (7), together with the assumption that A_i is static, implies that $\mathbf{Wn}_e^{A_i}\langle \Delta' \rangle = \wp$. Consequently, $\mathbf{Wn}_e^{A_1 \sqcap A_2}\langle \Delta \rangle = \wp$.

Case 2: $\wp = \bot$. If Δ has the same first labmove $\bot i$ as Γ has, we can use the same argument as above to conclude that $\mathbf{Wn}_e^{A_1 \sqcap A_2}\langle \Delta \rangle = \wp$. Otherwise, if $\bot i$ is not the first labmove of Δ, in view of clause 1 of Definition 4.1, we must have that the first move of Δ is $\top$-labeled. But then it is an illegal initial labmove of $e[A_1 \sqcap A_2]$, and we again have $\mathbf{Wn}_e^{A_1 \sqcap A_2}\langle \Delta \rangle = \bot = \wp$. □

11. Blind operations

Definition 11.1. Suppose game $A(x)$ is unistructural in x. The *blind universal quantification* $\forall x A(x)$ of $A(x)$ is defined by

- $\mathbf{Lr}_e^{\forall x A(x)} = \mathbf{Lr}_e^{A(x)}$.
- $\mathbf{Wn}_e^{\forall x A(x)}\langle \Gamma \rangle = \top$ iff, for every constant c, $\mathbf{Wn}_e^{A(c)} = \top$.

Note that the operation $\forall x$ is only defined when its argument is x-unistructural. This operation preserves the finitary property of games for a similar reason as $\sqcap$ does.

The *blind existential quantification* $\exists x A(x)$ of an x-unistructural game $A(x)$ is defined by

$$\exists x A(x) =_{\text{def}} \neg \forall x \neg A(x).$$

The following easy-to-verify fact can be used as a direct definition of $\exists$, obtained by replacing $\top$ with $\bot$ in Definition 11.1:

Proposition 11.2. *For any x-unistructural game $A(x)$, we have*:

- $\mathbf{Lr}_e^{\exists x A(x)} = \mathbf{Lr}_e^{A(x)}$.
- $\mathbf{Wn}_e^{\exists x A(x)}\langle \Gamma \rangle = \bot$ *iff, for every constant* c, $\mathbf{Wn}_e^{A(c)} = \bot$.

As noted in Section 1, $\forall$ and $\exists$ can be thought of as versions of $\sqcap$ and $\sqcup$ where the initial legal move remains invisible, so that the game is played "blindly". Notice that, unlike any other game operations we have introduced, the blind quantifiers do not change the structure of their arguments, so that $\mathbf{Lr}_e^{\forall x A} = \mathbf{Lr}_e^{\exists x A} = \mathbf{Lr}_e^{A}$, and the only difference between $\forall x A(x)$ and $\exists x A(x)$ is that, in $\forall x A(x)$, $\top$ has to win $A(x)$ for every possible value of x, while in $\exists x A(x)$ winning $A(x)$ for just one value of x is sufficient.

One can easily verify that, for any unilegal initial labmove λ of $\forall x A(x)$, we have

$$\langle \lambda \rangle \forall x A(x) = \forall x \langle \lambda \rangle A(x). \tag{8}$$

Obviously, in the above equation, $\langle\lambda\rangle$ can be replaced by any unilegal position Φ of $A(x)$. The situation with $\exists$ is, of course, similar. Hence, when runs are represented as sequences of games in the style of Example 10.5, the $\forall,\exists$-structure of those games (unlike their $\sqcap,\sqcup,\bigsqcap,\bigsqcup$-structure) remains unchanged as demonstrated below.

Example 11.3. The unilegal run $\langle\bot 5,\top 2\rangle$ of game $\forall x\bigsqcap y(A(x,y)\sqcup B(x,y,z))$ is represented by the following sequence of games:

0. $\forall x\bigsqcap y(A(x,y)\sqcup B(x,y,z))$.
1. $\forall x(A(x,5)\sqcup B(x,5,z))$.
2. $\forall xB(x,5,z)$.

Note how $\forall x$ was retained throughout the sequence.

Eq. (8), together with the following initial conditions, forms an inductive characterization of $\forall$:

$$\begin{aligned}&\mathbf{Lm}\wp_e^{\forall xA(x)}\langle\rangle=\mathbf{Lm}\wp_e^{A(x)}\langle\rangle,\\&\mathbf{Wn}_e^{\forall xA(x)}\langle\rangle=\top\text{ iff, for all }c,\ \mathbf{Wn}_e^{A(c)}\langle\rangle=\top.\end{aligned}\tag{9}$$

As the blind quantifiers do not affect the **Lr** component of their arguments, they preserve all the structural properties of games. And we also have:

Proposition 11.4. *The operations $\forall x$ and $\exists x$ are static.*

Proof. As $\exists$ is expressible in terms of $\forall$ and $\neg$ and $\neg$ is static, it is sufficient to prove the proposition only for $\forall$. Assume $A(x)$ is a static game unistructural in x, $\mathbf{Wn}_e^{\forall xA(x)}\langle\Gamma\rangle=\wp$ and Δ is a $\wp$-delay of Γ. We need to verify that $\mathbf{Wn}_e^{\forall xA(x)}\langle\Delta\rangle=\wp$. If $\Gamma\notin\mathbf{Lr}_e^{\forall xA(x)}$, then Γ must be a $\neg\wp$-illegal run of $e[\forall xA(x)]$, hence it is also a $\neg\wp$-illegal run of $e[A(x)]$, whence, by Lemma 4.7.2, Δ is a $\neg\wp$-illegal run of $e[A(x)]$, whence Δ is a $\neg\wp$-illegal run of $e[\forall xA(x)]$ and hence a $\wp$-won run of that game. So, for the rest of the proof, assume that $\Gamma\in\mathbf{Lr}_e^{\forall xA(x)}$. Since $A(x)$ is static, by Proposition 7.5, so is $A(c)$. Therefore, as Δ is a $\wp$-delay of Γ, we have:

$$\textit{For every constant } c,\ \textit{if } \mathbf{Wn}_e^{A(c)}\langle\Gamma\rangle=\wp,\ \textit{then } \mathbf{Wn}_e^{A(c)}\langle\Delta\rangle=\wp.\tag{10}$$

Suppose $\wp=\top$. Then $\mathbf{Wn}_e^{A(c)}\langle\Gamma\rangle=\top$ for every c, whence, by (10), we also have $\mathbf{Wn}_e^{A(c)}\langle\Delta\rangle=\top$ for every c, which means that $\mathbf{Wn}_e^{\forall xA(x)}\langle\Delta\rangle=\top=\wp$.

And if $\wp=\bot$, then there is a constant c for which $\mathbf{Wn}_e^{A(c)}\langle\Gamma\rangle=\bot$. By (10), this implies $\mathbf{Wn}_e^{A(c)}\langle\Delta\rangle=\bot$, whence $\mathbf{Wn}_e^{\forall xA(x)}\langle\Delta\rangle=\bot=\wp$. □

12. Parallel (multiplicative) operations

The following notational convention will be frequently used in this and later sections:

Convention 12.1. Let s be any string over the alphabet $\{0-9,.,:\}$. Then Γ^s will stand for the result of first removing from Γ all the (labeled) moves except those whose

prefix is s, and then deleting the prefix s in the remaining labeled moves, preserving all the labels.

For example, $\langle \top 11.3.3, \bot 1.2, \bot 00, \top 1.1.0\rangle^{1.} = \langle \bot 2, \top 1.0\rangle$ (the first and the third moves removed, and then the prefix "1." deleted in the second and the fourth moves).

Definition 12.2. The *parallel* (*multiplicative*) *conjunction* $A_1 \wedge \cdots \wedge A_n$ of games $A_1, \ldots, A_n$ $(n \geqslant 2)$ is defined as follows. Whenever Φ is a legal position and Γ a legal run of $e[A_1 \wedge \ldots \wedge A_n]$ (see Remark 2.2), we have

- $\mathbf{Lm}\wp_e^{A_1\wedge\cdots\wedge A_n}\langle\Phi\rangle = \{i.\alpha \mid i \in \{1,\ldots,n\},\ \alpha \in \mathbf{Lm}\wp_e^{A_i}\langle\Phi^{i.}\rangle\}$.
- $\mathbf{Wn}_e^{A_1\wedge\cdots\wedge A_n}\langle\Gamma\rangle = \top$ iff, for each $1 \leqslant i \leqslant n$, $\mathbf{Wn}_e^{A_i}\langle\Gamma^{i.}\rangle = \top$.

As this was explained in Section 1, playing $A_1 \wedge \cdots \wedge A_n$ means playing the n games $A_1, \ldots, A_n$ in parallel, and $\top$ needs to win all of these games to be the winner in $A_1 \wedge \cdots \wedge A_n$. To indicate that the move α is made in conjunct #i, the player has to prefix α with "$i.$". Any move that does not have one of such n possible prefixes will be considered illegal. Every legal run Γ of $A_1 \wedge \cdots \wedge A_n$ will thus consist of n disjoint subruns: $\Gamma_1, \ldots, \Gamma_n$—with each Γ_i consisting of the $i.$-prefixed labmoves of Γ—interleaved in an arbitrary way, where, for each i, the result of deleting the prefix "$i.$" in Γ_i, i.e. the run $\Gamma^{i.}$, is a legal run of A_i.

The *parallel* (*multiplicative*) *disjunction* $A_1 \vee \cdots \vee A_n$ of games $A_1, \ldots, A_n$ $(n \geqslant 2)$ is defined by

$$A_1 \vee \cdots \vee A_n =_{\text{def}} \neg(\neg A_1 \wedge \cdots \wedge \neg A_n).$$

Obviously, we can obtain a direct definition of parallel disjunction by replacing $\top$ with $\bot$ in the definition of parallel conjunction:

Proposition 12.3. *For any games* $A_1, \ldots, A_n$ $(n \geqslant 2)$, *whenever* Φ *is a legal position and* Γ *a legal run of* $e[A_1 \vee \cdots \vee A_n]$, *we have*:

- $\mathbf{Lm}\wp_e^{A_1\vee\cdots\vee A_n}\langle\Phi\rangle = \{i.\alpha \mid i \in \{1,\ldots,n\},\ \alpha \in \mathbf{Lm}\wp_e^{A_i}\langle\Phi^{i.}\rangle\}$.
- $\mathbf{Wn}_e^{A_1\vee\cdots\vee A_n}\langle\Gamma\rangle = \bot$ *iff*, *for each* $1 \leqslant i \leqslant n$, $\mathbf{Wn}_e^{A_i}\langle\Gamma^{i.}\rangle = \bot$.

As an immediate consequence of Proposition 9.2, we have

$$\neg(A_1 \wedge \cdots \wedge A_n) = \neg A_1 \vee \cdots \vee \neg A_n,$$
$$\neg(A_1 \vee \cdots \vee A_n) = \neg A_1 \wedge \cdots \wedge \neg A_n.$$

The *strong* (*linear*) *reduction* $A \to B$ of game B (called the *consequent*) to game A (called the *antecedent*) is defined by

$$A \to B =_{\text{def}} (\neg A) \vee B.$$

Alternative names for $\to$ are *parallel implication* and *multiplicative implication*.

For $\wp i.\alpha$ $(1 \leqslant i \leqslant n)$ to be a legal initial labmove of the e-instantiation of $A_1 \wedge \cdots \wedge A_n$ or $A_1 \vee \cdots \vee A_n$, $\wp\alpha$ needs to be a legal initial labmove of $e[A_i]$. So, $\wp i.\alpha$ is a unilegal

initial labmove of $A_1 \wedge \cdots \wedge A_n$ iff α is a unilegal initial labmove of A_i, and in that case we obviously have

$$\langle \wp i.\alpha\rangle(A_1 \wedge \cdots \wedge A_n) = A_1 \wedge \cdots \wedge A_{i-1} \wedge \langle \wp\alpha\rangle A_i \wedge A_{i+1} \wedge \cdots \wedge A_n,$$
$$\langle \wp i.\alpha\rangle(A_1 \vee \cdots \vee A_n) = A_1 \vee \cdots \vee A_{i-1} \vee \langle \wp\alpha\rangle A_i \vee A_{i+1} \vee \cdots \vee A_n.$$

With $\rightarrow$ the situation is just slightly different. For $\wp 2.\alpha$ to be a legal initial labmove of $e[A \rightarrow B]$, as in the case of (the binary) $\wedge$ and $\vee$, $\wp\alpha$ needs to be a legal initial labmove of $e[B]$, in which case, as long as the move is unilegal, we have

$$\langle \wp 2.\alpha\rangle(A \rightarrow B) = A \rightarrow \langle \wp\alpha\rangle B.$$

Remembering statements (2) and (3) from Section 9, however, we can see that for $\wp 1.\alpha$ to be a legal initial labmove of $e[A \rightarrow B]$, $\neg\wp\alpha$ (rather than $\wp\alpha$) needs to be a legal initial labmove of $e[A]$. As long as such a labmove $\wp 1.\alpha$ is unilegal, we have

$$\langle \wp 1.\alpha\rangle(A \rightarrow B) = (\langle \neg\wp\alpha\rangle A) \rightarrow B.$$

Example 12.4. Let us see all this in work. Remember the problem of strongly reducing the acceptance problem to the halting problem from Section 1:

$$\sqcap x \sqcap y(\mathit{Halts}(x, y) \sqcup \neg \mathit{Halts}(x, y)) \rightarrow \sqcap x \sqcap y(\mathit{Accepts}(x, y) \sqcup \neg \mathit{Accepts}(x, y)).$$

Consider the run $\langle \bot 2.99, \bot 2.0, \top 1.99, \top 1.0, \bot 1.1, \top 2.2\rangle$. It follows the scenario described in Section 1: $\bot$ specified Turing machine (encoded by) 99 and input 0 for it in the consequent; then $\top$ specified the same machine and same input in the antecedent; to this $\bot$ replied by "left", thus claiming that machine 99 halts on input 0; finally, $\top$ selected "right" in the consequent, thus claiming that machine 99 does not accept input 0 (perhaps $\top$ came to this conclusion after simulating machine 99 on input 0).

The game-sequence representation of this run is:

0. $\sqcap x \sqcap y(\mathit{Halts}(x, y) \sqcup \neg \mathit{Halts}(x, y)) \rightarrow \sqcap x \sqcap y(\mathit{Accepts}(x, y) \sqcup \neg \mathit{Accepts}(x, y))$.
1. $\sqcap x \sqcap y(\mathit{Halts}(x, y) \sqcup \neg \mathit{Halts}(x, y)) \rightarrow \sqcap y(\mathit{Accepts}(99, y) \sqcup \neg \mathit{Accepts}(99, y))$.
2. $\sqcap x \sqcap y(\mathit{Halts}(x, y) \sqcup \neg \mathit{Halts}(x, y)) \rightarrow (\mathit{Accepts}(99, 0) \sqcup \neg \mathit{Accepts}(99, 0))$.
3. $\sqcap y(\mathit{Halts}(99, y) \sqcup \neg \mathit{Halts}(99, y)) \rightarrow (\mathit{Accepts}(99, 0) \sqcup \neg \mathit{Accepts}(99, 0))$.
4. $(\mathit{Halts}(99, 0) \sqcup \neg \mathit{Halts}(99, 0)) \rightarrow (\mathit{Accepts}(99, 0) \sqcup \neg \mathit{Accepts}(99, 0))$.
5. $\mathit{Halts}(99, 0) \rightarrow (\mathit{Accepts}(99, 0) \sqcup \neg \mathit{Accepts}(99, 0))$.
6. $\mathit{Halts}(99, 0) \rightarrow \neg \mathit{Accepts}(99, 0)$.

The run is then won by $\top$ iff machine 99 really does not accept input 0 (the consequent is true) or it does not really halt on that input (the antecedent is false). In other words, $\top$ wins the run iff $\langle\rangle$ is a won run of the last game of the above sequence of games.

Notice that the operator $\rightarrow$ was retained in all games of the sequence. Generally, as this was the case with the blind operations (and, in fact, is the case with $\neg$ as well), the $\wedge, \vee, \rightarrow$-structure of the game does not change throughout a run.

Our parallel operations preserve the finitary property of games for similar reasons as $\sqcap$ and $\sqcup$ do, and it is also pretty obvious that they preserve the elementary, finite-depth,

perifinite-depth, x-unistructural and unistructural properties. These operations, however, are not strict. For example, if A and B are (strict) games whose only non-empty legal runs are $\langle\top 0\rangle$ and $\langle\bot 0\rangle$, respectively, then, in the initial position $\langle\rangle$ of $A\wedge B$, $\top$ has the legal move 1.0 and $\bot$ has the legal move 2.0. Finally, as all the other game operations introduced in this paper, the parallel operations are static. To prove this, we need the following lemma:

Lemma 12.5. *Assume $A_1,\ldots,A_n$ are static games, Δ is a $\wp$-delay of Γ, and Δ is a $\wp$-illegal run of $e[A_1\wedge\cdots\wedge A_n]$. Then Γ is also a $\wp$-illegal run of $e[A_1\wedge\cdots\wedge A_n]$.*

Proof. We will prove this lemma by induction on the length of the shortest illegal initial segment of Δ. To save space and without loss of generality, we assume that $n=2$. For the same reason, we fix an arbitrary valuation e throughout the context and no longer explicitly mention it.

Assume the conditions of the lemma. We want to show that Γ is a $\wp$-illegal run of $A_1\wedge A_2$. Let $\langle\Psi,\wp\alpha\rangle$ be the shortest ($\wp$-) illegal initial segment of Δ. Let $\langle\Phi,\wp\alpha\rangle$ be the shortest initial segment of Γ containing all the $\wp$-labeled moves[7] of $\langle\Psi,\wp\alpha\rangle$. If Φ is a $\wp$-illegal position of $A_1\wedge A_2$, then so is Γ and we are done. Therefore, for the rest of the proof, we assume that

$$\Phi \text{ is not a } \wp\text{-illegal position of } A_1\wedge A_2. \tag{11}$$

Let Θ be the sequence of those $\neg\wp$-labeled moves of Ψ that are not in Φ. Obviously,

$$\langle\Psi,\wp\alpha\rangle \text{ is a } \wp\text{-delay of } \langle\Phi,\wp\alpha,\Theta\rangle. \tag{12}$$

We also claim that

$$\Phi \text{ is a legal position of } A_1\wedge A_2. \tag{13}$$

Indeed, suppose this was not the case. Then, by (11), Φ should be $\neg\wp$-illegal. This would make Γ a $\neg\wp$-illegal run of $A_1\wedge A_2$ with Φ as an illegal initial segment which is shorter than $\langle\Psi,\wp\alpha\rangle$. Then, by the induction hypothesis, any run for which Γ is a $\neg\wp$-delay, would be $\neg\wp$-illegal. But by Lemma 4.6, Γ is a $\neg\wp$-delay of Δ. So, Δ would be $\neg\wp$-illegal, which is a contradiction because, according to our assumption, Δ is $\wp$-illegal.

We are continuing our proof. There are two possible reasons to why $\langle\Psi,\wp\alpha\rangle$ is an illegal (while Ψ being legal) position of $A_1\wedge A_2$:

Reason 1: α does not have the form $1.\beta$ or $2.\beta$. Then, in view of (13), $\langle\Phi,\wp\alpha\rangle$ is a $\wp$-illegal position of $A_1\wedge A_2$. As $\langle\Phi,\wp\alpha\rangle$ happens to be an initial segment of Γ, the latter then is a $\wp$-illegal run of $A_1\wedge A_2$.

Reason 2: $\alpha=i.\beta$ ($i\in\{1,2\}$) and $\beta\notin\mathbf{Lm}\wp^{A_i}\langle\Psi^{i.}\rangle$. That is, $\langle\Psi,\wp\alpha\rangle^{i.}$ is a $\wp$-illegal position of A_i. (12) obviously implies that $\langle\Psi,\wp\alpha\rangle^{i.}$ is a $\wp$-delay of $\langle\Phi,\wp\alpha,\Theta\rangle^{i.}$. Therefore, since A_i is static, Lemma 4.7.1 yields that $\langle\Phi,\wp\alpha,\Theta\rangle^{i.}$ is a $\wp$-illegal position

[7] In this context, different occurrences of the same labmove count as different labmoves. So, a more accurate phrasing would be "as many $\wp$-labeled moves as..." instead "all the $\wp$-labeled moves of ...".

of A_i. Notice that $\langle \Phi, \wp\alpha, \Theta\rangle^{i.} = \langle \Phi^{i.}, \wp\beta, \Theta^{i.}\rangle$. A $\wp$-illegal position will remain $\wp$-illegal after removing a block of $\neg\wp$-labeled moves (in particular, $\Theta^{i.}$) at the end of it. Hence, $\langle \Phi^{i.}, \wp\beta\rangle$ is a $\wp$-illegal position of A_i. In view of (13), this implies that $\alpha = i.\beta \notin \mathbf{Lm}\wp^{A_1\wedge A_2}\langle\Phi\rangle$, so that $\langle \Phi, \wp\alpha\rangle$ is a $\wp$-illegal position of $A_1 \wedge A_2$, and then so is Γ because $\langle \Phi, \wp\alpha\rangle$ is an initial segment of it. □

Proposition 12.6. *The operations* $\wedge, \vee, \rightarrow$ *are static.*

Proof. As $\vee$ and $\rightarrow$ are expressible in terms of $\wedge$ and $\neg$ and we know that $\neg$ is static, it will be sufficient to verify that $\wedge$ is static. Also, considering the binary version of $\wedge$ should be sufficient as generalizing from the binary to an n-ary case is a straightforward job.

Assume that A_1 and A_2 are static games, $\mathbf{Wn}_e^{A_1\wedge A_2}\langle\Gamma\rangle = \wp$ and Δ is a $\wp$-delay of Γ. Our goal is to show that $\mathbf{Wn}_e^{A_1\wedge A_2}\langle\Delta\rangle = \wp$.

If Δ is a $\neg\wp$-illegal run of $e[A_1 \wedge A_2]$, then it is won by $\wp$ and we are done. So, assume that Δ is not $\neg\wp$-illegal. Then, by Lemmas 4.6 and 12.5, Γ is not $\neg\wp$-illegal, either. Γ also cannot be $\wp$-illegal, for otherwise it would not be won by $\wp$. Consequently, Δ cannot be $\wp$-illegal either, for otherwise, by Lemma 12.5, Γ would be $\wp$-illegal. Thus, we have narrowed down our considerations to the case when both Γ and Δ are legal runs of $e[A_1 \wedge A_2]$.

$\mathbf{Wn}_e^{A_1\wedge A_2}\langle\Gamma\rangle = \wp$, together with $\Gamma \in \mathbf{Lr}_e^{A_1\wedge A_2}$, implies that for both of the $i \in \{1,2\}$ (if $\wp = \top$) or one of the $i \in \{1,2\}$ (if $\wp = \bot$), we have that $\Gamma^{i.}$ is a $\wp$-won run of $e[A_i]$. Taking into account that $\Delta^{i.}$ is obviously a $\wp$-delay of $\Gamma^{i.}$ and that A_1 and A_2 are static, the above, in turn, implies that for both of the $i \in \{1,2\}$ (if $\wp = \top$) or one of the $i \in \{1,2\}$ (if $\wp = \bot$), $\Delta^{i.}$ is a $\wp$-won run of $e[A_i]$, which, taking into account that $\Delta \in \mathbf{Lr}_e^{A_1\wedge A_2}$, means nothing but that Δ is a $\wp$-won run of $e[A_1 \wedge A_2]$. □

13. Branching (exponential) operations

In Section 1 we characterized $!A$ as the game where, in order to win, $\top$ has to successfully play as many "copies" of A as $\bot$ requests. Such an abstract explanation might have suggested that $!A$ acts as the "infinite multiplicative conjunction" $A \wedge A \wedge A \wedge \cdots$. This guess would be close to the truth but not exactly right, because $!A$ is somewhat stronger (easier for $\bot$ to win) than the infinite multiplicative conjunction of A's.

Both $A \wedge A \wedge A \wedge \cdots$ and $!A$ can be thought of as games where $\bot$ is allowed to restart A an unlimited number of times without terminating the already-in-progress runs of A, creating, this way, more and more parallel plays of A with the possibility to try different strategies in them and become the winner as long as one of those strategies succeeds. What makes $!A$ stronger than $A \wedge A \wedge A \wedge \cdots$, however, is that, in $!A$, $\bot$ does not have to really restart A from the very beginning every time it "restarts" it, but rather it can select to continue it from any of the previous positions, thus depriving $\top$ of the possibility to reconsider the moves it has already made. A little more precisely, at any time $\bot$ is allowed to replicate (backup) any of the currently reached parallel positions of A before further modifying it, which gives it the possibility to come back later and continue playing A from the backed-up position. This way, we get a tree of labmoves

(each branch spelling a legal run of A) rather than just multiple parallel sequences of labmoves. Then $A \wedge A \wedge A \cdots$ can be thought of as a weak version of $!A$ where only the empty position of A can be replicated, that is, where branching in the tree can only happen at its root.

Blass [4,5] was the first to introduce a game operation of this kind, which he called the *repetition* operation R and which can be considered a direct precursor of our $!$. The general motivations and ideas behind both of these operations are similar. Technically, however, they are rather different. The repetition operation is strict while $!$ is free, which makes the latter more natural for the same reasons that make our free multiplicatives more natural than the strict multiplicatives of [5] also adopted in [12]. The formal definition given by Blass for R has no explicit mention of trees or replication. Instead, Blass defines $R(A)$ as the infinite multiplicative conjunction of A's where, as long as $\bot$ is making the same moves in any two of the conjuncts, $\top$ is obligated to also act the same way in those conjuncts.[8] Such a definition works fine with strict games but becomes inapplicable in the context of our free games. Some of the consequences of the technical differences in definitions include that in $R(A)$ only a countable number of parallel runs of A can be generated, while in $!A$ we may have a continuum of such runs. Since infinite-depth games generally have a continuum of legal runs, the operation R, unlike $!$, does not really allow $\bot$ to "try all possible runs of A" when playing $R(A)$ for an infinite-depth A. In [4] Blass also studied a relation on games called *weak ordering*. An attempt to turn weak ordering from a relation on games into a game operation yields an operation that could be called a "strict counterpart" of our free game operation $\Rightarrow$ defined later in this section in terms of $!$.

To visualize the scheme that lies under our definition of $!$, consider a play over $!Chess$. The play takes place between the computer ($\top$) and the user ($\bot$), and its positions are displayed on the screen. Remember from Section 8 that, using prefixation, each unilegal position can be thought of as a game, so that we will be identifying a (legal) position Φ of $Chess$ with the game $\langle\Phi\rangle Chess$. At the beginning, there is a window on the screen—call it Window ε—that displays the initial position of $Chess$:

Window ε

Chess

We denote this position by *Chess*, but the designer would probably make the window show a colorful image of a chess board with 32 chess pieces in their initial positions. The play starts and proceeds in an ordinary fashion: the players are making legal moves of *Chess*, which correspondingly update the position displayed in the window. At some (any) point—when the current position in the window is $\langle\Phi\rangle Chess$, $\bot$ may decide to replicate the position, perhaps because he wants to try different continuations

[8] This sort of requirement eventually (indirectly) amounts to $\bot$'s having branching capabilities and hence is natural. However, it may be arguable that it is perfectly appropriate when it comes to *directly* modeling real interactive situations, even when $\top$ is as (seemingly) deterministic and consistent in its actions as a computer would be. Try to play chess twice with a sufficiently sophisticated computer program. Chances are you will discover that the machine responded in different ways even if you attempted to play the two games in exactly the same way.

in different copies of it. This splits Window ε into two children windows named 0 and 1, each containing the same position $\langle\Phi\rangle Chess$ as the mother window contained at the time of split. The mother window disappears, and the picture on the screen becomes

Window 0	Window 1
$\langle\Phi\rangle Chess$	$\langle\Phi\rangle Chess$

(of course, again, with a real chess position depicted in the windows instead of the expression "$\langle\Phi\rangle Chess$").

From now on the play continues on two boards/in two windows. Either player can make a legal move in either window. After some time, when the game in Window 0 has evolved to $\langle\Phi,\Psi\rangle Chess$ and in Window 1 to $\langle\Phi,\Theta\rangle Chess$, $\bot$ can, again, decide to split one of the windows—say, Window 0. The mother window 0 will be replaced by two children windows: 00 and 01, each having the same contents as their mother had at the moment of split, so that now the screen will be showing three windows:

Window 00	Window 01	Window 1
$\langle\Phi,\Psi\rangle Chess$	$\langle\Phi,\Psi\rangle Chess$	$\langle\Phi,\Theta\rangle Chess$

If and when, at some later moment, $\bot$ decides to make a third split—say, in Window 01, the picture on the screen will look like

Window 00	Window 010	Window 011	Window 1
$\langle\Phi,\Psi,\Lambda\rangle Chess$	$\langle\Phi,\Psi,\Sigma\rangle Chess$	$\langle\Phi,\Psi,\Sigma\rangle Chess$	$\langle\Phi,\Theta,\Pi\rangle Chess$

etc. At the end, the game will be won by $\top$ if each of the windows contains a winning position of *Chess*.

The above four-window position can also be represented as the binary tree shown in Fig. 1 where the name of each window is uniquely determined by its position

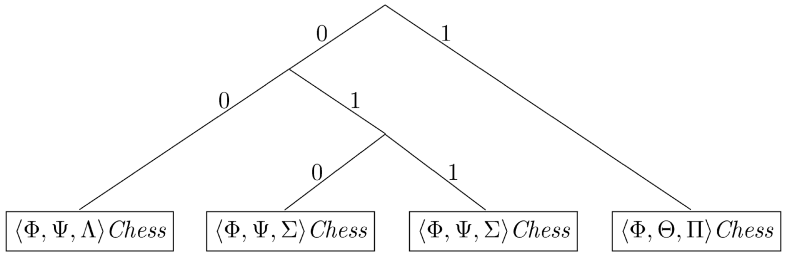

Fig. 1.

in the tree. Window names will be used by the players to indicate in which of the windows they are making a move. Sometimes the window in which a player is trying to make a move no longer exists. For example, in the position preceding the position shown in Fig. 1, $\top$ might have decided to make move α in Window 01. However, before $\top$ actually made this move, $\bot$ made a replicative move in Window 01, which took us to the four-window position shown in Fig. 1. $\top$ may not notice this replicative move and complete its move in Window 01 by the time when this window no longer exists. This kind of a move is still considered legal, and its effect will be making the move α in all (in our case—both) of the descendants of the no-longer-existing Window 01. The result will be

Window 00	Window 010	Window 011	Window 010
$\langle \Phi, \Psi, \Lambda \rangle Chess$	$\langle \Phi, \Psi, \Sigma, \top\alpha \rangle Chess$	$\langle \Phi, \Psi, \Sigma, \top\alpha \rangle Chess$	$\langle \Phi, \Theta, \Pi \rangle Chess$

This feature is crucial in ensuring the static property of the operation $!$.

The initial position in the example that we have just discussed was one-window. This, generally, is not necessary. The operation $!$ can be applied to any construction in the above style, such as, say,

Window 0	Window 1
Chess	*Checkers*

A play over this game will proceed in a way similar to what we saw, where more and more windows can be created, some (those whose names are 0-prefixed) displaying positions of chess, and some (with 1-prefixed names) displaying positions of checkers. In order to win, the machine will have to win all of the multiple parallel plays of chess and checkers that will be generated. We thus get something resembling $Chess \wedge Checkers$, which explains why the word "conjunction" appears in the name of $!$.

Let us get down to formal definitions. We will be referring to sequences of 0's and 1's, including the empty sequence ε and infinite sequences, as *bit strings*. Bit strings will usually be written without the sequence delimiters "$\langle$", "$\rangle$" around them or commas between bits, so that, say, instead of $\langle 1,0,0,1 \rangle$ we will just write 1001.

We will be using the letters w, u, v as metavariables for bit strings. The expression wu, meaningful only when w is finite, will stand for the concatenation of strings w and u. We write $w \preccurlyeq u$ to mean that w is a (not necessarily proper) initial segment of u.

Convention 13.1. (1) By a *tree* in the present context we mean a non-empty set T of bit strings, called *branches* of the tree, such that the following two conditions are satisfied:
(a) If $w \in T$ and $u \preccurlyeq w$, then $u \in T$ (all w, u);
(b) $w0 \in T$ iff $w1 \in T$ (all finite w).
If a branch w of T is finite, then it is also said to be a *node* of T.

(2) A *complete branch* of tree T is a branch w of T such that for no bit string u with $w \leqslant u \neq w$ do we have $u \in T$. A finite complete branch of T is also said to be a *leaf* of T.

(3) A *tree $\mathcal{T}$ of games* is a pair (T, G), where T is a finite tree and G is a function that assigns to each leaf w of T a game $G(w)$. By abuse of terminology, we will often identify such a $\mathcal{T}$ with its T component.

Notice that tree T is finite iff all of its branches are finite. Hence, the terms "complete branch" and "leaf" are synonyms as long as we deal with a finite tree. In Fig. 1 we see an example of a tree (T, G) of games, with $T = \{\varepsilon, 0, 1, 00, 01, 010, 011\}$ and $G(00) = \langle \Phi, \Psi, \Lambda \rangle Chess$, $G(010) = G(011) = \langle \Phi, \Psi, \Sigma \rangle Chess$, $G(1) = \langle \Phi, \Theta, \Pi \rangle Chess$.

A more space-saving way to represent a tree $\mathcal{T} = (T, G)$ of games is the following:

- If $T = \{\varepsilon\}$ with $G(\varepsilon) = A$, then $\mathcal{T}$ is represented by A.
- Otherwise, $\mathcal{T}$ is represented by $E_0 \circ E_1$, where E_0 and E_1 represent the subtrees of $\mathcal{T}$ rooted at 0 and 1, respectively.

For example, the tree of games of Fig. 1 can be written as

$$((\langle \Phi, \Psi, \Lambda \rangle Chess) \circ ((\langle \Phi, \Psi, \Sigma \rangle Chess) \circ (\langle \Phi, \Psi, \Sigma \rangle Chess))) \circ (\langle \Phi, \Theta, \Pi \rangle Chess).$$

We are going to define ! as an operation that may take as an argument not only just a game A—which can be identified with the one-node tree of games that A denotes—but any tree of games as well.

Definition 13.2. Let $\mathcal{T} = (T, G)$ be a tree of games. We define the notion of *prelegal position of* $!\mathcal{T}$, together with the function $Tree^{\mathcal{T}}$ that associates a finite tree $Tree^{\mathcal{T}}_{\Phi}$ to each such position Φ, by the following induction:

1. $\langle\rangle$ is a prelegal position of $!\mathcal{T}$, and $Tree^{\mathcal{T}}_{\langle\rangle} = T$.
2. $\langle \Phi, \lambda \rangle$ is a prelegal position of $!\mathcal{T}$ iff Φ is so and one of the following two conditions is satisfied:
   (a) $\lambda = \bot w$: for some leaf w of $Tree^{\mathcal{T}}_{\Phi}$. We call this sort of a labmove λ a *replicative labmove*. In this case $Tree^{\mathcal{T}}_{\langle \Phi, \lambda \rangle} = Tree^{\mathcal{T}}_{\Phi} \cup \{w0, w1\}$.
   (b) λ is $\bot w.\alpha$ or $\top w.\alpha$ for some node w of $Tree^{\mathcal{T}}_{\Phi}$ and move α. We call this sort of a labmove λ a *non-replicative labmove*. In this case $Tree^{\mathcal{T}}_{\langle \Phi, \lambda \rangle} = Tree^{\mathcal{T}}_{\Phi}$.

   The terms "replicative" and "non-replicative" will also be extended from labmoves to moves.

Intuitively, $Tree^{\mathcal{T}}_{\Phi}$ is the tree, in the style of Fig. 1, that will be seen on the screen after the sequence Φ of labmoves has been made when playing the game $!\mathcal{T}$; $Tree^{\mathcal{T}}_{\Phi}$ only describes the tree structure of the position represented on the screen, without showing the contents of its windows. Therefore, whether Φ is a prelegal position of $!\mathcal{T}$ and what the value of $Tree^{\mathcal{T}}_{\Phi}$ is, only depends on the T component of $\mathcal{T}$ and does not depend on its G component.

The meaning of a replicative labmove $\bot w$: is that $\bot$ replicates (splits) window w; the meaning of a non-replicative labmove $\wp w.\alpha$ is that player $\wp$ makes the move α in all windows whose names start with w.

The concept of prelegal position of $!\mathcal{T}$ can be generalized to all runs by stipulating that a *prelegal run of* $!\mathcal{T}$ is a run whose every finite initial segment is a prelegal position of $!\mathcal{T}$. Similarly, the function $Tree^{\mathcal{T}}$ can be extended to all prelegal runs of $!\mathcal{T}$ by stipulating that

$$Tree^{\mathcal{T}}_{\langle\lambda_1,\lambda_2,\lambda_3,\ldots\rangle} = Tree^{\mathcal{T}}_{\langle\rangle} \cup Tree^{\mathcal{T}}_{\langle\lambda_1\rangle} \cup Tree^{\mathcal{T}}_{\langle\lambda_1,\lambda_2\rangle} \cup Tree^{\mathcal{T}}_{\langle\lambda_1,\lambda_2,\lambda_3\rangle} \cup \cdots$$

Convention 13.3. Let u be a bit string and Γ any run. Then $\Gamma^{\preccurlyeq u}$ will stand for the result of first removing from Γ all the labmoves except those that look like $\wp w.\alpha$ for some bit string w with $w \preccurlyeq u$, and then deleting this sort of prefixes "$w.$" in the remaining labmoves, i.e. replacing each remaining labmove $\wp w.\alpha$ (where w is a bit string) by $\wp\alpha$.

Example: If $u = 101000\ldots$ and $\Gamma = \langle \top.\alpha_1, \bot:, \bot 1.\alpha_2, \top 0.\alpha_3, \bot 1:, \top 10.\alpha_4\rangle$, then $\Gamma^{\preccurlyeq u} = \langle \top\alpha_1, \bot\alpha_2, \top\alpha_4\rangle$.

Being a prelegal run of $!\mathcal{T}$ is a necessary but not a sufficient condition for being a legal run of this game. For simplicity, let us consider the case when $\mathcal{T}$ is a one-node tree A of games. It was mentioned at the beginning of this section that a legal run Γ of $!A$ can be thought of as consisting of multiple parallel legal runs of A. In particular, these runs will be the runs $\Gamma^{\preccurlyeq u}$, where u is a complete branch of $Tree^A_\Gamma$. The labmoves of $\Gamma^{\preccurlyeq u}$ for such a u are those $\wp\alpha$ that emerged as a result of making (non-replicative) labmoves of the form $\wp w.\alpha$ with $w \preccurlyeq u$. For example, to the branch 010 in Fig. 1 corresponds the run $\langle \Phi, \Psi, \Sigma\rangle$, where the labmoves of Φ originate from the non-replicative labmoves of the form $\wp\varepsilon.\alpha$ (i.e. $\wp.\alpha$) made before the first replicative move, the labmoves of Ψ originate from the non-replicative labmoves of the form $\wp w.\alpha$ with $w \preccurlyeq 0$ made between the first and the second replicative moves, and the labmoves of Σ originate from the non-replicative labmoves of the form $\wp w.\alpha$ with $w \preccurlyeq 01$ made between the second and the third replicative moves. Generally, for a prelegal run Γ of $!A$ to be a legal run, it is necessary and sufficient that all of the runs $\Gamma^{\preccurlyeq u}$, where u is a complete branch of $Tree^A_\Gamma$, be legal runs of A. And for such a Γ to be a $\top$-won run, it is necessary and sufficient that all of those $\Gamma^{\preccurlyeq u}$ be $\top$-won runs of A. In fact, the meaning of what we have just said will not change if we let u range over all infinite bit strings rather than all complete branches of $Tree^A_\Gamma$. Indeed, if u is a finite complete branch (i.e. leaf) of $Tree^A_\Gamma$, for any w we obviously have $\Gamma^{\preccurlyeq u} = \Gamma^{\preccurlyeq uw}$, so that it does not matter whether we talk about u or any of its infinite extensions uw. And vice versa: if an infinite bit string u does not happen to be a complete branch of $Tree^A_\Gamma$, in view of Convention 13.1(1b), it is clear that the shortest initial segment v of u that is a node of $Tree^A_\Gamma$ will be a complete branch (leaf) of $Tree^A_\Gamma$, so that, again because $\Gamma^{\preccurlyeq u} = \Gamma^{\preccurlyeq v}$, it does not matter whether we talk about u or v.

When $\mathcal{T}$ is a multiple-node tree of games, the situation is similar. For example, for Γ to be a legal (resp. $\top$-won) run of $!(Chess \circ Checkers)$, along with being a prelegal run, it is necessary that, for every infinite bit string $0u$, $\Gamma^{\preccurlyeq 0u}$ be a legal (resp. $\top$-won) run of *Chess* and, for every infinite bit string $1u$, $\Gamma^{\preccurlyeq 1u}$ be a legal (resp. $\top$-won) run of *Checkers*.

This intuition is summarized below in our formal definition of $!$.

Definition 13.4. Suppose $\mathcal{T}=(T,G)$ is a tree of games. The *branching* (*exponential*) *conjunction* $!\mathcal{T}$ of $\mathcal{T}$ is defined as follows:
1. $\Gamma\in\mathbf{Lr}_e^{!\mathcal{T}}$ iff the following two conditions are satisfied:
   (a) Γ is a prelegal run of $!\mathcal{T}$.
   (b) For every infinite bit string wu where w is a leaf of T, $\Gamma^{\preccurlyeq wu}\in\mathbf{Lr}_e^{G(w)}$.
2. As long as $\Gamma\in\mathbf{Lr}_e^{!\mathcal{T}}$, $\mathbf{Wn}_e^{!\mathcal{T}}\langle\Gamma\rangle=\top$ iff $\mathbf{Wn}_e^{G(w)}\langle\Gamma^{\preccurlyeq wu}\rangle=\top$ for every infinite bit string wu where w is a leaf of T.

As always, we want to have an inductive characterization (see page 26) of the game operation we have just introduced. The initial conditions are pretty obvious:

Proposition 13.5. *Let $\mathcal{T}=(T,G)$ be a tree of games. Then*:
1. *λ is a legal initial labmove of $!\mathcal{T}$ w.r.t. e iff one of the following conditions is satisfied*:
   (a) *λ is the* (*replicative*) *labmove $\bot w$: for some leaf w of T.*
   (b) *λ is the* (*non-replicative*) *labmove $\wp w.\alpha$ for some node w of T and move α such that $\alpha\in\mathbf{Lm}\wp_e^{G(u)}\langle\rangle$ for every leaf u of T with $w\preccurlyeq u$.*
2. $\mathbf{Wn}_e^{!\mathcal{T}}\langle\rangle=\top$ *iff, for each leaf w of T,* $\mathbf{Wn}_e^{G(w)}\langle\rangle=\top$.

For the inductive clause (Proposition 13.8), we need the following two definitions.

Definition 13.6. Let $\mathcal{T}=(T,G)$ be a tree of games and w a leaf of T. We define $Rep_w[\mathcal{T}]$ as the following tree (T',G') of games:
1. $T'=T\cup\{w0,w1\}$.
2. (a) $G'(w0)=G'(w1)=G(w)$.
   (b) For every other (different from $w0,w1$) leaf u of T', $G'(u)=G(u)$.

Example. $Rep_{10}[A\circ(B\circ C)]=A\circ((B\circ B)\circ C)$.

Definition 13.7. Suppose $\mathcal{T}=(T,G)$ is a tree of games, w is a node of T and, for every leaf u of T with $w\preccurlyeq u$, λ is a unilegal initial labmove of $G(u)$. We define $Nonrep_w^{\lambda}[\mathcal{T}]$ as the tree (T,G') of games such that:
1. For every leaf u of T with $w\preccurlyeq u$, $G'(u)=\langle\lambda\rangle G(u)$.
2. For every other leaf u of T, $G'(u)=G(u)$.

Example. $Nonrep_1^{\lambda}[A\circ(B\circ C)]=A\circ(\langle\lambda\rangle B\circ\langle\lambda\rangle C)$ (any λ that is a unilegal initial labmove of both B and C).

Proposition 13.8. *Suppose $\mathcal{T}$ is a tree of games, and λ is a unilegal initial labmove of $!\mathcal{T}$.*
1. *If λ is a replicative labmove $\bot w$:, then $\langle\lambda\rangle!\mathcal{T}=!Rep_w[\mathcal{T}]$.*
2. *If λ is a non-replicative labmove $\wp w.\alpha$, then $\langle\lambda\rangle!\mathcal{T}=!Nonrep_w^{\wp\alpha}[\mathcal{T}]$.*

We postpone the proof of this fact till the end of the section. As for now, let us see an application of the above inductive characterization of $!$.

Example 13.9. In view of Propositions 13.5 (initial conditions) and 13.8 (inductive clause), the sequence $\langle \bot:, \bot 0.4, \top 0.16, \bot 1:, \bot 10.5, \top 10.25\rangle$ is a unilegal, $\top$-won run of the game $!\sqcap x \sqcup y(y = x^2)$, and the game-sequence representation of this run is:

0. $!\,(\sqcap x \sqcup y(y = x^2))$.
1. $!\,(\sqcap x \sqcup y(y = x^2) \circ \sqcap x \sqcup y(y = x^2))$.
2. $!\,(\sqcup y(y = 4^2) \circ \sqcap x \sqcup y(y = x^2))$.
3. $!\,((16 = 4^2) \circ \sqcap x \sqcup y(y = x^2))$.
4. $!\,((16 = 4^2) \circ (\sqcap x \sqcup y(y = x^2) \circ \sqcap x \sqcup y(y = x^2)))$.
5. $!\,((16 = 4^2) \circ (\sqcup y(y = 5^2) \circ \sqcap x \sqcup y(y = x^2)))$.
6. $!\,((16 = 4^2) \circ ((25 = 5^2) \circ \sqcap x \sqcup y(y = x^2)))$.

Every version of conjunction has its dual disjunction, and so does $!$. For a tree $\mathscr{T} = (T, G)$ of games, let $\neg\mathscr{T}$ mean the tree (T, G') of games where, for each leaf w of T, $G'(w) = \neg G(w)$. Then we define the *branching* (*exponential*) *disjunction* $?\mathscr{T}$ of $\mathscr{T}$ by

$$?\mathscr{T} =_{\text{def}} \neg!\neg\mathscr{T}.$$

As we may guess, the meaning of $?\mathscr{T}$ is the same as that of $!\mathscr{T}$, only with the roles of the two players interchanged: in a $?$-game, it is $\top$ rather than $\bot$ who can replicate positions, and, in order to win, it is sufficient for $\top$ to win in at least one rather than all of the parallel runs that will be created during the play.

Remembering that a game is nothing but a one-node tree of games, from now on we will usually treat $!$ and $?$ as game operations rather than (the more general) operations on trees of games. The only reason why we originally defined them as operations on trees of games was to make inductive characterizations and hence visualizations in the style of Example 13.9 possible.

The *weak reduction* $A \Rightarrow B$ of game B (called the *consequent*) to game A (called the *antecedent*) is defined by

$$A \Rightarrow B =_{\text{def}} (!A) \rightarrow B.$$

Alternative names for $\Rightarrow$ are *branching implication* and *exponential implication*. We saw an informal description of a play over a $\Rightarrow$-game in Section 1 when discussing the intuitive meaning of weak reduction.

For both $!$ and $?$ we can define their *bounded versions* $!^b$ and $?^b$, where b is a positive integer. The meanings of $!^bA$ and $?^bA$ are the same as those of $!A$ and $?A$, with the only difference that the number of parallel plays of A that the corresponding player can create should not exceed b, that is, at most $b-1$ replicative moves can be made legally.

As an exercise showing the essential difference between branching and parallel operations (something that Example 13.9 fails to really demonstrate), the reader may want to compare the games $?^2 D$ and $D \vee D$, where

$$D = (Chess \sqcup \neg Chess) \sqcap (Checkers \sqcup \neg Checkers).$$

Which of the two games is easier for $\top$ to win?

It is evident that the operations $!, ?, \Rightarrow$, being similar to the parallel operations, are not strict. Nor are they elementary, for replicative moves, even though pointless, would still be legal in $!A$ with an elementary A, so that $!A$ would not even be perifinite-depth, let alone elementary. So, the branching operations are not finite- or perifinite-depth, either. It is left as an exercise for the reader to verify that $!$ (and hence $?$ and $\Rightarrow$) are finitary, unistructural and x-unistructural. Finally, like all the other game operations that we have seen, the branching operations are static. To prove this, we need the following lemma:

Lemma 13.10. *Assume A is a static game, Δ is a $\wp$-delay of Γ, and Δ is a $\wp$-illegal run of $e[!A]$. Then Γ is also a $\wp$-illegal run of $e[!A]$.*

Proof. Assume the conditions of the lemma. Valuation e will be fixed throughout the context and does not need to be explicitly mentioned. We want to show that Γ is a $\wp$-illegal run of $!A$.

Let $\langle \Psi, \wp\alpha \rangle$ be the shortest initial segment of Δ that is a $\wp$-illegal position of $!A$. Let $\langle \Phi, \wp\alpha \rangle$ be the shortest initial segment of Γ containing all the $\wp$-labeled moves of $\langle \Psi, \wp\alpha \rangle$, and let Θ be the sequence of those $\neg\wp$-labeled moves of Ψ that are not in Φ, so that we obviously have

$$\langle \Psi, \wp\alpha \rangle \text{ is a } \wp\text{-delay of } \langle \Phi, \wp\alpha, \Theta \rangle. \tag{14}$$

Excluding from our considerations the trivial case when Φ is a $\wp$-illegal position of $!A$ and reasoning exactly as in the proof of Lemma 12.5, the following statement can be shown to be true:

$$\Phi \text{ is a legal position of } !A. \tag{15}$$

There are two possible reasons to why $\langle \Psi, \wp\alpha \rangle$ is a $\wp$-illegal (while Ψ being legal) position of $!A$:

Reason 1: $\langle \Psi, \wp\alpha \rangle$ is not a prelegal position of $!A$. We claim that then $\langle \Phi, \wp\alpha \rangle$ is not a prelegal position of $!A$ which, in view of (15), implies that $\langle \Phi, \wp\alpha \rangle$ and hence its extension Γ is a $\wp$-illegal run of $!A$. To prove this claim, suppose, for a contradiction, that $\langle \Phi, \wp\alpha \rangle$ is a prelegal position of $!A$. Note that $Tree^A_\Psi$ and $Tree^A_\Phi$ (Definition 13.2) are both defined because both Ψ and Φ are legal (Ψ by our original assumption and Φ by (15)) and hence prelegal positions of $!A$. There are two cases to consider:

Case 1: $\wp = \bot$. Note that then Φ and Ψ have the same subsequences of $\bot$-labeled moves and hence (as all replicative moves are $\bot$-labeled) the same subsequences of

replicative labmoves. From Definition 13.2 then it is obvious that $Tree^A_\Psi = Tree^A_\Phi$. This is so because $Tree^A$ is modified—in particular, extended—only by replicative moves. Then we immediately get that $\langle \Psi, \wp\alpha\rangle$ is a prelegal position of $!A$ if and only if $\langle \Phi, \wp\alpha\rangle$ is so, which is a contradiction.

Case 2: $\wp = \top$. This means that $\alpha = w.\beta$ for some $w \in Tree^A_\Phi$. Note that, since $\wp = \top$, the subsequence of $\bot$-labeled moves of Φ is an initial segment of that of Ψ, so that the subsequence of replicative labmoves of Φ is an initial segment of that of Ψ. This obviously implies that $Tree^A_\Phi \subseteq Tree^A_\Psi$, from which we can conclude that $w \in Tree^A_\Psi$ and hence $\langle \Psi, \top w.\beta\rangle$, i.e. $\langle \Psi, \wp\alpha\rangle$, is a prelegal position of $!A$, which, again, is a contradiction.

Reason 2: There is an infinite bit string u such that $\langle \Psi, \wp\alpha\rangle^{\leqslant u}$ is not a legal position of A. Since Ψ, by our original assumption, is a legal position of $!A$, $\Psi^{\leqslant u}$ must be a legal position of A. Hence, $\langle \Psi, \wp\alpha\rangle^{\leqslant u}$ is a $\wp$-illegal position of A. From (14) we can see that $\langle \Psi, \wp\alpha\rangle^{\leqslant u}$ is a $\wp$-delay of $\langle \Phi, \wp\alpha, \Theta\rangle^{\leqslant u}$. Therefore, taking into account that A is static, Lemma 4.7.1 implies that $\langle \Phi, \wp\alpha, \Theta\rangle^{\leqslant u}$ is a $\wp$-illegal position of A. Then $\langle \Phi, \wp\alpha\rangle^{\leqslant u}$ is also a $\wp$-illegal position of A, because all the moves of Θ are $\neg\wp$-labeled. This makes $\langle \Phi, \wp\alpha\rangle$ and hence its extension Γ a $\wp$-illegal run of $!A$. □

Proposition 13.11. *The operations* $!$, $?$ *and* $\Rightarrow$ *are static.*

Proof. As always, it will be sufficient to prove the proposition for only the basic operation of the group, in particular, $!$. It should be pointed out that the proof we give below can be easily converted into a proof of the fact that the operation $!$, applied to any tree of static games rather than a single-node tree A, also yields a static game.

Assume A is a static game, Δ is a $\wp$-delay of Γ, and $\mathbf{Wn}^{!A}_e\langle\Gamma\rangle = \wp$. We need to show that $\mathbf{Wn}^{!A}_e\langle\Delta\rangle = \wp$.

If Δ is a $\neg\wp$-illegal run of $e[!A]$, then it is won by $\wp$ and we are done. So, assume that Δ is not $\neg\wp$-illegal. Then, by Lemmas 4.6 and 13.10, Γ is not $\neg\wp$-illegal, either. Γ also cannot be $\wp$-illegal, for otherwise it would not be won by $\wp$. Consequently, Δ cannot be $\wp$-illegal either, for otherwise, by Lemma 13.10, Γ would be $\wp$-illegal. Thus, we have narrowed down our considerations to the case when both Γ and Δ are legal runs of $e[!A]$.

$\mathbf{Wn}^{!A}_e\langle\Gamma\rangle = \wp$, together with $\Gamma \in \mathbf{Lr}^{!A}_e$, implies that for each infinite bit string u (if $\wp = \top$), or one of such strings u (if $\wp = \bot$), $\Gamma^{\leqslant u}$ is a $\wp$-won run of $e[A]$. Taking into account that $\Delta^{\leqslant u}$ is obviously a $\wp$-delay of $\Gamma^{\leqslant u}$ and that A is static, the above, in turn, implies that for each infinite bit string u (if $\wp = \top$), or one of such strings u (if $\wp = \bot$), $\Delta^{\leqslant u}$ is a $\wp$-won run of $e[A]$, which, taking into account that $\Delta \in \mathbf{Lr}^{!A}_e$, means nothing but that Δ is a $\wp$-won run of $e[!A]$. □

Our remaining duty is to prove Proposition 13.8, to which the rest of this section is devoted. The reader who has no reasons to doubt the correctness of that proposition, may safely skip this part.

Proof of Proposition 13.8. *Clause* 1: Assume $\mathcal{T} = (T, G)$ and $\bot w$: is a unilegal initial (replicative) labmove of $!\mathcal{T}$. Let us fix the tree $\mathcal{T}' = (T', G')$ of games with

$\mathscr{T}' = Rep_w[\mathscr{T}]$. Our goal is to show that both the **Lr** and the **Wn** components of the games $!\mathscr{T}'$ and $\langle \bot w{:}\rangle !\mathscr{T}$ are the same.

At first, we claim that, for any Φ,

$$\Phi \text{ is a prelegal position of } !\mathscr{T}' \text{ iff } \langle \bot w{:}, \Phi\rangle \text{ is a prelegal position of } !\mathscr{T}. \tag{16}$$

This claim can be verified by induction on the length of Φ, showing, in parallel, that when $\langle \bot w{:}, \Phi\rangle$ is a prelegal position of $!\mathscr{T}$ (and hence Φ a prelegal position of $!\mathscr{T}'$), $Tree_{\Phi}^{\mathscr{T}'} = Tree^{\mathscr{T}}_{\langle \bot w{:}, \Phi\rangle}$. We omit here this straightforward induction which is based on a simple analysis of Definition 13.2.

Consider an arbitrary position Φ. We want to show that $\Phi \notin \mathbf{Lr}_e^{\langle \bot w{:}\rangle !\mathscr{T}}$ iff $\Phi \notin \mathbf{Lr}_e^{!\mathscr{T}'}$.

Assume $\Phi \notin \mathbf{Lr}_e^{\langle \bot w{:}\rangle !\mathscr{T}}$, i.e. (by Definition 8.1) $\langle \bot w{:}, \Phi\rangle \notin \mathbf{Lr}_e^{!\mathscr{T}}$. If the reason for $\langle \bot w{:}, \Phi\rangle \notin \mathbf{Lr}_e^{!\mathscr{T}}$ is that $\langle \bot w{:}, \Phi\rangle$ is not a prelegal position of $!\mathscr{T}$, then, by (16), Φ is not a prelegal position of $!\mathscr{T}'$ and hence $\Phi \notin \mathbf{Lr}_e^{!\mathscr{T}'}$. Suppose now the reason for $\langle \bot w{:}, \Phi\rangle \notin \mathbf{Lr}_e^{!\mathscr{T}}$ is that, for some leaf v of T and some infinite bit string u with $v \preccurlyeq u$, $\langle \bot w{:}, \Phi\rangle^{\preccurlyeq u} \notin \mathbf{Lr}_e^{G(v)}$, i.e., since we clearly have $\langle \bot w{:}, \Phi\rangle^{\preccurlyeq u} = \Phi^{\preccurlyeq u}$, $\Phi^{\preccurlyeq u} \notin \mathbf{Lr}_e^{G(v)}$. If v happens to be a leaf of T', then we immediately get $\Phi \notin \mathbf{Lr}_e^{!\mathscr{T}'}$. Otherwise, we must have $v = w$. Then $w0 \preccurlyeq u$ or $w1 \preccurlyeq u$. In either case there is a leaf v' ($= w0$ or $w1$) of T' such that $v' \preccurlyeq u$ and $\Phi^{\preccurlyeq u} \notin \mathbf{Lr}_e^{G(w)}$. But note that, by Definition 13.6(2a), $G(w) = G'(v')$. Thus, again we get that $\Phi \notin \mathbf{Lr}_e^{!\mathscr{T}'}$.

Now assume $\Phi \notin \mathbf{Lr}_e^{!\mathscr{T}'}$. If the reason for $\Phi \notin \mathbf{Lr}_e^{!\mathscr{T}'}$ is that Φ is not a prelegal position of $!\mathscr{T}'$, then, by (16), $\langle \bot w{:}, \Phi\rangle$ is not a prelegal position of $!\mathscr{T}$, whence $\langle \bot w{:}, \Phi\rangle \notin \mathbf{Lr}_e^{!\mathscr{T}}$, which means that $\Phi \notin \mathbf{Lr}_e^{\langle \bot w{:}\rangle !\mathscr{T}}$. Suppose now the reason for $\Phi \notin \mathbf{Lr}_e^{!\mathscr{T}'}$ is that, for some leaf v of T' and some infinite bit string u with $v \preccurlyeq u$, $\Phi^{\preccurlyeq u}$ is not a legal position of $G'(v)$ w.r.t. e. If v happens to be a leaf of T, then $G'(v) = G(v)$ and therefore $\Phi^{\preccurlyeq u} \notin \mathbf{Lr}_e^{G(v)}$, i.e. $\langle \bot w{:}, \Phi\rangle^{\preccurlyeq u} \notin \mathbf{Lr}_e^{G(v)}$, whence $\langle \bot w{:}, \Phi\rangle \notin \mathbf{Lr}_e^{!\mathscr{T}}$, i.e. $\Phi \notin \mathbf{Lr}_e^{\langle \bot w{:}\rangle !\mathscr{T}}$. Suppose now v is not a leaf of T. Then $v = w0$ or $v = w1$, so that $G'(v) = G(w)$ and hence $\Phi^{\preccurlyeq u} \notin \mathbf{Lr}_e^{G(w)}$, i.e. $\langle \bot w{:}, \Phi\rangle^{\preccurlyeq u} \notin \mathbf{Lr}_e^{G(w)}$ which, taking into account that $w \preccurlyeq u$, again implies that $\langle \bot w{:}, \Phi\rangle \notin \mathbf{Lr}_e^{!\mathscr{T}}$, i.e. $\Phi \notin \mathbf{Lr}_e^{\langle \bot w{:}\rangle !\mathscr{T}}$.

Next, consider an arbitrary run Γ with $\Gamma \in \mathbf{Lr}_e^{!\mathscr{T}'} = \mathbf{Lr}_e^{\langle \bot w{:}\rangle !\mathscr{T}}$. We want to show that $\mathbf{Wn}_e^{\langle \bot w{:}\rangle !\mathscr{T}}\langle \Gamma\rangle = \bot$ iff $\mathbf{Wn}_e^{!\mathscr{T}'}\langle \Gamma\rangle = \bot$.

Assume $\mathbf{Wn}_e^{\langle \bot w{:}\rangle !\mathscr{T}}\langle \Gamma\rangle = \bot$, i.e. $\mathbf{Wn}_e^{!\mathscr{T}}\langle \bot w{:}, \Gamma\rangle = \bot$. Then, by Definition 13.4.2, there is a leaf v of T and an infinite bit string u with $v \preccurlyeq u$ such that $\mathbf{Wn}_e^{G(v)}\langle \bot w{:}, \Gamma\rangle^{\preccurlyeq u} = \bot$, i.e. $\mathbf{Wn}_e^{G(v)}\langle \Gamma^{\preccurlyeq u}\rangle = \bot$. If v also happens to be a leaf of T', then, by Definition 13.6, $G'(v) = G(v)$ and, hence, $\mathbf{Wn}_e^{!\mathscr{T}'}\langle \Gamma\rangle = \bot$. Otherwise, we have $v = w$. But then $w0 \preccurlyeq u$ or $w1 \preccurlyeq u$ and, since $w0, w1$ are leaves of T' and $G'(w0) = G'(w1) = G(v)$, we again have that $\mathbf{Wn}_e^{!\mathscr{T}'}\langle \Gamma\rangle = \bot$.

Now assume that $\mathbf{Wn}_e^{!\mathscr{T}'}\langle \Gamma\rangle = \bot$. There must be a leaf v of T' and an infinite bit string u with $v \preccurlyeq u$ such that $\mathbf{Wn}_e^{G'(v)}\langle \Gamma^{\preccurlyeq u}\rangle = \bot$. Just as in the previous case, if v also happens to be a leaf of T, we get that $\mathbf{Wn}_e^{\langle \bot w{:}\rangle !\mathscr{T}}\langle \Gamma\rangle = \bot$. Otherwise, v must be $w0$ or $w1$. In either case $w \preccurlyeq u$ and $\mathbf{Wn}_e^{G(w)}\langle \Gamma^{\preccurlyeq u}\rangle = \bot$, which, again, implies that $\mathbf{Wn}_e^{\langle \bot w{:}\rangle !\mathscr{T}}\langle \Gamma\rangle = \bot$. □

Clause 2: Assume $\mathcal{T}=(T,G)$ and $\wp w.\alpha$ is a unilegal initial non-replicative labmove of $!\mathcal{T}$. Let us fix the tree $\mathcal{T}'=(T,G')$ of games with $\mathcal{T}'=Nonrep_w^{\wp\alpha}[\mathcal{T}]$. Our goal is to show that both the **Lr** and the **Wn** components of the games $!\mathcal{T}'$ and $\langle\wp w.\alpha\rangle!\mathcal{T}$ are the same.

Just as in the proof of clause 1 (statement (16))—in fact, in an even easier way—we can show that, for any Φ,

$$\Phi \text{ is a prelegal position of } !\mathcal{T}' \text{ iff } \langle\wp w.\alpha,\Phi\rangle \text{ is a prelegal position of } !\mathcal{T}. \tag{17}$$

Consider an arbitrary position Φ. We want to show that $\Phi\notin\mathbf{Lr}_e^{\langle\wp w.\alpha\rangle!\mathcal{T}}$ iff $\Phi\notin\mathbf{Lr}_e^{!\mathcal{T}'}$.

Assume $\Phi\notin\mathbf{Lr}_e^{\langle\wp w.\alpha\rangle!\mathcal{T}}$, i.e. $\langle\wp w.\alpha,\Phi\rangle\notin\mathbf{Lr}_e^{!\mathcal{T}}$. If the reason for $\langle\wp w.\alpha,\Phi\rangle\notin\mathbf{Lr}_e^{!\mathcal{T}}$ is that $\langle\wp w.\alpha,\Phi\rangle$ is not a prelegal position of $!\mathcal{T}$, then, by (17), Φ is not a prelegal position of $!\mathcal{T}'$, which implies that $\Phi\notin\mathbf{Lr}_e^{!\mathcal{T}'}$. Suppose now the reason for $\langle\wp w.\alpha,\Phi\rangle\notin\mathbf{Lr}_e^{!\mathcal{T}}$ is that for some leaf v of T and some infinite bit string u with $v\preccurlyeq u$, $\langle\wp w.\alpha,\Phi\rangle^{\preccurlyeq u}\notin\mathbf{Lr}_e^{G(v)}$. If $w\not\preccurlyeq v$, then $G(v)=G'(v)$ and $\langle\wp w.\alpha,\Phi\rangle^{\preccurlyeq u}=\Phi^{\preccurlyeq u}$, so that $\Phi^{\preccurlyeq u}\notin\mathbf{Lr}_e^{G'(v)}$ and hence $\Phi\notin\mathbf{Lr}_e^{!\mathcal{T}'}$. Suppose now $w\preccurlyeq v$. Notice that then $\langle\wp w.\alpha,\Phi\rangle^{\preccurlyeq u}=\langle\wp\alpha,\Phi^{\preccurlyeq u}\rangle$. So, since $\langle\wp w.\alpha,\Phi\rangle^{\preccurlyeq u}\notin\mathbf{Lr}_e^{G(v)}$, we have $\langle\wp\alpha,\Phi^{\preccurlyeq u}\rangle\notin\mathbf{Lr}_e^{G(v)}$, i.e. $\Phi^{\preccurlyeq u}\notin\mathbf{Lr}_e^{\langle\wp\alpha\rangle G(v)}$. But by Definition 13.7.1, $\langle\wp\alpha\rangle G(v)=G'(v)$. Hence, $\Phi^{\preccurlyeq u}\notin\mathbf{Lr}_e^{G'(v)}$, which means that $\Phi\notin\mathbf{Lr}_e^{!\mathcal{T}'}$.

Now assume $\Phi\notin\mathbf{Lr}_e^{!\mathcal{T}'}$. If the reason for $\Phi\notin\mathbf{Lr}_e^{!\mathcal{T}'}$ is that Φ is not a prelegal position of $!\mathcal{T}'$, then, by (17), $\langle\wp w.\alpha,\Phi\rangle$ is not a prelegal position of $!\mathcal{T}$, so that $\langle\wp w.\alpha,\Phi\rangle\notin\mathbf{Lr}_e^{!\mathcal{T}}$, i.e. $\Phi\notin\mathbf{Lr}_e^{\langle\wp w.\alpha\rangle!\mathcal{T}}$. Suppose now the reason for $\Phi\notin\mathbf{Lr}_e^{!\mathcal{T}'}$ is that, for some leaf v of T and some infinite bit string u with $v\preccurlyeq u$, $\Phi^{\preccurlyeq u}\notin\mathbf{Lr}_e^{G'(v)}$. If $w\not\preccurlyeq v$, then $G'(v)=G(v)$ and $\Phi^{\preccurlyeq u}=\langle\wp w.\alpha,\Phi\rangle^{\preccurlyeq u}$, so that $\langle\wp w.\alpha,\Phi\rangle^{\preccurlyeq u}\notin\mathbf{Lr}_e^{G(v)}$ and hence $\langle\wp w.\alpha,\Phi\rangle\notin\mathbf{Lr}_e^{!\mathcal{T}}$, i.e. $\Phi\notin\mathbf{Lr}_e^{\langle\wp w.\alpha\rangle!\mathcal{T}}$. Suppose now $w\preccurlyeq v$. Then $\langle\wp w.\alpha,\Phi\rangle^{\preccurlyeq u}=\langle\wp\alpha,\Phi^{\preccurlyeq u}\rangle$. Since $\Phi^{\preccurlyeq u}\notin\mathbf{Lr}_e^{G'(v)}$ and $G'(v)=\langle\wp\alpha\rangle G(v)$, we have that $\Phi^{\preccurlyeq u}\notin\mathbf{Lr}_e^{\langle\wp\alpha\rangle G(v)}$, i.e. $\langle\wp\alpha,\Phi^{\preccurlyeq u}\rangle\notin\mathbf{Lr}_e^{G(v)}$, whence $\langle\wp w.\alpha,\Phi\rangle^{\preccurlyeq u}\notin\mathbf{Lr}_e^{G(v)}$, which means that $\langle\wp w.\alpha,\Phi\rangle\notin\mathbf{Lr}_e^{!\mathcal{T}}$, i.e. $\Phi\notin\mathbf{Lr}_e^{\langle\wp w.\alpha\rangle!\mathcal{T}}$.

Next, consider an arbitrary run Γ with $\Gamma\in\mathbf{Lr}_e^{!\mathcal{T}'}=\mathbf{Lr}_e^{\langle\wp w.\alpha\rangle!\mathcal{T}}$. We want to show that $\mathbf{Wn}_e^{\langle\wp w.\alpha\rangle!\mathcal{T}}\langle\Gamma\rangle=\bot$ iff $\mathbf{Wn}_e^{!\mathcal{T}'}\langle\Gamma\rangle=\bot$.

Assume $\mathbf{Wn}_e^{\langle\wp w.\alpha\rangle!\mathcal{T}}\langle\Gamma\rangle=\bot$, i.e. $\mathbf{Wn}_e^{!\mathcal{T}}\langle\wp w.\alpha,\Gamma\rangle=\bot$. Then there is a leaf v of T and an infinite bit string u with $v\preccurlyeq u$ such that $\mathbf{Wn}_e^{G(v)}\langle\wp w.\alpha,\Gamma\rangle^{\preccurlyeq u}=\bot$. If $w\not\preccurlyeq v$, then $G(v)=G'(v)$ and $\langle\wp w.\alpha,\Gamma\rangle^{\preccurlyeq u}=\Gamma^{\preccurlyeq u}$, so that we have $\mathbf{Wn}_e^{G'(v)}(\Gamma^{\preccurlyeq u})=\bot$, which implies $\mathbf{Wn}_e^{!\mathcal{T}'}\langle\Gamma\rangle=\bot$. Suppose now $w\preccurlyeq v$. Then $G'(v)=\langle\wp\alpha\rangle G(v)$ and $\langle\wp w.\alpha,\Gamma\rangle^{\preccurlyeq u}=\langle\wp\alpha,\Gamma^{\preccurlyeq u}\rangle$. Therefore we have $\mathbf{Wn}_e^{G(v)}\langle\wp\alpha,\Gamma^{\preccurlyeq u}\rangle=\bot$, i.e. $\mathbf{Wn}_e^{\langle\wp\alpha\rangle G(v)}\langle\Gamma^{\preccurlyeq u}\rangle=\bot$, i.e. $\mathbf{Wn}_e^{G'(v)}\langle\Gamma^{\preccurlyeq u}\rangle=\bot$, which, again, implies that $\mathbf{Wn}_e^{!\mathcal{T}'}\langle\Gamma\rangle=\bot$.

Now, assume $\mathbf{Wn}_e^{!\mathcal{T}'}\langle\Gamma\rangle=\bot$. There must be a leaf v of T and an infinite bit string u with $v\preccurlyeq u$ such that $\mathbf{Wn}_e^{G'(v)}\langle\Gamma^{\preccurlyeq u}\rangle=\bot$. Just as in the previous case, if $w\not\preccurlyeq v$, then $G'(v)=G(v)$ and $\Gamma^{\preccurlyeq u}=\langle\wp w.\alpha,\Gamma\rangle^{\preccurlyeq u}$, so that $\mathbf{Wn}_e^{G(v)}\langle\wp w.\alpha,\Gamma\rangle^{\preccurlyeq u}=\bot$, which implies $\mathbf{Wn}_e^{\langle\wp w.\alpha\rangle!\mathcal{T}}\langle\Gamma\rangle=\bot$. Suppose now $w\preccurlyeq v$. Then, as in the above paragraph, $G'(v)=\langle\wp\alpha\rangle G(v)$ and $\langle\wp w.\alpha,\Gamma\rangle^{\preccurlyeq u}=\langle\wp\alpha,\Gamma^{\preccurlyeq u}\rangle$. Therefore $\mathbf{Wn}_e^{\langle\wp\alpha\rangle G(v)}\langle\Gamma^{\preccurlyeq u}\rangle=\bot$, i.e.

$\mathbf{Wn}_e^{G(v)}\langle \wp\alpha, \Gamma^{\preccurlyeq u}\rangle = \bot$, whence $\mathbf{Wn}_e^{G(v)}\langle \wp w.\alpha, \Gamma\rangle^{\preccurlyeq u} = \bot$, which, again, implies $\mathbf{Wn}_e^{\langle \wp w.\alpha\rangle !\mathcal{T}}\langle \Gamma\rangle = \bot$. □

14. Summary of game operation properties

The following theorem summarizes our observations or proofs of what "desirable" properties hold or do not hold for the game operations that we introduced.

Theorem 14.1. *For each of the following eight game properties, the class of the games with that property is closed* (+) *or not* (−) *under our game operations as shown below*:

	prefixation, $\bot, \top, \neg, \exists, \forall$	$\vee, \wedge$, $\rightarrow$	$\sqcup, \sqcap$, $\bigsqcup, \bigsqcap$	$!, ?$, $\Rightarrow$	*subst. of variables*
static	+	+	+	+	+
finitary	+	+	+	+	+
unistructural	+	+	+	+	+
x-unistructural	+	+	+	+	−
finite-depth	+	+	+	−	+
perifinite-depth	+	+	+	−	+
elementary	+	+	−	−	+
strict	+	−	+	−	+

In view of this theorem and the fact that predicates as elementary games are static and unistructural, the closure of predicates under all our game operations forms a natural class of static, unistructural games. If we restrict predicates to finitary ones, then all the games in this closure will also be finitary. The most important and hence most carefully proven row in the above table is the first one of course, for a "−" in that row would disqualify the operation as an operation on computational problems which, as we agreed, are nothing but static games, and are viewed as foundational entities by our philosophy. *Dynamic* (non-static, or not-necessarily-static) games, with various additional timing conditions, can be of interest only when studying real-time computability and the like rather than "pure" computability, and we exclude them from the scope of our present considerations, even though all of the definitions that we have seen so far equally make sense for both static and dynamic games. The rest of the paper will be mostly (Part II) or exclusively (Part III) focused on static games.

Part II—Computability, i.e. winnability

The definitions of the previous part explain when a game, or a combination of games, should be considered won by a given player once we already have a particular run of the game. A separate question that has not been formally addressed so far is

about how those runs are produced. In other words, how games are *played* up. This Part contains a precise answer to that question and, based on it, offers a formalization of our intuitive concept of interactive algorithmic solvability (computability)—in particular, a generalization of the Church-Turing thesis from simple (two-step) problems to all interactive computational problems.

15. The hard-play model

We understand that the two players $\top$ and $\bot$ are agents with the capability of making moves, and the sequence of the moves that they generate during the process of playing the game forms the run. Before trying to mathematically formalize what such "agents" mean, any game semantics needs to answer the question regarding what the *procedural rules* for playing games are—the rules regulating when and which player is allowed to move.

In strict games an answer to the procedural rules question comes naturally: in a position where there are legal moves, a move can (or should?) be made by the very player who has a legal move. This also yields a simple answer to the question regarding what kind of mathematical objects the players are: a player $\wp$ in a strict game can be understood as a function $f_\wp$ that assigns, to every position Φ where $\wp$ has legal moves, one of those moves. Then, given a position Φ where $\wp$ has legal moves, the next move will be the ($\wp$-labeled) value of $f_\wp(\Phi)$; in case none of the players had a legal move in Φ, this position will become the final position.

Our games, however, are not strict. It is possible that in a given position Φ both $\top$ and $\bot$ have legal moves. Which of them can then make the next move? As this was mentioned earlier, the answer we choose is simple: *either*. Moreover, it is not necessary for a player to have a legal move in Φ to get a chance to move: the player may make an illegal move as well, which, however, will result in an immediate loss for that player. This simple answer to the procedural rules question, however, somewhat complicates formalization of players as mathematical objects, making understanding their strategies as functions from positions to moves no longer possible. Indeed: if, say, for position Φ $\top$'s functional strategy returns α and $\bot$'s functional strategy returns β, then it is not clear whether the next labmove should be $\top\alpha$ or $\bot\beta$.

The fact that strategies can (or should) no longer be formalized as functions is a natural and welcome implication of the main advantages of our free-game approach over the strict-game approach discussed in Section 3. Functions are basically designed to perform non-interactive tasks, and any attempt to reduce interactive strategies to functions could mean some sort of "cheating" (oversimplification), the penalty for which would be losing much of the potential of games to model truly interactive tasks. An eloquent demonstration of this is the strategy described in Example 3.2, turning which into a function would hardly be possible without counterfeiting the essence of the computational problem it was meant to solve. Note also that, in real life, hardly many interactive computer programs—even those performing strict tasks (playing strict games)—act as functions that start computing from scratch every time they have to make a move, forgetting/destroying everything they have done before (their internal

state, that is) and looking only at the current position of the game. Acting this way, after all, could seriously impair their efficiency.

So, now the question that needs to be answered is the question about how to define, mathematically, what the two players or their possible behaviors are. Our philosophical point of departure is that the agent $\top$ should be implementable as a computer program, with a fully determined, predictable and computable behavior, while the behavior of $\bot$, who represents the environment—a capricious user or the blind forces of nature—can be arbitrary.

As Turing machines are the most popular mathematical model for computer programs, we are going to define the agent $\top$ as a sort of Turing machine, with the capability of making moves. This machine is assumed to be fully informed. In particular, it has full information regarding the valuation of **Variables** with respect to which the outcome of the play will be evaluated. And, at any moment, it also has full information regarding what moves and in what order have been made so far—that is, information on the current position. As for the agent $\bot$, there is no formal necessity to formalize it as a separate mathematical object (even though we will still do so in Section 20). Rather, its arbitrary behavior can be accounted for in the definition of what the possible branches of computation (play) for $\top$ are, where different branches correspond to different possible behaviors by $\bot$.

The basic mathematical model of $\top$ is what we call a "hard-play machine" (HPM). This is a version of Turing machine which, along with the standard read/write *work tape* (and hence all the computational power of an ordinary Turing machine), has two additional read-only tapes: the *valuation tape*, serving as a static input, and the *run tape*, serving as a dynamic input. The valuation tape contains a full description of the valuation, and its contents remain fixed throughout the work of the machine. As for the run tape, at any time it contains a full description of the current position, and is constantly updated by appending, to its contents, the new moves made by the players. Some of the states of the machine are designated as *move states*. Entering a move state results in making, by the machine, a move α, where α is the contents of a certain segment of the work tape.

To formally define hard-play machines, let us agree on some technical details, terminology and notation, most of which are standard. The computation proceeds in discrete *steps*, and we refer to a transfer from one step to another as a *transition*. It can help create a clear picture of the work of a machine if we assume that transitions happen instantaneously. The time interval between any two transitions (i.e. the time during which the machine remains in a given step), whose length we assume to be constant, is called a *clock cycle*.

Each of the three tapes of the machine is infinite in one (right) direction—has the beginning but no end. They are divided into (infinitely many) *cells*. Each tape has its own *scanning head*, which, at any step, scans one of the cells of the tape. On a transition, the machine can move the head to the next cell in either direction, or choose to let the head stay put. An attempt to move the head to the left when it is scanning the first cell of the tape has no effect—the head will remain in its location over the first cell. Each cell contains one of the finitely many symbols of the *tape alphabet*, where each tape may have its own alphabet. On the valuation tape and the run tape,

the head can only read the symbol from the scanned cell, while on the work tape it can both read and write.

A tape alphabet always includes the *blank symbol*, which we denote by $-$. A cell containing $-$ can be thought of as an empty cell, and an *empty tape* will mean a tape whose every cell contains $-$. The *contents of a tape* (or of a certain segment of it) can be thought of as a string over the alphabet of that tape, corresponding to the order in which the symbols appear in the tape cells. The *non-trivial part* of such contents is the initial segment of it up to the point beginning from which (to the right of which) all cells are empty. We will usually identify the contents of a tape with its non-trivial part, as the full contents, if needed, can be uniquely restored from it.

The following three alphabets $\mathscr{A}_V$, $\mathscr{A}_W$ and $\mathscr{A}_R$ will be used as tape alphabets for the valuation tape, work tape and run tape, respectively:

$$\mathscr{A}_V = \{0,\ldots,9,-\},$$
$$\mathscr{A}_W = \textbf{Move alphabet} \cup \{-\},$$
$$\mathscr{A}_R = \textbf{Move alphabet} \cup \{-,\top,\bot\}.$$

Next, we need to fix a way to represent ("*spell*") positions and valuations as tape contents, i.e. as (possibly infinite in case of valuations) strings over the tape alphabet—the alphabet $\mathscr{A}_R$ for positions, and the alphabet $\mathscr{A}_V$ for valuations. Here are our conventions:

The *spelling of a position* will be a string of the form $-\wp_1\alpha_1\wp_1\alpha_2\ldots\wp_1\alpha_n$, where the $\wp_i$ are elements of **Players**, and the α_i are elements of **Moves**. Such a string *spells* the position $\langle\wp_1\alpha_1,\wp_2\alpha_2,\ldots,\wp_n\alpha_n\rangle$. We will be using the expression R_Φ to denote the string that spells position Φ.

The *spelling of a valuation* will be an infinite string of the form $-c_0-c_1-c_2-\ldots$, where the c_i are elements of **Constants**. Such a string *spells* the valuation that assigns c_i to each $v_i \in \textbf{Variables} = \{v_0, v_1, v_2, \ldots\}$. We will be using the expression V_e to denote the string that spells valuation e.

Here comes a formal definition of HPM:

Definition 15.1. A *hard-play machine* (HPM) is a tuple (S, M, s, T), where:
- S is a finite set, whose elements are called *states*.
- M is a subset of S, whose elements are called *move states*.
- s is an element of S, called the *start state*.
- T, called the *transition function*, is a function of the type

$$S \times \mathscr{A}_V \times \mathscr{A}_W \times \mathscr{A}_R \to S \times \mathscr{A}_W \times \{-1,0,1\} \times \{-1,0,1\} \times \{-1,0,1\}.$$

Here the meaning of the equality $T(a,q_1,q_2,q_3) = (b,q_4,d_1,d_2,d_3)$, which will be more precisely explained in Definition 15.3, is that when the current state of the machine is a and the symbols scanned on the three (valuation, work and run) tapes are q_1,q_2,q_3, respectively, the next state of the machine will become b, the symbol q_2 will be replaced by the symbol q_4 on the work tape, and the three scanning heads will move in the directions d_1,d_2,d_3, respectively.

A configuration of an HPM, formally defined below, is a full description of a ("current") computation step of it, in terms of the current state, current locations of the three scanning heads, the non-trivial contents of the work tape, and the position spelled on the run tape. The contents of the valuation tape is not included in this description as it does not change from step to step.

Definition 15.2. Let $\mathscr{H}=(S,M,s,T)$ be an HPM.

1. A *configuration* of $\mathscr{H}$ is a tuple $C=(a,v,W,w,\Phi,r)$, where:
   - $a\in S$ (the *state component* of C),
   - $v\in\{1,2,\ldots\}$ (the *valuation tape head location component* of C),
   - W is a finite string over $\mathscr{A}_W$ (the *work tape contents component* of C),
   - $w\in\{1,2,\ldots\}$ (the *work tape head location component* of C),
   - Φ is a position (the *position component* of C),
   - $r\in\{1,2,\ldots\}$ (the *run tape head location component* of C).

2. The *initial configuration* of $\mathscr{H}$ is the configuration where the machine is in its start state, all three scanning heads are at the beginning of their tapes, the work tape is empty, and so is the run tape, thus spelling the position $\langle\rangle$. That is, this is the configuration $(s,1,\varepsilon,1,\langle\rangle,1)$, where ε, as always, stands for the empty string.

For a positive integer k and string K, we use $K(k)$ to denote the kth symbol of K. If here k is greater than the length of K, $K(k)$ is understood as the blank symbol.

Definition 15.3. Let e be any valuation, $\mathscr{H}=(S,M,s,T)$ a hard-play machine, Θ any finite (possibly empty) sequence of $\bot$-labeled moves, and $C=(a,v,W,w,\Phi,r)$ a configuration of $\mathscr{H}$ with $T(a,V_e(v),W(w),R_\Phi(r))=(b,q,d_V,d_W,d_R)$ (recall that V_e and R_Φ are, respectively, the strings that spell valuation e and position Φ). Then the (e,Θ)-*successor* of C in $\mathscr{H}$ is the configuration $C'=(a',v',W',w',\Phi',r')$, where:

- $a'=b$.
- $v'=v+d_V$, except when $v=1$ and $d_V=-1$, in which case $v'=1$; similarly for w',w,d_W and r',r,d_R.
- W' is the result of replacing by q the wth symbol in W.
- – If $a\notin M$, then $\Phi'=\langle\Phi,\Theta\rangle$.
  – If $a\in M$, then $\Phi'=\langle\Phi,\Theta,\top\alpha\rangle$, where α is the segment of W starting from (including) its wth symbol up to the first blank symbol.

When we simply say "*an e-successor* of C", we mean the (e,Θ)-successor of C for some arbitrary finite sequence Θ of $\bot$-labeled moves.

Thus, an e-successor C' of configuration C is the description of a next computation step that can legally follow the step described by C when e is spelled on the valuation tape. The state of the machine, locations of the three scanning heads and the contents of the work tape at that step are uniquely determined in the standard (standard for multitape Turing machines) way by C and the transition function of the machine. The

only non-determined parameter of configuration C' is the position Φ' spelled on the run tape. In particular, the latter will extend the position component Φ of C by adding any finite (possibly zero) number of moves made by $\bot$ since the previous transition. This corresponds to the understanding that there are no restrictions on the possible behavior of $\bot$ including its speed, except that $\bot$ can only make a finite number of moves per time unit. On top of (after) these $\bot$-labeled moves, Φ' will also include the move made by the machine if the latter made a move in configuration C. Technically, the machine makes a move by entering a move state, in which case the contents of the work tape from the current location of the read/write head up to the first blank cell is interpreted as the move the machine made.

Definition 15.4. Let $\mathscr{H}$ be an HPM and e a valuation.

1. An *e-computation branch* of $\mathscr{H}$ is an infinite sequence $C_1, C_2, C_3, \ldots$ of configurations such that C_1 is the initial configuration of $\mathscr{H}$ and, for each j, C_{j+1} is an e-successor of C_j in $\mathscr{H}$.
2. The *run spelled by such a branch* is the unique run Γ such that, for every initial segment Φ of Γ, Φ is an initial segment of the position component of some configuration C_j (as well as all C_k with $k > j$).
3. A *run generated by $\mathscr{H}$ on e* is the run spelled by some e-computation branch of $\mathscr{H}$. Simply a *run generated by $\mathscr{H}$* is a run generated by $\mathscr{H}$ on e for some arbitrary valuation e.

Thus, an e-computation branch is a possible sequence of configurations through which the machine will go with e spelled on its valuation tape. Which of the branches will be actually followed depends on the behavior of the adversary $\bot$ of the machine—in particular, on what moves, in what order and when (during which clock cycles of $\mathscr{H}$) $\bot$ makes. The set of possible computation branches captures the set of all possible behaviors of $\bot$.

Each computation branch B generates a run—the run spelled by B. This run is indeed spelled, incrementally, on the run tape of the machine as it goes from one configuration of B to another.

Definition 15.5. (1) We say that HPM $\mathscr{H}$ *wins game A on valuation e* iff, for every run Γ generated by $\mathscr{H}$ on e, we have $\mathbf{Wn}_e^A\langle\Gamma\rangle = \top$.

(2) We say that HPM $\mathscr{H}$ (simply) *wins game A* iff it wins A on every valuation.

16. The easy-play model

Generally, the hard-play model makes it really "hard" for $\top$ to win a game. It strongly favors $\bot$, whose advantage is not only that its possible behaviors (*strategies*) are not limited to algorithmic ones, but also that its relative *speed* is unlimited as well: during one clock cycle of the HPM against which the game is played, $\bot$ can make an arbitrary number of moves. As a result, only some special sort of

games can be winnable by HPMs—games where the speed of the players is of no importance. As we are going to see shortly, the class of such "special" games is very wide—in particular, all static games fall under this category. However, when considering dynamic (non-static) games, the relative speeds of the two players can play a crucial role. In that context, which can be of potential interest in the field of real-time computing, we may want to introduce limited-speed versions of plays. This can be done by including an additional *speed-limit parameter* in the hard-play model of the previous section—a parameter whose reciprocal establishes a minimal interval, in terms of clock cycles of the HPM, that should elapse between any two moves made by $\bot$, so that, say, the speed of 0.5 would mean that $\bot$ can make at most one move every two clock cycles, and the speed of 2 would mean that $\bot$ can make at most two moves per clock cycle. By varying the speed-limit parameter, we would get a strict linear hierarchy of winnability (= existence of a winning machine for a given game) concepts—the lower the relative speed of $\bot$, the higher the chances of a game to be winnable. The hard-play model of the previous section, where the speed-limit parameter was absent, can then be thought of as a special case of the limited-speed model with the speed limit of ω, which would put winnability in the sense of Definition 15.5 at the bottom (hardest-to-win) of the hierarchy of limited-speed winnabilities.

Of course, the minimal interval between $\bot$'s moves (the reciprocal of its speed limit) does not have to be constant and, in finer-level approaches, it can be understood as a function of the lengths of the pairs of $\bot$'s moves it separates.[9] This could open some new insights into complexity theory, where we could asymptotically analyze limited-speed winnability and talk about linear speed-limit winnability, polynomial speed-limit winnability, etc.

In the present paper, however, we are not going to try to elaborate this line, leaving it for the future to tackle. Instead of adding to the hard-play model a parameter whose instantiations would generate different degrees of speed limitation for $\bot$, we are simply going to introduce a new model where the idea of limiting the speed of $\bot$ is taken to the extreme. In this model, that we call the *easy-play model*, $\bot$ can make a move only when $\top$ explicitly allows it to do so. The only "fairness" requirement that $\top$ should satisfy is that every once in a while it should give $\bot$ a chance to move, even though how long this "while" lasts is totally up to $\top$. This full control over $\bot$'s speed means that $\top$ can select its own pace for the play, and obviously makes winning a game easier than the hard-play model with or without a speed-limit parameter.

In the easy-play model, $\top$ is formalized as a sort of Turing machine called an easy-play machine (EPM). An EPM is nothing but an HPM with an additional sort of states called permission states. $\bot$ can make a move only when the machine enters a permission state, which yields a corresponding change in the definition of the successor relation between configurations. Here is a formal definition of EPM:

[9] In order to determine when the first move can be made, a certain constant length for the imaginary "0th move", such as length 0, can be assumed.

Definition 16.1. An *easy-play machine* (*EPM*) is a tuple (S, M, P, s, T), where:
- S is a finite set, whose elements are called *states*.
- M and P are two disjoint subsets of S, whose elements are called *move states* and *permission states*, respectively.
- s is an element of S, called the *start state*.
- T, called the *transition function*, is a function of the type

$$S \times \mathscr{A}_V \times \mathscr{A}_W \times \mathscr{A}_R \to S \times \mathscr{A}_W \times \{-1,0,1\} \times \{-1,0,1\} \times \{-1,0,1\}.$$

The notions of a *configuration* and the *initial configuration* of an EPM are defined in exactly the same way as in the case of hard-play machines (Definition 15.2). The successor relation is redefined as follows:

Definition 16.2. Let e be a valuation, $\mathscr{E} = (S, M, P, s, T)$ an EPM and $C = (a, v, W, w, \Phi, r)$ a configuration of it, with $T(a, V_e(v), W(w), R_\Phi(r)) = (b, q, d_V, d_W, d_R)$. Then we say that a configuration $C' = (a', v', W', w', \Phi', r')$ is an *e-successor* of C in $\mathscr{E}$ iff:
- $a' = b$.
- $v' = v + d_V$ except when $v = 1$ and $d_V = -1$, in which case $v' = 1$; similarly for w', w, d_W and r', r, d_R.
- W' is the result of replacing by q the wth symbol in W.
- – If $s \notin M \cup P$, then $\Phi' = \Phi$.
  – If $s \in M$, then $\Phi' = \langle \Phi, \top\alpha \rangle$, where α is the segment of W starting from (including) its wth symbol up to the first blank symbol.
  – If $s \in P$, then either $\Phi' = \Phi$ or $\Phi' = \langle \Phi, \bot\alpha \rangle$ for some move α.

Observe that unless the state component of C is a permission state, C has exactly one e-successor in an EPM. On the other hand, each configuration of a hard-play machine has infinitely many e-successors.

We define an *e-computation branch* of a EPM, the *run spelled by such a branch* and a *run generated by an EPM* (with or without a valuation parameter) exactly as for hard-play machines, so that Definition 15.4 can be repeated here literally, only with "EPM" instead of "HPM".

As for the concept of winning a game by an EPM, it needs to incorporate the fairness condition mentioned earlier. Let us call an e-computation branch of an EPM *fair* iff there are infinitely many configurations in the branch whose state component is a permission state. And we will say that a given EPM is *fair* iff every e-computation branch (for every valuation e) of that machine is fair.

Definition 16.3. (1) We say that EPM $\mathscr{E}$ *wins game A on valuation e* iff, for every e-computation branch B of $\mathscr{E}$ and the run Γ spelled by B,
(a) $\mathbf{Wn}_e^A\langle\Gamma\rangle = \top$ and
(b) B is fair unless Γ is a ($\bot$-) illegal run of A w.r.t. e.

(2) We say that EPM $\mathscr{E}$ (simply) *wins game A* iff it wins A on every valuation.

Thus, in order to win, it is necessary for an EPM to generate a won run in a fair play, that is, with a fair branch; the only case when unfair play will be excused is the case when the environment makes an illegal move: making a (first) illegal move will result in an immediate loss for the environment, no matter what happens after that move. Hence:

Remark 16.4. When trying to show that a given EPM wins a given game on a given valuation, it is always perfectly safe to assume that the environment never makes an illegal move, for if it does, the machine automatically wins (unless the machine itself has made an illegal move earlier, in which case it does not matter what the environment did afterwards anyway, so that we may still assume that the environment did not make any illegal moves).

Henceforth such an assumption will always be implicitly present in our EPM-winnability proofs.

The following lemma establishes that in the easy-play model it is really "easier" (well, more precisely, not harder) to win whatever game than in the hard-play model:

Lemma 16.5. *Suppose A is any (not necessarily static) game and there is an HPM that wins A. Then there is an EPM—in particular, a fair EPM—that wins A.*

Moreover, there is an effective procedure that converts any HPM $\mathscr{H}$ into a fair EPM $\mathscr{E}$ such that, for any valuation e and any game A, $\mathscr{E}$ wins A on e whenever $\mathscr{H}$ does so.

Proof. The idea is to construct, for a given HPM $\mathscr{H}$, an EPM $\mathscr{E}$ which imitates $\mathscr{H}$ in a way that guarantees that each run generated by $\mathscr{E}$ on an arbitrary given valuation is also a run generated by $\mathscr{H}$ on the same valuation.

A straightforward way to imitate $\mathscr{H}$ would be to make $\mathscr{E}$ no different from $\mathscr{H}$—after all, every HPM can be thought of as an EPM with an empty set of permission states. However, such an EPM would not be fair and hence would never win a play. So, what we need to do is to somehow ensure that every once in a while $\mathscr{E}$ enters a permission state, and otherwise it follows the steps of $\mathscr{H}$. One of the ways to achieve the purpose of imitating $\mathscr{H}$ while playing fair is the following.

Let $\mathscr{H}=(S,M,s,T)$ be the HPM for which we are constructing an easy-play counterpart $\mathscr{E}$. We define $\mathscr{E}=(S_{\mathscr{E}},M_{\mathscr{E}},P_{\mathscr{E}},s_{\mathscr{E}},T_{\mathscr{E}})$ as follows:

- For each state $a\in S$, we create a unique state a° with $a^\circ\notin S$, called the *prelude* of a, and define $P_{\mathscr{E}}$ as the set of all those prelude states. That is, $P_{\mathscr{E}}=\{a^\circ \mid a\in S\}$.
- $S_{\mathscr{E}}=S\cup P_{\mathscr{E}}$; $M_{\mathscr{E}}=M$; $s_{\mathscr{E}}=s^\circ$.
- Let $(a,q_1,q_2,q_3)\in S\times\mathscr{A}_V\times\mathscr{A}_W\times\mathscr{A}_R$, with $T(a,q_1,q_2,q_3)=(b,q,d_W,d_V,d_R)$. We define $T_{\mathscr{E}}$ for the two tuples (a°,q_1,q_2,q_3) and (a,q_1,q_2,q_3) as follows:
  - $T_{\mathscr{E}}(a^\circ,q_1,q_2,q_3)=(a,q_2,0,0,0)$;
  - $T_{\mathscr{E}}(a,q_1,q_2,q_3)=(b^\circ,q,d_V,d_W,d_R)$.

Thus, from each prelude state a°, $\mathscr{E}$ goes to a, without changing the contents of the work tape and without moving the scanning heads; then, in state a, it does exactly what $\mathscr{H}$ would do, only, instead of going to the state b to which $\mathscr{H}$ would go, $\mathscr{E}$ goes to the prelude of b. The effect achieved this way is that every other clock cycle $\mathscr{E}$ goes through a permission (prelude) state which makes $\mathscr{E}$ fair, and otherwise it "acts exactly like $\mathscr{H}$".

Suppose $\mathscr{E}$ does not win a game A on valuation e. This means that, for the run Γ spelled by some e-computation branch $B_{\mathscr{E}} = C_1, C_2, C_3, \ldots$ of $\mathscr{E}$, we have $\mathbf{Wn}_e^A\langle\Gamma\rangle = \bot$. A little analysis of the situation can show us that then the sequence $B_{\mathscr{H}} = C_2, C_4, C_6, \ldots$ is an e-computation branch of $\mathscr{H}$ that spells the same run Γ as $B_{\mathscr{E}}$ does. Hence, $\mathscr{H}$ does not win A on e, either. To complete the proof of the lemma, it remains to make the straightforward observation that the way we constructed $\mathscr{E}$ from $\mathscr{H}$ was effective. □

17. Equivalence of the hard- and easy-play models for static games

Vice versa to Lemma 16.5, in a general case, does not hold, which means that winning in the easy-play model is "properly easier" than in the hard-play model. A simple example is the game F discussed in Section 4 that has only two non-empty legal runs: $\langle\top 0\rangle$, won by $\top$, and $\langle\bot 0\rangle$, won by $\bot$. Obviously, this game will be won by an EPM that makes the move 0 (without entering a permission state) and then goes into an infinite loop in a permission state. On the other hand no HPM can win this game, for during the very first clock cycle, $\bot$ can make the move 0, after which there is nothing that $\top$ can do to win.

The above game F, however, is not static. Is there a static game that is winnable by an EPM but not by an HPM? The following Lemma 17.1 contains a negative answer to this question. Just as with medical tests, a negative answer here means positive news. Lemma 17.1 establishes a nice property of static games, justifying their name "static". According to it, for static games speed cannot be an issue: even when playing as an HPM, the player can still pretend that it is playing as an EPM, which means that it can select its own pace for the play and think peacefully only about *what* moves to make without any concerns about *how fast* to make them. No pressure for time and no rush.

Lemma 17.1. *Suppose A is a static game and there is an EPM that wins A. Then there is an HPM that wins A.*

Moreover, there is an effective procedure that converts any EPM $\mathscr{E}$ into an HPM $\mathscr{H}$ such that for every static game A and valuation e, $\mathscr{H}$ wins A on e whenever $\mathscr{E}$ does so.

In the proof of this lemma, unlike the proof of Lemma 16.5, we will no longer be able to afford the luxury of giving a full formal description of the machine. Instead, we will have to be satisfied with a high-level description of how it works. In this sort of descriptions, used many times not only here but also in later sections, we will

often assume that the machine maintains, on its work tape, certain records (variables), usually containing the encodings of some high-level objects such as configurations of other machines or positions. If R is such a record and R' is an admissible value for it, the expression $R := R'$, following the standard notational practice used in codes or pseudocodes, will stand for the operation of assigning the value R' to R, i.e. overwriting the current contents of R by R'. Our descriptions will also be using the constructs "**do**", "**if ... then**", "**while**" etc. with their standard meanings.

Proof of Lemma 17.1. Let $\mathscr{E} = (S, M, P, s, T)$ be an EPM. We need to show how to construct an HPM $\mathscr{H}$ that wins the same static games as $\mathscr{E}$ does. Here is an informal description of such a machine. $\mathscr{H}$ maintains two records: Υ and C. At any time, C contains (the encoding of) a certain configuration (a, v, W, w, R, r) of machine $\mathscr{E}$, and Υ contains (the encoding of) either $\langle\rangle$ or $\langle\bot\alpha\rangle$ for some move α. These two variables are updated by the routines TRANSITION and FETCH, respectively.

Let us first describe FETCH. It is only during this routine when $\mathscr{H}$ may change the location of the read head on its run tape, which it only moves in the rightward direction. What FETCH does is that it reads the next (leftmost) $\bot$-labeled move $\bot\alpha$ on the run tape that the machine has not read yet, and puts $\langle\bot\alpha\rangle$ in Υ. If there is no such move in the current position, then it puts $\langle\rangle$ in Υ. Here is a little more precise description of this procedure:

Procedure FETCH. Beginning from the current cell scanned by the run tape head, read run tape cells, in the left-to-right direction, until you see either $-$ or $\bot$. If you see $-$, stop there and **do** $\Upsilon := \langle\rangle$; and if you see $\bot$, continue moving right—remembering the symbols you see on your way—till you encounter $-$, $\top$ or $\bot$, stop there, and **do** $\Upsilon := \langle\bot\alpha\rangle$, where $\bot\alpha$ is the $\bot$-labeled move that you have just finished reading. An exception is when FETCH is performed the very first time. In this case the above actions should be preceded by moving the run tape head one cell to the right to skip the first cell of the tape which is always blank.

Now about TRANSITION. Let e be the valuation spelled on the valuation tape of $\mathscr{H}$. The purpose of TRANSITION is to replace the configuration (a, v, W, w, Φ, r) stored in C by a certain e-successor $(a', v', W', w', \Phi', r')$ (in $\mathscr{E}$) of this configuration and, in addition, if $a \in M$, to make the same move as $\mathscr{E}$ makes in configuration (a, v, W, w, Φ, r). When $a \notin P$, the configuration $(a', v', W', w', \Phi', r')$ to which TRANSITION needs to update C is uniquely determined by (a, v, W, w, Φ, r), the valuation e and the description of $\mathscr{E}$, and TRANSITION calculates it using e (spelled on its valuation tape) and its built-in (full, finite) knowledge of $\mathscr{E}$. If $a \in P$, then the parameter Φ' is no longer uniquely determined (while a', v', W', w', r' still are). To select Φ', TRANSITION calls FETCH, and sets Φ' to be the result of appending, to Φ, the ($\leqslant 1$-length) position that FETCH returns in Υ. Here is a more precise description of this procedure:

Procedure TRANSITION. Let e be the valuation spelled on the valuation tape (of $\mathscr{H}$), and (a, v, W, w, Φ, r)—the current contents of C. Act depending on which of the following three cases applies:

Case of $a \notin M \cup P$: Calculate the (unique) e-successor C' of C in $\mathscr{E}$, and **do** $C := C'$.

Case of $a \in M$:
1. Find the move α that $\mathscr{E}$ makes in configuration C, and make the move α;
2. Calculate the (unique) e-successor C' of C in $\mathscr{E}$, and **do** $C := C'$.

Case of $a \in P$:
1. **Do** FETCH;
2. Calculate the (unique) e-successor $C' = (a', v', W', w', \Phi', r')$ of (a, v, W, w, Φ, r) in $\mathscr{E}$ with $\Phi' = \langle \Phi, \Upsilon \rangle$, and **do** $C := C'$.

Now, the entire work of $\mathscr{H}$ can be described by the following interactive algorithm:

1. Create records Υ and C, initialized to $\langle\rangle$ and the initial configuration of $\mathscr{E}$, respectively.
2. Loop forever doing TRANSITION.

Our claim is that for any static game A and valuation e, if $\mathscr{E}$ wins A on e, then so does $\mathscr{H}$. To prove this, suppose $\mathscr{H}$ does not win a certain static game A on a certain valuation e. Fix these A and e. We want to show that $\mathscr{E}$ does not win A on e, either.

That $\mathscr{H}$ does not win A on e means that there is an e-computation branch $B_{\mathscr{H}}$ of $\mathscr{H}$, which spells a run $\Gamma_{\mathscr{H}}$, such that $\mathbf{Wn}_e^A\langle\Gamma_{\mathscr{H}}\rangle = \bot$. Let us fix these $B_{\mathscr{H}}$ and $\Gamma_{\mathscr{H}}$ for the rest of the proof.

Consider how $\mathscr{H}$ has worked along branch $B_{\mathscr{H}}$. After initializing C and Υ, the work of $\mathscr{H}$ consisted in iterating TRANSITION infinitely many times. Each iteration updates the value of C. Let then

$$B_{\mathscr{E}} = C_1, C_2, C_3, \ldots$$

be the sequence of values that the variable C goes through. That is, for any $n \geqslant 1$, C_n is the value of C at the beginning of the nth iteration of TRANSITION, and C_{n+1} is the value of C by the end of that iteration. From the description of $\mathscr{H}$ it is clear that $B_{\mathscr{E}}$ is an e-computation branch of $\mathscr{E}$. Let us fix $\Gamma_{\mathscr{E}}$ as the run spelled by $B_{\mathscr{E}}$.

Statement 1. *The subsequence of $\top$-labeled moves of $\Gamma_{\mathscr{H}}$ is the same as that of $\Gamma_{\mathscr{E}}$.*

Proof. Each $\top$-labeled move of $\Gamma_{\mathscr{E}}$ is made by $\mathscr{E}$ in some configuration C_n of $B_{\mathscr{E}}$. The same move will be made by $\mathscr{H}$ during the nth iteration of TRANSITION. And vice versa: every $\top$-labeled move of $\Gamma_{\mathscr{H}}$ is made by $\mathscr{H}$ during some nth iteration of TRANSITION, where (because) the same move is made by $\mathscr{E}$ in configuration C_n. So, both machines made exactly the same ($\top$-labeled) moves, and obviously in the same order. □

Statement 2. (a) *The subsequence of $\bot$-labeled moves of $\Gamma_{\mathscr{E}}$ is an initial segment of that of $\Gamma_{\mathscr{H}}$.*

(b) *If $B_{\mathscr{E}}$ is fair, then the subsequence of $\bot$-labeled moves of $\Gamma_{\mathscr{E}}$ is the same as that of $\Gamma_{\mathscr{H}}$.*

Proof. Let $\bot\alpha_1, \bot\alpha_2, \bot\alpha_3, \ldots$ be the subsequence of $\bot$-labeled moves of $\Gamma_{\mathscr{H}}$. These labmoves are read and copied to Υ, one at a time and in the order of their occurrence in the above sequence, by the routine FETCH. Each such read of $\bot\alpha_i$ occurs during some nth iteration of TRANSITION such that the state component of C_n is a permission state,

and then $\bot\alpha_i$ is appended to the position component of C_n when updating it to C_{n+1}. As this is the only way how $\bot$-labeled moves emerge in $\Gamma_{\mathscr{E}}$, all the $\bot$-labeled moves of $\Gamma_{\mathscr{E}}$ are among $\bot\alpha_1, \bot\alpha_2, \ldots$, in exactly this order, so that the subsequence of $\bot$-labeled moves of $\Gamma_{\mathscr{E}}$ is (at least) an initial segment of $\bot\alpha_1, \bot\alpha_2, \ldots$. What remains to verify is that, when $B_{\mathscr{E}}$ is fair, this initial segment is not proper, that is, that each of the $\bot\alpha_i$ appears in $\Gamma_{\mathscr{E}}$. Assume $B_{\mathscr{E}}$ is fair, so that there are infinitely many configurations among $C_1, C_2, \ldots$ whose state component is a permission state. For every such configuration C_n, the nth iteration of TRANSITION calls FETCH that will try to read the next yet unread $\bot$-labeled move on the run tape of $\mathscr{H}$. That is, FETCH will be called infinitely many times, which means that every $\bot$-labeled move of $\Gamma_{\mathscr{H}}$ will be read sooner or later—on some nth iteration of TRANSITION, and added to the position component of C_n when updating C_n to C_{n+1}. □

Statement 3. *If the* ith $\bot$*-labeled move is made earlier than the* jth $\top$*-labeled move in* $\Gamma_{\mathscr{E}}$*, then so is it in* $\Gamma_{\mathscr{H}}$.

Proof. Suppose the ith $\bot$-labeled move $\bot\alpha$ is made earlier than the jth $\top$-labeled move $\top\beta$ in $\Gamma_{\mathscr{E}}$. $\mathscr{H}$ must have added $\bot\alpha$ and $\top\beta$ to the simulated run tape contents of $\mathscr{E}$ (the position component of C) during certain kth and mth iterations of TRANSITION, respectively, with $k < m$. This means that during the kth iteration, FETCH has read $\bot\alpha$, so that $\bot\alpha$ was already on the run tape of $\mathscr{H}$. But $\top\beta$ was not there at that time—this move was made by $\mathscr{H}$ only during the mth iteration, so that $\top\beta$ would be appended, at that time, at the end of the position spelled on the run tape of $\mathscr{H}$ which already contained $\bot\alpha$. □

To finish the proof of the lemma, suppose $B_{\mathscr{E}}$ is fair. Then Statements 1, 2b and 3 mean nothing but that $\Gamma_{\mathscr{H}}$ is a $\top$-delay of $\Gamma_{\mathscr{E}}$. Hence, since A is static and $\mathbf{Wn}_e^A\langle\Gamma_{\mathscr{H}}\rangle \neq \top$, we must have $\mathbf{Wn}_e^A\langle\Gamma_{\mathscr{E}}\rangle \neq \top$. The latter means that $\mathscr{E}$ does not win A on e.

Now consider the case when $B_{\mathscr{E}}$ is not fair and assume, for a contradiction, that $\mathscr{E}$ wins A on e. Then $\Gamma_{\mathscr{E}}$ must be a $\bot$-illegal run of $e[A]$. Let n be the number of $\bot$-labeled moves in the shortest illegal initial segment of $\Gamma_{\mathscr{E}}$. Let $\Gamma'_{\mathscr{E}}$ and $\Gamma'_{\mathscr{H}}$ be the results of deleting in $\Gamma_{\mathscr{E}}$ and $\Gamma_{\mathscr{H}}$ (respectively) all the $\bot$-labeled moves except the first n $\bot$-labeled moves. Then Statements 1, 2a and 3 imply that $\Gamma'_{\mathscr{H}}$ is a $\top$-delay of $\Gamma'_{\mathscr{E}}$. The latter is obviously $\bot$-illegal and consequently, by Lemma 4.7.2, so is the former. Let then Θ be the shortest $\bot$-illegal initial segment of $\Gamma'_{\mathscr{H}}$. Notice that Θ is also an initial segment of $\Gamma_{\mathscr{H}}$, which makes $\Gamma_{\mathscr{H}}$ a $\bot$-illegal and hence $\top$-won run of $e[A]$, contrary to our assumption.

This, together with the straightforward observation that the way we constructed $\mathscr{H}$ from $\mathscr{E}$ was effective, completes the proof of Lemma 17.1. □

The following theorem is just a combination of Lemmas 16.5 (restricted to static games) and 17.1, put in one place for convenience of future references:

Theorem 17.2. *For any static game* A*, the following statements are equivalent*:
(i) *there is an EPM that wins* A;

(ii) *there is an HPM that wins A*;
(iii) *there is a fair EPM that wins A*.

Moreover, there is an effective procedure that converts any EPM (*resp. HPM*) $\mathscr{M}$ *into an HPM* (*resp. fair EPM*) $\mathscr{N}$ *such that, for every static game A and valuation e,* $\mathscr{N}$ *wins A on e whenever* $\mathscr{M}$ *does so*.

18. Algorithmic vs. non-algorithmic winnability

In this paper our interests are focused on static games. As we saw in Theorem 17.2, as long as static games are concerned, the hard and easy-play models are equivalent. Why did we need then to introduce those two different models in the first place? There are both philosophical and technical reasons. Out of the two models, only the hard-play model may seem natural and intuitively convincing, as it is directly derived from the fact of life that the environment ($\bot$) can be arbitrarily fast, and the agent ($\top$) has no control over the actions (moves) of the environment. However, this model is technically very inconvenient to use when we want to show that a certain game can be won, and we will be much better off using the easy-play model for describing winning strategies. On the other hand, when we want to show that a certain game is impossible to win, we will find that the hard-play model is more handy, as describing $\bot$'s (counter)strategies for the hard-play model is much easier. In particular, as we are going to see this in Section 20, when $\top$ plays as an HPM, $\bot$'s strategies can be described in terms of EPMs, which makes the easy-play model a natural complement of the hard-play model.

Anyway, it can be a little confusing to have two or even three names (winnability by an HPM, winnability by an EPM, winnability by a fair EPM) for the same child. The following definition establishes one name:

Definition 18.1. Let A be a static game. We say that A is *winnable* iff there is an HPM that wins A; equivalently, A is winnable iff there is an EPM that wins A; equivalently, A is winnable iff there is a fair EPM that wins A.

The following statement, in conjunction with Thesis 4.4, presents a generalization of the Church–Turing thesis to interactive problems:

Thesis 18.2. *The concept of winnability is an adequate formal counterpart of our intuitive notion of computability, i.e. algorithmic solvability of* (*interactive*) *computational problems.*

In other words, what can be called *interactive algorithms* are nothing but our play machines. Whatever we would perceive as an interactive algorithm, can eventually be formalized as a play machine, and vice versa: every play machine does nothing but following some interactive algorithm.

An ordinary, non-interactive algorithm receives an input at the very beginning of its work, and the work of the algorithm ends with producing an output. As for interactive

algorithms, both their input ($\bot$'s moves) and output ($\top$'s moves) can be multiple and dynamic, and they typically neither start nor end the procedure, but rather they are spread throughout it.

As we will usually describe interactive algorithms in terms of EPMs rather than HPMs, it would be convenient to agree on some jargon frequently used in high-level descriptions of EPMs. In particular, we assume that dynamic input an output takes place only through the routines called "grant permission", and "make the move...", respectively. Any other steps of interactive algorithms are no different from what ordinary algorithms do.

When performing the routine "*make the move* α", the EPM does the following: it constructs the move α on its work tape, delimits its right end with the blank symbol, then goes to the beginning of string α and enters a move state. The effect, indeed, is making the move α. We always assume that it is only during this routine that the machine enters a move state.

And when performing the routine "*grant permission*", the EPM does the following: it moves the read head of the run tape to the first (leftmost) non-initial blank cell, then enters a permission state once and immediately exits it, after which it checks what move (if any) the environment made in response. We always assume that it is only during this routine that the machine enters a permission state.

To see, once again but more formally, why our approach is indeed a generalization of the traditional approach to computability, let us revisit the traditional concepts of decidability and computability, as well as (in the next section) reducibility. These properties are usually defined only for what we can call *simple problems*—problems of one of the two types: (1) the problem of telling ('deciding') whether a certain fixed predicate is true for arbitrary given objects, or (2) the problem of finding ('computing') the value of a certain fixed function for arbitrary given objects. For known reasons, there is no essential difference between these two types of problems and one can be replaced by the other in all reasonable contexts, so let us only talk about simple problems of type (1) and hence what is traditionally called 'decidability' rather than 'computability'. Also, we may assume that the "fixed predicates" in such problems are always 1-ary (unary), with x as the only variable on which they depend. We will be identifying such a predicate $Q(x)$ with the simple problem itself, so that by "simple problem $Q(x)$" we will mean the problem of telling whether $Q(c)$ is true or not for a given constant c. It should be noted though that using the term "problem" when referring to this $Q(x)$ is somewhat inconsistent with our terminology. We agreed that a problem is nothing but a (static) game. What we call here "simple problem $Q(x)$", however, is not really the (elementary) game $Q(x)$, but rather the game $\sqcap x(Q(x) \sqcup \neg Q(x))$.

According to the standard definition of decidability, a simple problem $Q(x)$ is decidable iff there is a Turing machine (TM) that, for any input $c \in$ **Constants**, will sooner or later correctly tell us (by entering an accept or a reject state) whether $Q(c)$ is true or false. It is not hard to convert a TM that solves simple problem $Q(x)$ into an EPM that wins $\sqcap x(Q(x) \sqcup \neg Q(x))$, and vice versa. Say, to convert a TM into a corresponding EPM, we should make the EPM keep granting permission at the beginning until $\bot$ makes a move c ($c \in$ **Constants**), after which the machine should work exactly as the TM would work with the initial input c on its work tape, only, at the end, instead of

entering an accept or a reject state, make the move 1 or 2, respectively, and then go into an infinite loop in a permission state to formally guarantee fairness. And to covert an EPM into a corresponding TM, we can make the TM read its input c, then simulate the EPM in a play where, after the EPM (first) grants permission, $\bot$ responds by the move c; then, the TM should accept or reject depending on whether the EPM replies by the move 1 or 2. Thus, we have:

Proposition 18.3. *A simple problem $Q(x)$ is decidable iff $\sqcap x(Q(x) \sqcup \neg Q(x))$ is winnable.*

In a similar manner, we could obviously show that computability of a function f means nothing but winnability of $\sqcap x \sqcup y(y = f(x))$. Based on these facts, winnability, as noted before, indeed presents a generalization of decidability/computability from simple problems to any sort of computational problems (static games). Our concept of play machines, in turn, is a generalization of the concept of Turing machines: while Turing machines only solve (or fail to do so) simple problems, play machines solve (or fail to do so) any sort of interactive problems.

Just like the ordinary Turing machines, our hard- and easy-play machines have a number of natural variations, including variants with multiple or multidimensional work tapes, multiple scanning heads on each tape, etc.; to these should be added variations specific to our (hard- or easy-) play machines, such as: forbidding for the read heads of the valuation tape and the run tape to ever move in the leftward direction; machines with a separate, write-only tape where (instead of the work tape) the machine constructs the moves it makes; machines with only one move and/or permission state; machines where the run tape only shows $\bot$-labeled moves; machines where the always-illegal move $\spadesuit$ is physically impossible to make by either player; models where $\bot$ never makes illegal moves; models where $\bot$'s moves may be arriving (over a slow network) with some unknown delays, or where $\top$'s moves may appear on the run tape with delays; models that directly implement the distributed arbitration scenario of Example 4.3; etc. Even if with possibly different degrees of efficiency, all of these variations would still yield the same class of winnable static games.

Another sort of a variation of ordinary Turing machines is oracle Turing machines which, unlike most other variations, increases the computational power of the machines and, in fact, makes them as powerful as God as long as there is no restriction on what kind of oracles are allowed. It is worthwhile to consider the oracle versions of our play machines. We will only give a semiformal definition of this sort of machines to save space. Turning it into a strict formal definition would not present any problem.

Definition 18.4. Let $R(x_1,\ldots,x_n)$ be a (possibly infinitary) predicate with an attached tuple of variables. An *HPM* (resp. *EPM*) *with an oracle for* $R(x_1,\ldots,x_n)$ is an HPM (resp. EPM) that has the additional capability to query a device, called an *oracle for* $R(x_1,\ldots,x_n)$, regarding any particular n-tuple $c_1,\ldots,c_n$ of constants. To such a query the oracle instantaneously responds by "yes" or "no", depending on whether $R(c_1,\ldots,c_n)$ is true at e or not, where e is the valuation spelled on the valuation tape.

Technically, this sort of machines can be obtained, for example, by adding to the ordinary HPM (EPM) an additional sort of states called *oracle states*; the machine consults the oracle regarding whether $R(c_1,\ldots,c_n)$ is true by putting, at the end of the non-trivial part of the tape, a string that spells or encodes the tuple $c_1,\ldots,c_n$, and then entering an oracle state while scanning the first symbol of that string. This results in that symbol (or the whole string) automatically getting replaced by 1 or 0, depending on whether $R(c_1,\ldots,c_n)$ is true at e or not.

When we simply say "an oracle HPM (EPM)", we mean an HPM (EPM) with an oracle for some arbitrary predicate $R(\vec{x})$.

Even though, formally, only predicates can be used in oracles, it is perfectly safe to include, in high-level descriptions of oracle play machines, steps that query an oracle regarding the value of a given function (rather than predicate) at given arguments. For example, querying the oracle regarding the value of function $f(x)$ at c and receiving the answer c' can be understood as a series of queries regarding the truth of the predicate $f(x)=y$: asking the oracle whether $f(c)=0$ is true, then asking whether $f(c)=1$ is true, etc. until receiving a positive answer for $f(c)=c'$.

The notions such as configuration, computation branch, winning a game, etc. naturally extend to the oracle versions of play machines, and there is no need to redefine them here. Similarly, our result regarding the equivalence of hard- and easy-play machines for static games extends to the oracle versions of these machines:

Theorem 18.5. *For any static game A and predicate $R(\vec{x})$, the following statements are equivalent*:

(i) *there is an EPM with an oracle for $R(\vec{x})$ that wins A*;
(ii) *there is an HPM with an oracle for $R(\vec{x})$ that wins A*;
(iii) *there is a fair EPM with an oracle for $R(\vec{x})$ that wins A.*

Moreover, there is an effective procedure that, for whatever predicate $R(\vec{x})$, converts any EPM (resp. HPM) $\mathcal{M}$ with an oracle for $R(\vec{x})$ into an HPM (resp. fair EPM) $\mathcal{N}$ with an oracle for $R(\vec{x})$ such that, for every static game A and valuation e, $\mathcal{N}$ wins A on e whenever $\mathcal{M}$ does so.

Proof. The proof of Theorem 17.2 can be applied here with hardly any substantial modifications. The main technical idea used in that proof was letting one machine imitate or simulate the other. No problem to do the same with oracle machines as well: every step of one machine, including querying the oracle, can be simulated by the other machine as the two machines are using oracles for the same $R(\vec{x})$. □

Definition 18.6. Let A be a static game.

1. For a predicate $R(\vec{x})$, we say that A is *winnable with an oracle for $R(\vec{x})$* iff there is an HPM with an oracle for $R(\vec{x})$ that wins A; equivalently, A is winnable with an oracle for $R(\vec{x})$ iff there is an EPM with an oracle for $R(\vec{x})$ that wins A; equivalently,

A is winnable with an oracle for $R(\vec{x})$ iff there is a fair EPM with an oracle for $R(\vec{x})$ that wins A.

2. We say that A is *positive* (resp. *negative*) iff there is a predicate $R(\vec{x})$ such that A (resp. $\neg A$) is winnable with an oracle for $R(\vec{x})$.
3. We say that A is *determined* iff every instance of A is either positive or negative; otherwise A is *indetermined.*

One can show that a static game A is positive (or negative) if and only if all of its instances are so. This fact does not immediately follow from the definitions though.

Philosophically speaking, winnability of a static game means existence of $\top$'s algorithmic winning strategy for (every instance of) that game, while positiveness means existence of *any* (not necessarily algorithmic) winning strategy: since the definition of positiveness does not restrict the strength of an oracle that can be employed, as long as a winning strategy exists, the (easy-play) machine could use an oracle that knows how that winning strategy would act in a given situation, and then act exactly as the winning strategy would. Symmetrically, negativeness means existence of $\bot$'s winning strategy. As for determinacy of a given static game, it means that for every instance of the game one of the players has a (not necessarily algorithmic) winning strategy.

The existence of indetermined games is a non-trivial fact and its proof relies on the axiom of choice. This fact—for constant strict games—was first discovered by Gale and Stewart [9] and then more systematically analyzed by Blass [4,5] who heavily exploited it in completeness proofs. Such games are always infinite-depth (Theorem 18.7) and present highly abstract mathematical objects. Our focus is on winnability rather than positiveness due to the constructive character/computational meaning of the former and direct applicability in computer science or other spheres of real life. As conjectured in the present paper (Conjecture 24.4) and illustrated in [12], considering winnability instead of positiveness eliminates the need to ever appeal to indetermined games in completeness proofs, which makes such proofs far more informative than if they were based on indetermined counterexamples.

According to a known theorem due to Zermelo, what we would call perifinite-depth strict constant games are always determined. But our static games are generally neither strict nor constant, so this theorem needs to be proven all over again.

Theorem 18.7. *Every perifinite-depth static game is determined.*

Proof. Assume A is a perifinite-depth static game. To prove that A is determined, we need to show that every instance of A is either positive or negative. So, to simplify things, we may assume that A is already instantiated, i.e. is a constant game. This makes valuations irrelevant and the valuation parameter can be safely omitted throughout our analysis. We define two unary functions f and g from positions to moves as follows:

$$f\langle\Phi\rangle = \begin{cases} \alpha \text{ if } \langle\Phi,\top\alpha\rangle \in \mathbf{LR}^A \text{ and } \langle\Phi,\top\alpha\rangle A \text{ is positive} \\ \quad \text{(if there are many such } \alpha\text{, select the lexicographically} \\ \quad\quad \text{smallest one);} \\ \spadesuit \text{ if such a move } \alpha \text{ does not exist.} \end{cases}$$

$$g\langle\Phi\rangle = \begin{cases} \alpha \text{ if } \langle\Phi,\bot\alpha\rangle \in \mathrm{LR}^A \text{ and } \langle\Phi,\bot\alpha\rangle A \text{ is negative} \\ \quad \text{(if there are many such } \alpha\text{, select the lexicographically} \\ \quad\quad \text{smallest one)}; \\ \spadesuit \text{ if such a move } \alpha \text{ does not exist.} \end{cases}$$

Of course, with some appropriate encoding for positions and moves in mind, f and g can be thought of as functions of the type **Constants** $\rightarrow$ **Constants**. Now, for each position Φ, we recursively define two non-effective procedures $\mathscr{F}\langle\Phi\rangle$ and $\mathscr{G}\langle\Phi\rangle$. $\mathscr{F}$ uses an oracle for f and $\mathscr{G}$ uses an oracle for g as this can be understood from the following descriptions.

Procedure $\mathscr{F}\langle\Phi\rangle$. **if** $f\langle\Phi\rangle = \alpha \neq \spadesuit$, **then** make the move α and then **do** $\mathscr{F}\langle\Phi,\top\alpha\rangle$; **else** keep granting permission until the adversary makes a move α, after which **do** $\mathscr{F}\langle\Phi,\bot\alpha\rangle$.

Procedure $\mathscr{G}\langle\Phi\rangle$. **if** $g\langle\Phi\rangle = \alpha \neq \spadesuit$, **then** make the move α and then **do** $\mathscr{G}\langle\Phi,\bot\alpha\rangle$; **else** keep granting permission until the adversary makes a move α, after which **do** $\mathscr{G}\langle\Phi,\top\alpha\rangle$.

We will identify $\mathscr{F}\langle\Phi\rangle$ with an oracle EPM that follows this strategy; similarly for $\mathscr{G}\langle\Phi\rangle$. Notice that these two strategies/machines never read the valuation tape, so that their behavior will be the same no matter what that tape spells, and it is safe to just say "wins" instead of "wins on valuation e'". To complete the proof of the theorem, it would be sufficient to show that

$$\textit{For every } \Phi \in \mathrm{LR}^A, \textit{ either } \mathscr{F}\langle\Phi\rangle \textit{ wins } \langle\Phi\rangle A, \textit{ or } \mathscr{G}\langle\Phi\rangle \textit{ wins } \neg(\langle\Phi\rangle A). \qquad (18)$$

This statement can be proven by transfinite induction on the depth $|\langle\Phi\rangle A|$ of $\langle\Phi\rangle A$ (see Section 8). Our proof silently relies on a fact formally established only later in Proposition 20.6, according to which no static game can be both positive and negative. Pick an arbitrary $\Phi \in \mathrm{LR}^A$. There are three cases to consider:

Case 1: $f\langle\Phi\rangle = \alpha \neq \spadesuit$. This means that $\langle\Phi,\top\alpha\rangle \in \mathrm{LR}^A$ and $\langle\Phi,\top\alpha\rangle A$ is positive. We have $|\langle\Phi,\top\alpha\rangle A| < |\langle\Phi\rangle A|$. Hence, by the induction hypothesis, either (a) $\mathscr{F}\langle\Phi,\top\alpha\rangle$ wins $\langle\Phi,\top\alpha\rangle A$, or (b) $\mathscr{G}\langle\Phi,\top\alpha\rangle$ wins $\neg(\langle\Phi,\top\alpha\rangle A)$. (b) is impossible because then $\langle\Phi,\top\alpha\rangle A$ would be negative and hence not positive. Thus, $\mathscr{F}\langle\Phi,\top\alpha\rangle$ wins game $\langle\Phi,\top\alpha\rangle A$, i.e. game $\langle\top\alpha\rangle\langle\Phi\rangle A$. Then machine $\mathscr{F}\langle\Phi\rangle$, that does nothing but makes the move α and plays the rest of the game as $\mathscr{F}\langle\Phi,\top\alpha\rangle$, obviously wins $\langle\Phi\rangle A$.

Case 2: $g\langle\Phi\rangle = \alpha \neq \spadesuit$. This means that $\langle\Phi,\bot\alpha\rangle \in \mathrm{LR}^A$ and $\langle\Phi,\bot\alpha\rangle A$ is negative. We have $|\langle\Phi,\bot\alpha\rangle A| < |\langle\Phi\rangle A|$. Hence, by the induction hypothesis, either (a) $\mathscr{F}\langle\Phi,\bot\alpha\rangle$ wins $\langle\Phi,\bot\alpha\rangle A$, or (b) $\mathscr{G}\langle\Phi,\bot\alpha\rangle$ wins $\neg(\langle\Phi,\bot\alpha\rangle A)$. (a) is impossible because then $\langle\Phi,\bot\alpha\rangle A$ would not be negative. Thus, $\mathscr{G}\langle\Phi,\bot\alpha\rangle$ wins $\neg(\langle\Phi,\bot\alpha\rangle A)$, i.e. $\neg(\langle\bot\alpha\rangle\langle\Phi\rangle A)$, i.e. $\langle\top\alpha\rangle\neg(\langle\Phi\rangle A))$ (see statements (2) and (3) of Section 9). Then machine $\mathscr{G}\langle\Phi\rangle$, that does nothing but makes the move α and plays the rest of the game as $\mathscr{G}\langle\Phi,\bot\alpha\rangle$, obviously wins $\neg(\langle\Phi\rangle A)$.

Case 3: $f\langle\Phi\rangle = g\langle\Phi\rangle = \spadesuit$. Here we consider two subcases:

Subcase 3.1: $\mathbf{Wn}^{\langle\Phi\rangle A}\langle\rangle = \top$. What $\mathscr{F}\langle\Phi\rangle$ does in this case is that it keeps granting permission waiting for the adversary to make a move. If such a move is never made, then the generated run is $\langle\rangle$ which is a $\top$-won run of $\langle\Phi\rangle A$. Suppose now the adversary

makes a move α. In view of Remark 16.4, we may assume that α is $\bot$'s (uni)legal initial move in $\langle\Phi\rangle A$, i.e. $\langle\Phi,\bot\alpha\rangle \in \mathrm{LR}^A$. Since $g\langle\Phi\rangle = \spadesuit$, $\langle\Phi,\bot\alpha\rangle A$ cannot be negative, so that $\mathscr{G}\langle\Phi,\bot\alpha\rangle$ does not win $\neg(\langle\Phi,\bot\alpha\rangle A)$. Therefore, by the induction hypothesis, $\mathscr{F}\langle\Phi,\bot\alpha\rangle$ wins $\langle\Phi,\bot\alpha\rangle A$, i.e. $\langle\bot\alpha\rangle\langle\Phi\rangle A$. From here it is obvious that $\mathscr{F}\langle\Phi\rangle$ wins $\langle\Phi\rangle A$ because, after the move α is made, $\mathscr{F}\langle\Phi\rangle$ continues the rest of the play as $\mathscr{F}\langle\Phi,\bot\alpha\rangle$.

Subcase 3.2: $\mathbf{Wn}^{\langle\Phi\rangle A}\langle\rangle = \bot$. Just as this was the case with $\mathscr{F}\langle\Phi\rangle$, if the adversary never makes an initial move, then the run generated by $\mathscr{G}\langle\Phi\rangle$ is $\langle\rangle$ which is a $\bot$-won run of $\langle\Phi\rangle A$ and hence a $\top$-won run of $\neg(\langle\Phi\rangle A)$. Suppose now the adversary makes a move α. As in the previous subcase, we may assume that α is $\bot$'s legal initial move in $\neg(\langle\Phi\rangle A)$, i.e. α is $\top$'s legal initial move in $\langle\Phi\rangle A$, i.e. $\langle\Phi,\top\alpha\rangle \in \mathbf{LR}^A$. Since $f\langle\Phi\rangle = \spadesuit$, $\langle\Phi,\top\alpha\rangle A$ cannot be positive, so that $\mathscr{F}\langle\Phi,\top\alpha\rangle$ does not win $\langle\Phi,\top\alpha\rangle A$. Therefore, by the induction hypothesis, $\mathscr{G}\langle\Phi,\top\alpha\rangle$ wins $\neg(\langle\Phi,\top\alpha\rangle A)$, i.e. $\langle\bot\alpha\rangle\neg(\langle\Phi\rangle A)$. From here it is obvious that $\mathscr{G}\langle\Phi\rangle$ wins $\neg(\langle\Phi\rangle A)$ because, after the move α is made, $\mathscr{G}\langle\Phi\rangle$ continues the rest of the play as $\mathscr{G}\langle\Phi,\top\alpha\rangle$. □

19. Reducibility

Remember the standard definition of Turing reducibility: simple problem $Q(x)$ is *Turing reducible* to simple problem $R(x)$ iff there is a TM with an oracle for $R(x)$ that decides $Q(x)$. Let us define another, stronger version of Turing reducibility: simple problem $Q(x)$ is *linearly Turing reducible* to simple problem $R(x)$ iff there is a TM with an oracle for $R(x)$ that decides $Q(x)$ using the oracle at most once.

Proposition 19.1. *Let $R(x)$ and $Q(x)$ be simple problems.*

1. *$Q(x)$ is linearly Turing reducible to $R(x)$ iff the following game is winnable*:

$$\sqcap x(R(x)\sqcup\neg R(x)) \rightarrow \sqcap x(Q(x)\sqcup\neg Q(x)).$$

2. *$Q(x)$ is Turing reducible to $R(x)$ iff the following game is winnable*:

$$\sqcap x(R(x)\sqcup\neg R(x)) \Rightarrow \sqcap x(Q(x)\sqcup\neg Q(x)).$$

Proof. *Clause* 1, "*only if*": Suppose $Q(x)$ is linearly Turing reducible to $R(x)$. That is, there is a Turing machine $\mathscr{T}$ with an oracle for $R(x)$ that, on any input $c \in$ **Constants**, will tell us whether $Q(c)$ or not, consulting the oracle at most once. For clarity, we may assume that the oracle will be consulted exactly once. Let then $\mathscr{E}$ be an EPM that works as follows. At the beginning, $\mathscr{E}$ keeps granting permission, waiting for $\bot$ to make the first move. If $\bot$ never makes a move, the game $\sqcap x(R(x)\sqcup\neg R(x)) \rightarrow \sqcap x(Q(x)\sqcup\neg Q(x))$ will be won, because $\langle\rangle$ is a won run of this game. If $\bot$ makes a (legal) move, this move should be $2.c$ for some $c \in$ **Constants**. This is a unilegal move that takes us to the position represented by $\sqcap x(R(x)\sqcup\neg R(x)) \rightarrow (Q(c)\sqcup\neg Q(c))$. Then, $\mathscr{E}$ starts simulating $\mathscr{T}$ on input c. When $\mathscr{E}$ sees that $\mathscr{T}$ consulted the oracle regarding whether $R(b)$ is true (some $b \in$ **Constants**), $\mathscr{E}$ temporarily stops simulation and makes the move $1.b$. This takes us to the position $(R(b)\sqcup\neg R(b)) \rightarrow (Q(c)\sqcup\neg Q(c))$. After this $\mathscr{E}$ loops granting permission and waiting for $\bot$ to make a (its second legal)

move. Obviously, if $\bot$ never makes such a move, it will lose. Otherwise, $\bot$'s move should be either 1.1 or 1.2. If it is 1.1, $\mathscr{E}$ resumes simulation of $\mathscr{T}$, assuming that the latter got the answer "yes" from the oracle. And if $\bot$'s move is 1.2, then $\mathscr{E}$ assumes that $\mathscr{T}$ got the answer "no" from the oracle. $\mathscr{E}$ will continue simulation until $\mathscr{T}$ halts. If $\mathscr{T}$ halts in an accept state, $\mathscr{E}$ makes the move 2.1, otherwise makes the move 2.2. It can be seen that $\mathscr{E}$ then wins the game. Indeed, if $\bot$ has selected the true $\sqcup$-disjunct in the antecedent, then the simulation was perfect because the oracle would indeed also have given the corresponding (true) answer. And if $\bot$ has selected the false disjunct, then $\bot$ loses the game no matter what subsequent (legal) moves $\top$ makes.

Clause 1, "*if*": Suppose $\sqcap\!\!\!\!\sqcap x(R(x)\sqcup\neg R(x))\rightarrow\sqcap\!\!\!\!\sqcap x(Q(x)\sqcup\neg Q(x))$ is winnable. Let $\mathscr{H}$ be an HPM that wins this game. We construct a Turing machine $\mathscr{T}$ with an oracle for $R(x)$ that decides $Q(x)$, using the oracle at most once, as follows. On input c, $\mathscr{T}$ simulates $\mathscr{H}$ for the case when, during the very first cycle, $\bot$ makes the move 2.c and then does not make any moves as long as $\top$ does not make moves. If $\mathscr{H}$ never makes a move, the game would obviously be lost by $\mathscr{H}$, which is impossible, because, by our assumption, $\mathscr{H}$ wins the game. So, $\mathscr{H}$ should make a move either in the consequent or in the antecedent.

A move by $\mathscr{H}$ in the consequent should consist in selecting one of the disjuncts $Q(c)$ or $\neg Q(c)$. Observe that in this case the disjunct that $\mathscr{H}$ selects must be true. Indeed, otherwise $\mathscr{H}$ would lose the game: if it never makes a subsequent move in the antecedent, the game will obviously be lost, and if it makes a move in the antecedent that brings the latter to $R(b)\sqcup\neg R(b)$ for some constant b, $\bot$ could reply by selecting the true disjunct in the antecedent, which would result in a loss for $\top$, contrary to our assumption that $\mathscr{H}$ wins the game. So, $\mathscr{H}$'s selection can be trusted, and we let $\mathscr{T}$ halt in an accept state if $\mathscr{H}$ selected $Q(c)$, and halt in a reject state if $\mathscr{H}$ selected $\neg Q(c)$.

And a move by $\mathscr{H}$ in the antecedent should consist in selecting a value b for x. This brings the antecedent to $R(b)\sqcup\neg R(b)$. Then $\mathscr{T}$ temporarily stops simulation and queries the oracle (the only time when it does so) regarding whether $R(b)$ is true. If the oracle answers "yes", $\mathscr{T}$ continues simulation of $\mathscr{H}$, assuming that $\bot$ made a move in the antecedent by selecting $R(b)$. Otherwise, it assumes that $\bot$ selected $\neg R(b)$. Thus, now the run tape of the simulated $\mathscr{H}$ spells the position represented by $R(b)\rightarrow(Q(c)\sqcup\neg Q(c))$ or $\neg R(b)\rightarrow(Q(c)\sqcup\neg Q(c))$, where the antecedent is true. Obviously, $\mathscr{H}$ will lose the game if it does not make a move in the consequent by selecting the true disjunct. That is, $\mathscr{H}$'s selection, again, can be trusted. So, $\mathscr{T}$ waits till $\mathscr{H}$ makes such a move. If, by this move, $\mathscr{H}$ selects $Q(c)$, then $\mathscr{T}$ halts in an accept state, and if $\mathscr{H}$ selects $\neg Q(c)$, then $\mathscr{T}$ halts in a reject state.

Clause 2: The reasoning here is similar to the one in the proof of clause 1. The difference in the "only if" part is that $\mathscr{T}$ can query the oracle an arbitrary number of times rather than only once. No problem: to each such query corresponds a pair of moves in the antecedent that $\mathscr{E}$ makes: it first replicates $\sqcap\!\!\!\!\sqcap x\big(R(x)\sqcup\neg R(x)\big)$ to save it for future use, and then, in one of the copies, makes a move by specifying x as the constant regarding which $\mathscr{T}$ queried the oracle. As for the "if" part, to each of the many possible non-replicative moves made by $\mathscr{H}$ in the antecedent, corresponds a step querying the oracle by $\mathscr{T}$ for the corresponding constant. $\square$

Both Turing reducibility and linear Turing reducibility are defined only for simple problems. Simple problems are just a small subclass of all problems that are captured by our concept of static games. So, the natural need arises to generalize reducibility to all interactive problems. In view of Proposition 19.1, the operations $\rightarrow$ and $\Rightarrow$ capture the essence of these two sorts of reducibility, and it is natural to use them to define the ultimate generalization of reducibility as follows:

Definition 19.2. Let A and B be static games.
1. *Generalized linear Turing reducibility*: We say that A is *reducible* (or *$\rightarrow$-reducible*, or *strongly reducible*, or *linearly reducible*) to B iff $B \rightarrow A$ is winnable.
2. *Generalized Turing reducibility*: We say that A is *weakly reducible* (or $\Rightarrow$-reducible) to B iff $B \Rightarrow A$ is winnable.
3. *Equivalence*: We say that A and B are *equivalent* iff A and B are mutually $\rightarrow$-reducible.

20. User's strategies formalized

In Section 15 we defined winning as a binary relation between $\top$'s possible strategies, understood as hard-play machines, and games. In that definition $\bot$'s strategies were not visible; rather, they were thought of as arbitrary or non-deterministic sequences of $\bot$'s moves that could be made with an arbitrary frequency. While this was sufficient to fully capture our intuition of winnability, for both technical and philosophical reasons, we may want to have a formal concept of deterministic $\bot$'s strategies. In that case, we would get a ternary relation of winning: "$\top$'s strategy f wins game A against $\bot$'s strategy g". Being able to describe $\bot$'s strategies will be technically useful or necessary when trying to show that a certain game is not winnable. Obviously in such cases we would select the hard-play model (for $\top$) which, even though does not really increase the chances of $\bot$ to defeat $\top$ in a play over a static game, still strongly favors $\bot$ in the sense that it allows $\bot$ to achieve victory with less efforts. For this reason, we are going to attempt to formalize $\bot$'s strategies only for the case when $\top$ is an HPM.

It turns out that when $\top$ plays as an HPM, $\bot$'s strategy can be understood as a fair oracle EPM, so that we get two machines playing against each other over a game A, with the HPM ($\top$) trying to win A, and the EPM ($\bot$) trying to win $\neg A$—the version of A as seen by $\bot$.

Here is how we can visualize the idea. When HPM $\mathscr{H}$ plays against EPM $\mathscr{E}$, most of the time $\mathscr{H}$ remains inactive (sleeping); it is woken up only when $\mathscr{E}$ enters a permission state, on which event $\mathscr{H}$ makes a (one single) transition—that may or may not result in making a move—and goes back to sleep that will continue until $\mathscr{E}$ enters a permission state again, and so on. From $\mathscr{E}$'s prospective, $\mathscr{H}$ acts as a patient adversary who makes one or zero move only when granted permission, just as the easy-play model assumes. And from $\mathscr{H}$'s prospective, who, like a person in a coma, has no sense of time during its sleep and hence can think that the wake-up events that it calls the beginning of a clock cycle happen at a constant rate, $\mathscr{E}$ acts as an adversary

who can make any finite number of moves during a clock cycle, just as the hard-play model assumes.

Of course, when EPM $\mathscr{E}$ plays in the role of $\bot$, the moves it makes in the run that is being generated will get the label $\bot$ (rather than $\top$), and the moves made by its adversary get the label $\top$. In other words, the run $\mathscr{E}$ generates is not the run spelled (in the sense of Section 16) by the corresponding computation branch B of $\mathscr{E}$, but rather the negation of that run, i.e. the run that we say to be cospelled by B:

Definition 20.1. Let $\mathscr{E}$ be an EPM. We say that e-computation branch B of $\mathscr{E}$ *cospells* run Γ iff B spells $\neg\Gamma$. A run cospelled by some e-computation branch of $\mathscr{E}$ is said to be *cogenerated* by $\mathscr{E}$ on e.

Now come the main definitions of this section:

Definition 20.2. Let $\mathscr{H}$ and $\mathscr{E}$ be an HPM and a fair EPM (possibly with oracles), respectively, and e any valuation. The $(\mathscr{E}, e, \mathscr{H})$-*branch* is the following sequence $B_{\mathscr{E}} = C_1, C_2, C_3, \ldots$ of configurations of $\mathscr{E}$, with each element of which is *associated* a configuration of $\mathscr{H}$, defined as follows:

- C_1 is the initial configuration of $\mathscr{E}$, and the configuration of $\mathscr{H}$ associated with C_1 is the initial configuration of $\mathscr{H}$.
- Suppose C_n is a configuration of $B_{\mathscr{E}}$, and the configuration of $\mathscr{H}$ associated with C_n is D. Let Φ_n be the position component of C_n. Then:
  - If the state component of C_n is not a permission state, then C_{n+1} is the unique e-successor (in $\mathscr{E}$) of C_n, and the configuration of $\mathscr{H}$ associated with C_{n+1} is the same D.
  - Suppose now the state component of C_n is a permission state. Let k be the smallest positive integer such that for any i with $k \leq i < n$, the state component of C_i is not a permission state; and let Φ_k be the position component of C_k. Obviously we must have $\Phi_n = \langle \Phi_k, \Theta \rangle$, where $\Theta = \langle \top\alpha_1 \ldots \top\alpha_m \rangle$ for some moves $\alpha_1, \ldots, \alpha_m$ (possibly $m = 0$). Fix this Θ.

    Then we define C_{n+1} to be the unique e-successor (in $\mathscr{E}$) of C_n whose position component Φ_{n+1} is defined by:
    * If $\mathscr{H}$ does not make a move in configuration D, then $\Phi_{n+1} = \Phi_n$;
    * If $\mathscr{H}$ makes a move β in configuration D, then $\Phi_{n+1} = \langle \Phi_n, \bot\beta \rangle$.

    And the configuration of $\mathscr{H}$ associated with C_{n+1} is the $(e, \neg\Theta)$-successor of D in $\mathscr{H}$.

Definition 20.3. Assume e, $\mathscr{E}$, $\mathscr{H}$ and $B_{\mathscr{E}} = C_1, C_2, \ldots$ are as in Definition 20.2. Let $C_{n_1}, C_{n_2}, \ldots$ be all the configurations of $B_{\mathscr{E}}$ (in their normal order) whose state components are permission states, and let $B_{\mathscr{H}} = D_1, D_2, \ldots$ be the sequence of the configurations of $\mathscr{H}$ associated with $C_{n_1}, C_{n_2}, \ldots$ (in the sense of Definition 20.2), respectively. Then $B_{\mathscr{H}}$ is said to be the $(\mathscr{H}, e, \mathscr{E})$-*branch*.

Lemma 20.4. *Suppose $\mathscr{H}$ and $\mathscr{E}$ are an HPM and a fair EPM (possibly with oracles), respectively, and e is any valuation. Then*:

(a) *The $(\mathscr{E}, e, \mathscr{H})$-branch is an e-computation branch of $\mathscr{E}$, and the $(\mathscr{H}, e, \mathscr{E})$-branch is an e-computation branch of $\mathscr{H}$.*

(b) *The run spelled by the $(\mathscr{H}, e, \mathscr{E})$-branch is the run cospelled by the $(\mathscr{E}, e, \mathscr{H})$-branch.*

(c) *As long as $\mathscr{E}$ and $\mathscr{H}$ are without oracles and e is effective, there is an effective function that, for any positive integer n, returns the n*th *configuration of the $(\mathscr{E}, e, \mathscr{H})$-branch. Similarly for the $(\mathscr{H}, e, \mathscr{E})$-branch.*

Proof. Clauses (a) and (b) can be verified by a routine examination of the relevant definitions. Clause (c) is also obvious: assuming the conditions of that clause, the work of either machine can be effectively traced up to any given computation step n. □

Definition 20.5. Suppose $\mathscr{H}$ and $\mathscr{E}$ are an HPM and a fair EPM (possibly with oracles), respectively, and e is any valuation. Then the run spelled by the $(\mathscr{H}, e, \mathscr{E})$-branch is called *the $\mathscr{H}$ vs $\mathscr{E}$ run on e*.

Proposition 20.6. *No static game can be both positive and negative.*

Proof. Suppose that A is a positive and negative static game. Let $\mathscr{H}$ be an oracle HPM that wins A and $\mathscr{E}$ a fair oracle EPM that wins $\neg A$. Fix an arbitrary valuation e, and consider the $\mathscr{H}$ vs $\mathscr{E}$ run Γ on e. In view of Lemma 20.4, Γ is a run generated by $\mathscr{H}$ on e, and $\neg\Gamma$ is a run generated by $\mathscr{E}$ on e. Since $\mathscr{H}$ wins A, we should have $\mathbf{Wn}_e^A\langle\Gamma\rangle = \top$. And since $\mathscr{E}$ wins $\neg A$, we should have $\mathbf{Wn}_e^{\neg A}\langle\neg\Gamma\rangle = \top$ and hence $\mathbf{Wn}_e^A\langle\Gamma\rangle = \bot$. This is a contradiction. □

Definition 20.7. Let A be a (not necessarily static) game, and $\mathscr{H}$ and $\mathscr{E}$ an HPM and a fair EPM, respectively, possibly with oracles. We say that *$\mathscr{H}$ wins A against $\mathscr{E}$ on valuation e* iff the $\mathscr{H}$ vs $\mathscr{E}$ run on valuation e is a $\top$-won run of A w.r.t. e.

Proposition 20.8. *An HPM (possibly with an oracle) $\mathscr{H}$ wins a (not necessarily static) game A on valuation e iff $\mathscr{H}$ wins A against every fair oracle EPM on e.*

Proof. Consider an arbitrary HPM (possibly with an oracle) $\mathscr{H}$, an arbitrary game A and an arbitrary valuation e. The *if* part is straightforward: Suppose $\mathscr{H}$ does not win A against a certain fair oracle EPM $\mathscr{E}$ on e. This means that the $\mathscr{H}$ vs $\mathscr{E}$ run on e—call it Γ—is a $\top$-lost run of $e[A]$. According to Lemma 20.4(a), Γ is a run generated by $\mathscr{H}$ on e. Hence, $\mathscr{H}$ does not win A on e.

Only if: Suppose $\mathscr{H}$ does not win A on e. This means that there is an e-computation branch $B_{\mathscr{H}} = C_1, C_2, \ldots$ of $\mathscr{H}$ such that the run spelled by this branch—call it Γ—is a $\top$-lost run of $e[A]$. In view of Definitions 15.4 and 15.3, there is an infinite sequence $\Theta_1, \Theta_2, \ldots$ of finite sequences of $\bot$-labeled moves such that, for every i, C_{i+1} is the (e, Θ_i)-successor of C_i in $\mathscr{H}$. Let $f(i)$ be the function that sends each i to Θ_i.

Now we define an EPM $\mathscr{E}$ with an oracle for $f(x)$ as follows. The work of $\mathscr{E}$ is divided into stages: stage 1, stage 2, ... On each stage i, $\mathscr{E}$ does the following: using the oracle for $f(x)$, it finds Θ_i. Let $\Theta_i = \langle \bot\alpha_1, \ldots, \bot\alpha_k \rangle$. Then $\mathscr{E}$ makes the k consecutive moves: $\alpha_1, \ldots, \alpha_k$, upon the completion of which grants permission, and goes to stage $i+1$.

A little thought can now convince us that the $\mathscr{H}$ vs $\mathscr{E}$ run on valuation e will be exactly the run Γ from which we started and, since the latter is a $\top$-lost run of $e[A]$, we conclude that $\mathscr{H}$ does not win A against $\mathscr{E}$ on e. □

Part III—The logic of computability

This part is devoted to the question of what principles of winnability are valid for static games. Since from now on we are going to be exclusively focused on static games, let us agree—without further reminding about this convention, that the letters A, B, C, D, that in the previous Parts ranged over all games, will now range over static games only. We also introduce the notations

$$\mathscr{M} \models_e A, \quad \mathscr{M} \models A \text{ and } \quad \models A$$

to mean that (hard- or easy-play) machine $\mathscr{M}$ wins game A on valuation e, that machine $\mathscr{M}$ wins game A and that game A is winnable, respectively.

In view of Thesis 18.2, we will be using the terms "*computable*" and "*computability*" as synonyms of "winnable" and "winnability". Similarly, in view of Thesis 4.4, we will be using the word "computational problem", or, simply, "*problem*" as a synonym of "static game".

We are going to prove a few winnability results by describing an EPM that wins a given game. These descriptions and the corresponding correctness proofs may be more relaxed and high level than most of the similar descriptions and proofs in Part II as we have already acquired sufficient experience in dealing with play machines. In concordance with Remark 16.4, this sort of a description typically will not say what the machine does if $\bot$ makes an illegal move, assuming that such an event never happens. We will also often omit the typically straightforward but annoying details showing that as long as the environment does not make illegal moves, neither does the machine, thus explicitly considering in our correctness proofs only the cases when the generated run is legal.

21. Basic closure properties of computability

Proposition 21.1. ($\sqcap$-closure) *If* $\models A(x)$, *then* $\models \sqcap x A(x)$.

Moreover, there is an effective procedure that, for any EPM $\mathscr{E}$, *returns an EPM* $\mathscr{E}'$ *such that, for every* $A(x)$ *and* e, *if* $\mathscr{E} \models_e A(x)$, *then* $\mathscr{E}' \models_e \sqcap x A(x)$.

Proof. This is how $\mathscr{E}'$ works on valuation e: At the beginning it keeps granting permission, waiting till $\bot$ makes the move c for some constant c. Then it starts working

exactly as $\mathscr{E}$ would work, pretending that the run tape does not contain the initial lab-move $\bot c$ and the valuation tape spells the valuation that evaluates x as c and agrees with e on all other variables. It is easy to see that, as long as $\mathscr{E}$ wins $A(x)$ on e, $\mathscr{E}'$ wins $\sqcap xA(x)$ on e. $\square$

Note that a similar closure principle fails with $\forall$ instead of $\sqcap$. For example, we obviously have $\models Even(x) \sqcup Odd(x)$ but not $\models \forall x(Even(x) \sqcup Odd(x))$.

Proposition 21.2 (!-closure). *If* $\models A$, *then* $\models !A$.

Moreover, there is an effective procedure that, for any EPM $\mathscr{E}$ *returns an EPM* $\mathscr{E}'$ *such that, for every* A *and* e, *if* $\mathscr{E} \models_e A$, *then* $\mathscr{E}' \models_e !A$.

Proof. The idea here is simple: if we (machine $\mathscr{E}$) can win A once, we can use the same strategy to win A an arbitrary number of times, and this is exactly what winning $!A$ is all about. The machine $\mathscr{E}'$ that materializes this idea works as follows. At any time, where Φ is the current (prelegal) position of the play over $!A$, for each leaf w of $Tree^A_\Phi$, $\mathscr{E}'$ maintains a record (w, C), where C is a certain configuration of $\mathscr{E}$. The initial position of the play over $!A$ is $\langle\rangle$, and $\mathscr{E}'$ creates the record (ε, C_0) for the only leaf ε of $Tree^A_{\langle\rangle}$, where C_0 is the initial configuration of $\mathscr{E}$. After this initialization step, with e spelled on its valuation tape, $\mathscr{E}'$ repeats the following procedure infinitely many times:

Procedure MAIN. Let Φ be the current position of the play, $w_1, \ldots, w_n$ all leaves of $Tree^A_\Phi$, and $(w_1, C_1), \ldots, (w_n, C_n)$ the corresponding records. Act as follows.

Step 1. For $i = 1$ **to** $i = n$, **while** the state component of C_i is not a permission state, **do** the following:

(a) **If** $\mathscr{E}$ makes a move α in configuration C_i, **then** make the move $w_i.\alpha$.

(b) Update (w_i, C_i) to (w_i, C'), where C' is the (unique) e-successor of C_i in $\mathscr{E}$.

We may assume that $\mathscr{E}$ is fair, so that this step ends sooner or later and, by that time, obviously the state component of each C_i $(1 \leqslant i \leqslant n)$ will be a permission state.

Step 2. Grant permission. Then:

(a) **If** $\bot$ responds by a replicative move w: (w a leaf of $Tree^A_\Phi$), **then** destroy the record (w, C) and create the two new records $(w0, C)$ and $(w1, C)$; after that update each of the existing records (u, D) to (u, D'), where D' is the e-successor of D in $\mathscr{E}$ whose position component is the same as that of D.

(b) **If** $\bot$ responds by a non-replicative move $w.\alpha$ (w a node of $Tree^A_\Phi$), **then** do the following:

(*) For each record (u, D) with $w \preccurlyeq u$, update (u, D) to (u, D'), where D' is the e-successor of D in $\mathscr{E}$ whose position component is the result of appending $\bot\alpha$ to the position component of D;

(**) For every other record (u, D), update (u, D) to (u, D'), where D' is the e-successor of D in $\mathscr{E}$ whose position component is the same as that of D.

(c) **If** $\bot$ does not respond with any (prelegal) move, **then** update each record (u, D) to (u, D'), where D' is the e-successor of D in $\mathscr{E}$ whose position component is the same as that of D.

Let B be an arbitrary e-computation branch of $\mathcal{E}'$, and let Γ be the run spelled by B. Assume $\mathcal{E} \models_e A$. In order to show that $\mathbf{Wn}_e^{!A}\langle\Gamma\rangle = \top$, we need to verify [10] that, for an arbitrary infinite bit string u (let us fix this u), $\mathbf{Wn}_e^{A}\langle\Gamma^{\preccurlyeq u}\rangle = \top$. The work of the machine along branch B after the initialization is made consists in iterating MAIN infinitely many times. Consider any such iteration. At the beginning of the iteration, when the current position of the play is Φ and $w_1, \ldots, w_n$ are all the leaves of $Tree_\Phi^A$, the machine has a record (w_i, C_i) per each $1 \leqslant i \leqslant n$. Obviously, there will be exactly one leaf among these w_i such that $w_i \preccurlyeq u$. For notational convenience, let us rename the record (w_i, C_i) into (s, G). During Step 1 of MAIN, the record (s, G) will (may) undergo updates only on the ith iteration of the **for** loop, by the **while** loop within that iteration. These updates will only modify the G component of the record, while the s component will remain unchanged. Let $(s, G_1), \ldots, (s, G_k)$, with $G_1 = G$, be the values that the record (s, G) will go through during this **while** loop, (s, G_k) thus being the value by the beginning of Step 2. Let us call the sequence $\vec{D} = G_1, \ldots, G_k$ the $\preccurlyeq u$-*sequence* of the given iteration of MAIN. Now, let B^u be the sequence $\vec{D}_1, \vec{D}_2, \ldots$ (i.e. the concatenation of $\vec{D}_1$, $\vec{D}_2$, ...), where each $\vec{D}_j$ is the $\preccurlyeq u$-sequence of the jth iteration of MAIN. An analysis of the description of $\mathcal{E}'$ can convince us that B^u is an e-computation branch of $\mathcal{E}$, and the run it spells is nothing but $\Gamma^{\preccurlyeq u}$. Therefore, since $\mathcal{E} \models_e A$, $\mathbf{Wn}_e^{A}\langle\Gamma^{\preccurlyeq u}\rangle = \top$. □

Proposition 21.3 (Modus Ponens). *If* $\models A$ *and* $\models A \rightarrow A'$, *then* $\models A'$.

Moreover, there is an effective procedure that, for any HPM $\mathcal{H}$ *and EPM* $\mathcal{E}$, *returns an EPM* $\mathcal{E}'$ *such that, for every* A, A' *and* e, *if* $\mathcal{H} \models_e A$ *and* $\mathcal{E} \models_e A \rightarrow A'$, *then* $\mathcal{E}' \models_e A'$.

Proof. The idea is to mimic the behavior of $\mathcal{E}$ in the consequent of $A \rightarrow A'$ while letting $\mathcal{H}$ play against $\mathcal{E}$ in the antecedent. As in the proof of Proposition 21.2, we may assume that $\mathcal{E}$ is fair. This is how the EPM $\mathcal{E}'$ works. It maintains three records: $C_\mathcal{H}$, holding a configuration of $\mathcal{H}$, $C_\mathcal{E}$, holding a configuration of $\mathcal{E}$, and Θ, holding a sequence of $\bot$-labeled moves. The initialization step consists in creating these three records with Θ set to $\langle\rangle$ and $C_\mathcal{H}$ and $C_\mathcal{E}$ set to the initial configurations of the corresponding two machines. After that $\mathcal{E}'$, with e spelled on its valuation tape, repeats the following procedure infinitely many times:

Procedure MAIN.

Step 1. While the state component of $C_\mathcal{E}$ is not a permission state, **do** the following:
(a) **If** $\mathcal{E}$ makes a move $2.\alpha$ in configuration $C_\mathcal{E}$, **then** make the move α.
(b) **If** $\mathcal{E}$ makes a move $1.\alpha$ in configuration $C_\mathcal{E}$, **then** update Θ to $\langle\Theta, \bot\alpha\rangle$.
(c) Update $C_\mathcal{E}$ to its (unique) e-successor in $\mathcal{E}$.

Step 2. If $\mathcal{H}$ makes a move α in configuration $C_\mathcal{H}$, **then** update $C_\mathcal{E}$ to the e-successor of $C_\mathcal{E}$ in $\mathcal{E}$ whose position component is the result of appending $\bot 1.\alpha$ to the position component of $C_\mathcal{E}$; **else** update $C_\mathcal{E}$ to the e-successor of $C_\mathcal{E}$ in $\mathcal{E}$ whose position component is the same as that of $C_\mathcal{E}$.

[10] Remember what we said on page 70 about considering only legal runs in this sort of proofs.

Step 3. Update $C_{\mathscr{H}}$ to its (Θ, e)-successor in $\mathscr{H}$, and (re)set Θ to $\langle\rangle$.

Step 4. Once again do exactly the same as in Step 1.

Step 5. Grant permission. **If** $\bot$ responds by a move α, **then** update $C_{\mathscr{E}}$ to its e-successor in $\mathscr{E}$ whose position component is the result of appending $\bot 2.\alpha$ to the position component of $C_{\mathscr{E}}$; **else** update $C_{\mathscr{E}}$ to its e-successor in $\mathscr{E}$ whose position component is the same as that of $C_{\mathscr{E}}$.

Let B be an arbitrary e-computation branch of $\mathscr{E}'$, and let Γ be the run spelled by B. Consider the post-initialization work of $\mathscr{E}'$ along branch B, which consists in iterating MAIN infinitely many times. For each $i \geqslant 1$, let C_i be the value of $C_{\mathscr{H}}$ at the beginning of the ith iteration of MAIN, $\vec{D}_i$ the values that the record $C_{\mathscr{E}}$ goes through during Step 1 of the ith iteration of MAIN, and $\vec{G}_i$ the values that the record $C_{\mathscr{E}}$ goes through during Step 4 of the ith iteration of MAIN. One can verify that $C_1, C_2, \ldots$ is an e-computation branch of $\mathscr{H}$—call the run spelled by this branch $\Delta_{\mathscr{H}}$, and $\vec{D}_1, \vec{G}_1, \vec{D}_2, \vec{G}_2 \ldots$ is an e-computation branch of $\mathscr{E}$—call the run spelled by it $\Delta_{\mathscr{E}}$, such that $\Delta^{1.}_{\mathscr{E}} = \neg\Delta_{\mathscr{H}}$ and $\Delta^{2.}_{\mathscr{E}} = \Gamma$. From this it is not hard to see that if Γ is a lost run of A' w.r.t. e, then either $\Delta_{\mathscr{H}}$ is a lost run of A w.r.t. e and hence $\mathscr{H} \not\models_e A$, or $\Delta_{\mathscr{E}}$ is a lost run of $A \to A'$ w.r.t. e and hence $\mathscr{E} \not\models_e A \to A'$. □

22. A glance at some valid principles of computability

In this section we will take a quick look at some simple principles of computability.

Proposition 22.1. (1) *For all* A, $\models A \vee \neg A$. *Moreover, there is an EPM* $\mathscr{E}$*,—let us call it the* mimicking EPM*,—such that, for every* A, $\mathscr{E} \models A \vee \neg A$.

(2) *For some* (*in fact finitary and elementary, or constant and finite-depth*) A, $\not\models A \sqcup \neg A$.

Proof. *Clause 1:* The mimicking EPM $\mathscr{E}$ just implements the idea of mimicking strategy informally described in Section 1. Here is an interactive algorithm for it:

1. Keep granting permission until $\bot$ makes a move α in one of the two disjuncts (i.e. the move $1.\alpha$ or $2.\alpha$).
2. Make the same move α in the other disjunct, and go back to step 1.

It is obvious that for any run Γ generated by this machine, we will have $\Gamma^{2.} = \neg\Gamma^{1.}$, which makes it a won run of any game $A \vee \neg A$ w.r.t. any e, because if a run Δ (in particular, $\Gamma^{1.}$) is a lost run of $e[A]$, then, by the definition of $\neg$, the run $\neg\Delta$ (in particular, $\Gamma^{2.}$) is a won run of $e[\neg A]$.

Clause 2: A simple way to disvalidate $A \sqcup \neg A$ is to select an A that depends on infinitely many variables, e.g., A can be the infinitary predicate that is true at e iff there are infinitely many variables x with $e(x) = 0$. Then, at no particular time would the machine be able to select the true disjunct. But counterexamples appealing to infinitary games are not very informative or interesting: the reason for failure to win $A \sqcup \neg A$ in this case is that it is simply impossible to ever finish reading the

infinite relevant part of the input (valuation tape), rather than that it is impossible to compute. So, we should always try to find a finitary counterexample for a given principle.

No problem. Let A be an arbitrary undecidable unary predicate $P(x)$ that depends on x. By Proposition 18.3 we have $\not\models \bigsqcap x(P(x) \sqcup \neg P(x))$. This, by Proposition 21.1, implies that $\not\models P(x) \sqcup \neg P(x)$, i.e. $\not\models A \sqcup \neg A$.

We can disvalidate $A \sqcup \neg A$ with a constant—still finite-depth even though no longer elementary—problem A as well. The proof of incomputability of $A \sqcup \neg A$ in this case is considerably harder. Here is a very brief explanation of the proof idea. One can show that the predicate of incomputability of (the problems expressed by) formulas of classical arithmetic, with the classical operators (including quantifiers) read as the corresponding choice operators, can be expressed in the language of arithmetic—in particular, can be expressed by a Π_3-formula. Using Gödel's self-reference technique, we can select A to be a Π_3-sentence saying "I am incomputable". Such an A cannot be computable because in that case what it says would be false, and one can show that a false formula can never be computable. Since A is incomputable, it is true (because this is just what it asserts), hence $\neg A$ is false, and hence $\neg A$ is not computable, either. Thus, neither disjunct of $A \sqcup \neg A$ is computable. This obviously implies the incomputability of the disjunction itself. □

All of the further similar computability/incomputability statements in this section go without proofs. Most of the positive statements can be verified rather easily, and typically, as in Proposition 22.1(1), we have something stronger than just computability: there is an EPM that wins the given compound game for every particular value of the subgames, the property that will be called "uniform validity" in Section 26. Proving incomputability statements are typically harder and may require serious thought. Since no proofs are given, the rest of the statements in this section are labeled "claims" rather than "propositions".

We will be using the expression $\models A \leftrightarrow B$ as an abbreviation of the two statements: $\models A \rightarrow B$ and $\models B \rightarrow A$. That is, $\models A \leftrightarrow B$ means that A and B are strongly equivalent. All of the following, —positive or negative, —claims satisfy the principle of duality: they remain true if the arrow $\rightarrow$ is reversed and $\bot, \sqcap, \wedge, \bigsqcap, \forall, !$ are interchanged with $\top, \sqcup, \vee, \bigsqcup, \exists, ?$, respectively.

Claim 22.2. *For all A, B and C, we have*:

1. *Domination*: $\models A \wedge \bot \leftrightarrow \bot$; $\models A \sqcap \bot \leftrightarrow \bot$.
2. *Identity*: $\models A \wedge \top \leftrightarrow A$; $\models A \sqcap \top \leftrightarrow A$.
3. *Commutativity*: $\models A \sqcap B \leftrightarrow B \sqcap A$; $\models A \wedge B \leftrightarrow B \wedge A$.
4. *Associativity*: $\models (A \sqcap B) \sqcap C \leftrightarrow A \sqcap (B \sqcap C)$; $\models (A \wedge B) \wedge C \leftrightarrow A \wedge (B \wedge C)$.
5. *Idempotence*: $\models A \sqcap A \leftrightarrow A$.
6. *Distributivity*: $\models A \vee (B \sqcap C) \leftrightarrow (A \vee B) \sqcap (A \vee C)$.

Thus, both the choice as well as parallel versions of disjunction and conjunction are commutative and associative; the domination and identity laws also hold for both sorts of conjunction and disjunction. As for distributivity, one can show that it only holds

in the form given in clause 6 of the above claim and its dual. And, while the choice operations are idempotent, the parallel operations are not:

Claim 22.3. *For some* (*in fact constant and finite-depth*) A, *we have* $\not\models A \to A \wedge A$.

Claims 22.2 and 22.3 might have suggested that Girard's linear logic has good chances to be sound and complete with respect to our semantics. Well, we claim that it is really sound, but pretty far from being complete, as follows from clause 1 of the following statement:

Claim 22.4. *For all* $A_1, \ldots, A_n$ $(n \geqslant 2)$ *and* $1 \leqslant i \leqslant n$, *we have*:
1. $\models A_1 \wedge \cdots \wedge A_n \to A_i$,
2. $\models A_1 \sqcap \cdots \sqcap A_n \to A_i$.

Well, both of the above principles are derivable in Affine logic, which is the extension of linear logic by adding the weakening rule to it. Maybe then Affine logic is sound and complete with respect to our semantics? The answer, again, is "yes" to soundness and "no" to completeness. Here are two valid principles of computability that are underivable in Affine logic:

Claim 22.5. *For all* A, B, C, D, *we have*:
1. $\models (A \wedge B) \vee (C \wedge D) \to (A \vee C) \wedge (B \vee D)$,
2. $\models (A \wedge (C \sqcap D)) \sqcap (B \wedge (C \sqcap D)) \sqcap ((A \sqcap B) \wedge C) \sqcap ((A \sqcap B) \wedge D) \to (A \sqcap B) \wedge (C \sqcap D)$.

The game-semantical validity of the first of the above two principles was first observed by Blass [5].

In the following statements the arrows cannot be reversed:

Claim 22.6. *For all* A, B *and* C, *the following partial distributivities hold*:
1. $\models A \wedge (B \sqcap C) \to (A \wedge B) \sqcap (A \wedge C)$,
2. $\models A \sqcap (B \vee C) \to (A \sqcap B) \vee (A \sqcap C)$,
3. $\models (A \sqcap B) \wedge (A \sqcap C) \to A \sqcap (B \wedge C)$.

Valid principles with branching operations include the following.

Claim 22.7. *For all* A, B *and* x:
1. $\models !A \to A$,
2. $\models !A \leftrightarrow !!A$,
3. $\models !A \leftrightarrow !A \wedge !A$,
4. $\models !(A \sqcap B) \leftrightarrow !(A \wedge B)$,
5. $\models !(A \wedge B) \leftrightarrow !A \wedge !B$,
6. $\models !(A \sqcup B) \leftrightarrow !A \sqcup !B$,

7. $\models !\bigsqcup xA \leftrightarrow \bigsqcup x!A$,
8. $\models !\forall xA \leftrightarrow \forall x!A$, —*in fact*, $!\forall xA = \forall x!A$ (*A unistructural in x*),
9. $\models \exists x!A \rightarrow !\exists xA$ (*A unistructural in x*).

There are some principles that only hold for elementary problems. Here is an easy-to-verify example:

Claim 22.8. *For all elementary A*, $\models A \leftrightarrow !A$.

23. Universal problems

When talking about computational problems, we often only consider elements of some fixed set of problems. In typical or at least most interesting cases such a set will be countable, and usually come with some standard enumeration of its elements, such as the lexicographic order of expressions representing the problems of the set. We call such enumerated sets "problem universes":

Definition 23.1. A *problem universe* is a sequence of problems.

Such a sequence can be finite or infinite. However, for convenience and without loss of generality, we will assume that all problem universes that we consider are infinite, for a finite problem universe can always be rewritten as an "equivalent" infinite universe by replicating its last problem infinitely many times. Also, even though, strictly speaking, a problem universe should be a sequence rather than a set, we will often describe a universe just by explaining what problems are in it, i.e. present it as a set of problems rather than a sequence of problems. This can be done when how those elements are arranged in the sequence is irrelevant, or when there is some obvious, standard arrangement such as the one based on the lexicographic order of the (possibly multiple) names of the problems of the universe. In any case, when we write $A \in U$ for a given problem universe U, we mean that problem A appears (possibly with repetitions) in U.

For problems A and B, let us call any EPM that wins $A \rightarrow B$ an *EPM-reduction* of B to A.

Definition 23.2. Let $U = \langle A_1, A_2, \ldots \rangle$ be a problem universe. A *universal problem* of U is a problem \$ such that:

1. Each $A_i \in U$ is reducible to \$ and,
2. furthermore, there is an effective procedure that, for each $i \geqslant 1$, returns an EPM-reduction of A_i to \$.

The intuition behind the concept of universal problem \$ is that this is a problem of universal strength: having it as a computational resource means being able to solve any of the problems of the universe as every problem of the universe is algorithmically

reducible to \$. However, just existence of those algorithmic reductions would have little value if there was no way to find them. The 'furthermore' clause of the above definition guarantees that reduction strategies not only exist, but can be effectively found.

It is evident that $\bot$ is a universal problem of every problem universe U. Furthermore, if U contains a problem whose negation is computable, then every universal problem of U will be equivalent to $\bot$. However, the concept of universal problem is more general than $\bot$. E.g., some natural problem universes may entirely consist of positive problems, in which case a much weaker—in particular, positive—problem can serve as a universal problem. The acceptance problem for the universal Turing machine, which of course is positive, would obviously be a universal problem for the universe that consists of acceptance problems for all particular Turing machines (with those problems arranged, say, in the lexicographic order of their standard descriptions, or standard descriptions of the corresponding Turing machines).

A universal problem does not necessarily have to be 0-ary. Here is a natural example. Let x range over some set T of axiomatic theories (all in the same language) based on first-order classical logic, with not necessarily recursive or consistent sets of axioms; let y range over formulas of the language of those theories, and z range over finite sequences of such formulas; let $Proof(x,y,z)$ be the predicate "z is a proof of y in x". Then $\$ = \sqcup z\ Proof(x,\bot,z)$, i.e. the problem of finding a proof of the absurd in theory x, will be a universal problem for the universe $U = \{\sqcup z\ Proof(x,F,z) \mid F \text{ is a formula}\}$, which consists of the problems of finding proofs in a given theory x for various particular formulas. Notice that if all theories from T are inconsistent, then the entire universe U (that includes \$) exclusively consists of positive problems; at the same time, unless T is finite or all theories from T are recursively enumerable, \$ and some other problems from U could be incomputable.

The list of strict mathematical examples of problem universe/universal problem pairs illustrating the meaning and usefulness of the idea of universal problem could be continued of course. But this idea is so natural that it can as well be explained in non-mathematical terms to laymen or even little children: understanding problems in a very general—not-only-computational—sense (see Section 26), the problem of catching the gold fish is a universal problem of the universe $\{Making\ W\ come\ true \mid W\ is\ any\ wish\}$.

In view of the resource intuition associated with our semantics (again, see Section 26 for a detailed discussion), our selection of symbol \$ for universal problems is no accident: money—an adult's gold fish—is a good example of a universal resource, a resource that can be converted into any other resource in the world where everything can be bought.

According to condition 2 of Definition 23.2, there is an effective procedure that finds a reduction of any given element of the problem universe to the universal problem. Generally, however, this procedure could be different for different problem universe/universal problem pairs, which would eventually make it impossible to treat \$ as a logical or semi-logical symbol as we are going to do in the next section. Fortunately, there is a way to standardize the reduction procedure so that it will be common for universal problems of all universes, that is, an element A of a problem universe can

be reduced to a universal problem \$ of that universe without knowing what particular problems A and \$ really are. This effect is achieved by restricting universal problems to what we call "standard universal problems" (Definition 23.3). Such a restriction does not yield loss of generality because every universal problem turns out to be equivalent to and hence can be thought of as some standard universal problem (Proposition 23.4). Putting it another way, standard universal problems can be considered standard representatives of equivalence classes of universal problems and in most contexts can be identified with them.

Definition 23.3. Let $U = \langle A_1, A_2, A_3, \ldots\rangle$ be a problem universe.

1. For a problem A_0, the *A_0-based standard universal problem of U* is the problem \$ defined as follows:
   - $\mathbf{Lr}^{\$}_e = \{\langle\rangle\} \cup \{\langle \bot i, \Gamma\rangle \mid i \in \{0,1,2,\ldots\},\ \Gamma \in \mathbf{Lr}^{A_i}_e\}$.
   - $\mathbf{Wn}^{\$}_e\langle\rangle = \top$; $\mathbf{Wn}^{\$}_e\langle \bot i, \Gamma\rangle = \mathbf{Wn}^{A_i}_e\langle\Gamma\rangle$.

2. A *standard universal problem* \$ *of U* is the A_0-based standard universal problem of U for some problem A_0; the latter is said to be the *base* of \$.

Notice that the A_0-based standard universal problem \$ of $U = \langle A_1, A_2, \ldots\rangle$ is nothing but what can be called the *infinite additive conjunction* $A_0 \sqcap A_1 \sqcap A_2 \sqcap \ldots$. Obviously, we could show that the infinite version of additive conjunction preserves the static property of games in the same way as we did for finite additive conjunctions in Section 10, so \$ indeed is a problem (static game). It is also easy to see that \$ is indeed a universal problem of U. The strategy that wins $\$ \rightarrow A_i$ for the ith element of U is to select the conjunct A_i of the infinite conjunction \$ thus bringing the game down to $A_i \rightarrow A_i$, and then continue the rest of the play as the mimicking EPM described in the proof of Proposition 22.1(1) Notice that, as we claimed a while ago, this strategy is indeed "uniform" in that it does not require any specific knowledge of \$ or A_i.

Proposition 23.4. *Suppose U is a problem universe and A_0 a universal problem of U. Then A_0 is equivalent to the A_0-based standard universal problem of U.*

Proof. Let $U = \langle A_1, A_2, \ldots\rangle$ be a problem universe, A_0 a universal problem of U, and \$ the A_0-based standard universal problem of U. We need to show that $\models \$ \rightarrow A_0$ and $\models A_0 \rightarrow \$$. The former is obvious: $\$ \rightarrow A_0$ will be won by an EPM that makes the move 1.0 and continues playing the resulting game $A_0 \rightarrow A_0$ using the mimicking strategy.

Let now f be an effective function that, for each $i \geqslant 1$, returns (the description of) an EPM that wins $A_0 \rightarrow A_i$. Such a function exists according to clause 2 of Definition 23.2. We extend the domain of f to 0 by letting $f(0)$ be the mimicking EPM that wins $A_0 \rightarrow A_0$. Here is a very relaxed description of an EPM $\mathscr{E}$ that wins $A_0 \rightarrow \$$ on an arbitrary valuation e. As always (Remark 16.4), this description assumes that the environment never makes illegal moves. The work of $\mathscr{E}$ consists of three stages.

During stage 1, $\mathscr{E}$ keeps granting permission until $\bot$ selects a conjunct A_i in the consequent, remembering the moves $\alpha_1,\ldots,\alpha_n$ that $\bot$ may meanwhile make in the antecedent. We may assume that such a selection of A_i will be made sooner or later, for otherwise $\mathscr{E}$ clearly wins.

During stage 2, using f, $\mathscr{E}$ finds an EPM $\mathscr{E}_i$ that wins $A_0 \rightarrow A_i$. In view of Theorem 17.2, we may assume that $\mathscr{E}_i$ is fair. Then $\mathscr{E}$ traces/simulates, on its work tape, the work of $\mathscr{E}_i$ up to the configuration C when $\mathscr{E}_i$ grants its $(n+1)$th permission, remembering the moves $\beta_1,\ldots,\beta_m$ that $\mathscr{E}_i$ makes meanwhile. This simulation assumes that to each jth permission $(1 \leqslant j \leqslant n)$ the environment responded by the move $1.\alpha_j$, and that the valuation tape of $\mathscr{E}_i$ spells the same e as the valuation tape of $\mathscr{E}$.

At the beginning of stage 3, $\mathscr{E}$ makes the above moves $\beta_1,\ldots,\beta_m$. After that it plays the rest of the game as $\mathscr{E}_i$ (with e on its valuation tape) would play starting from configuration C. That is, $\mathscr{E}$ resumes simulation of $\mathscr{E}_i$ only, unlike stage 2, now every time it sees that $\mathscr{E}_i$ made a move, $\mathscr{E}$ also makes the same move, and every time it sees that $\mathscr{E}_i$ granted permission, $\mathscr{E}$ also grants permission and then copies $\bot$'s response to the simulated run tape contents of $\mathscr{E}_i$.

The run generated by $\mathscr{E}$ in the above scenario will thus look like $\langle \bot 1.\alpha_1,\ldots,\bot 1.\alpha_n, \bot 2.i, \top\beta_1,\ldots,\top\beta_m, \Delta\rangle$. One can see that this is a won run of $e[A_0 \rightarrow \$]$ if $\Gamma = \langle \bot 1.\alpha_1, \ldots, \bot 1.\alpha_n, \top\beta_1,\ldots,\top\beta_m,\Delta\rangle$ is a won run of $e[A_0 \rightarrow A_i]$. Let Γ_i be the run spelled by the computation branch of $\mathscr{E}_i$ that was simulated by $\mathscr{E}$. Obviously $\Gamma_i = \langle \Theta, \Delta\rangle$ for some Θ such that $\langle \bot 1.\alpha_1,\ldots,\bot 1.\alpha_n, \top\beta_1,\ldots,\top\beta_m\rangle$ is a $\top$-delay of Θ. Hence Γ is a $\top$-delay of Γ_i and, as the latter is a won run of $e[A_0 \rightarrow A_i]$ (because $\mathscr{E}_i \models A_0 \rightarrow A_i$), so is the former. □

24. The universal language

The author could continue in the style of Section 22 and try to come up with a complete list of all valid principles of computability. But the publisher insists that the lengths of submitted papers be finite, so an alternative way has to be found. This section introduces a formal language, called the *universal language*, in which we may want to try to axiomatize the set of valid principles.

This is a comprehensive first-order specification language for computational problems with the same set **Variables** of *variables* as the one we fixed in Part I. A distinguishing feature of this language is that it has two sorts of *problem letters* (called "predicate letters" in classical logic): *elementary letters* and *general letters*. With each (elementary or general) letter is associated an integer $n \geqslant 0$ called its *arity*. Letters are divided into two categories: *non-logical* and *logical*. For each n, there are infinitely many n-ary elementary and general non-logical letters. As for logical letters, there are three of them, all 0-ary: two elementary logical letters $\bot$ and $\top$, and one general logical letter \$. Where L is a (logical or non-logical, elementary or general) n-ary letter and $x_1,\ldots,x_n$ any variables, the expression $L(x_1,\ldots,x_n)$ is called a (logical or non-logical, elementary or general) n-ary *atom*. Thus, each 0-ary letter is, at the same time, an atom. *Formulas* are built from atoms in the standard way using the operators $\neg$, $\wedge$, $\vee$,

$\rightarrow$, $\sqcap$, $\sqcup$, !, ?, $\Rightarrow$, $\forall$, $\exists$, $\bigsqcap$, $\bigsqcup$. Henceforth by "formula" we always mean a formula of the universal language unless otherwise specified.

The definition of the *scope* of a given occurrence of a logical operator is standard. For quantifiers we also need a more restricted concept of scope called immediate scope. Let F be a formula, O a particular occurrence of a certain subexpression in F, x a variable, and $\mathcal{Q}$ one of the quantifiers $\forall, \exists, \bigsqcap, \bigsqcup$. We say that O is in the *immediate scope* of $\mathcal{Q}x$ in F iff F has a subformula $\mathcal{Q}xG$ such that O is in G and, if G has a subformula $\mathcal{Q}'xG'$ where $\mathcal{Q}'$ is a quantifier, then O is not in G'. Example: where x and y are distinct variables, the first occurrence of $P(x, y)$ in $\forall x \exists y(P(x, y) \sqcap Q \rightarrow \bigsqcap x P(x, y))$ is in the immediate scope of $\forall x$ as well as $\exists y$; the second occurrence is in the immediate scope of $\bigsqcap x$ as well as $\exists y$; it is in the scope but not in the immediate scope of $\forall x$ because $\forall x$ is "overridden" by $\bigsqcap x$. The definition of what a *free occurrence* of variable x means is also standard: such an occurrence is free iff it is not in the scope of $\forall x, \exists x, \bigsqcap x$ or $\bigsqcup x$; otherwise it is *bound*.

In the spirit of Convention 7.3 we will sometimes fix a certain tuple $x_1, \ldots, x_n$ of distinct variables for a given formula F and write F as $F(x_1, \ldots, x_n)$. Then the expression $F(y_1, \ldots, y_n)$ will stand for the result of replacing all free occurrences of $x_1, \ldots, x_n$ in $F(x_1, \ldots, x_n)$ by $y_1, \ldots, y_n$, respectively. It should be noted that when representing F as $F(x_1, \ldots, x_n)$, we usually do not mean that $x_1, \ldots, x_n$ are all the free variables of F, nor do we mean that each of these variables has a free occurrence in F.

Definition 24.1. An *interpretation* is a function * that assigns, to every elementary (resp. general) n-ary letter L, an elementary (resp. static) game $^*L$ with an attached tuple of n (distinct) variables such that the following conditions are satisfied:

1. $^*\bot = \bot$ and $^*\top = \top$.
2. $^*\$$ is a standard universal problem of the universe

$$\{C \mid C \text{ is a tuple-instantiation of } {}^*L \text{ for some general non-logical letter } L\}.$$

Where L is an n-ary letter with $^*L = A(x_1, \ldots, x_n)$ and $c_1, \ldots, c_n$ are constants, the expression "$L(c_1, \ldots, c_n)$" is considered a *name* of the problem $A(c_1, \ldots, c_n)$, and the elements of the above universe are assumed to be arranged in a sequence according to the lexicographic order of their names.

When B is the base of $^*\$$ (as a standard universal problem), interpretation * is said to be *B-based*.

Definition 24.2. For a formula F, an *F-admissible interpretation* is an interpretation * such that, for each n-ary letter L with $^*L = A(x_1, \ldots, x_n)$, the following two conditions are satisfied:

1. $A(x_1, \ldots, x_n)$ does not depend on any of the variables that are not among $x_1, \ldots, x_n$ but occur in F.
2. If, for some $y_1, \ldots, y_n$ and $1 \leqslant i \leqslant n$, F has an occurrence of the atom $L(y_1, \ldots, y_n)$ that is in the immediate scope of $\forall y_i$ or $\exists y_i$, then $A(x_1, \ldots, x_n)$ is unistructural in x_i.

We turn such an interpretation * into a function that assigns a problem G^* to each subformula G of F as follows:

- Where L is an n-ary letter with ${}^*L = A(x_1,\ldots,x_n)$ and $y_1,\ldots,y_n$ are any variables, $(L(y_1,\ldots,y_n))^* = A(y_1,\ldots,y_n)$;
- $(\neg G)^* = \neg(G^*)$; similarly for $!, ?$;
- $(G \to H)^* = G^* \to H^*$; similarly for $\Rightarrow$;
- $(G_1 \wedge \cdots \wedge G_n)^* = G_1^* \wedge \cdots \wedge G_n^*$; similarly for $\vee, \sqcap, \sqcup$;
- $(\bigsqcap x G)^* = \bigsqcap x(G^*)$; similarly for $\bigsqcup, \forall, \exists$.

When saying "*an interpretation of* F", we mean the problem F^* for some F-admissible interpretation *. When * is an F-admissible interpretation with $F^* = A$, we say that * *interprets* F *as* A. The same terminology applies to letters: we will say that interpretation * interprets letter L as the problem ${}^*L$.

Thus, (admissible) interpretations turn formulas of the universal language into particular problems, interpreting elementary atoms as elementary problems, general atoms as any problems, and logical operators as the same-name game operations. The main reason for having two sorts of atoms in the language is to make it more expressive: the point is that whether a given principle of computability is valid may greatly depend on whether the problems it involves are elementary or not; since our language can capture the difference between elementary and general (not necessarily elementary) problems, it allows us to describe computability principles at a finer level. Another reason for having elementary atoms along with general atoms is that, as we are going to see in the next section, this can make the job of axiomatizing the logic easier.

Each interpretation induces a natural problem universe—the one consisting of all tuple-instantiations of the interpretations of general letters. According to the above definition, \$ is interpreted as a universal problem of that universe. Adding to the language an "elementary counterpart" of \$—i.e. an elementary logical letter that would be interpreted as a universal problem of the universe that consists of all tuple-instantiations of the interpretations of elementary letters—would not make much sense. Indeed, $\bot$ is an elementary letter and hence the game $\bot$ $(= {}^*\bot)$ is in the above universe. Therefore, as we noted in the previous section, any universal problem of such a universe would be equivalent to $\bot$ and hence equal to $\bot$ because for elementary games equivalence means equality. Thus, $\bot$ in fact acts as what can be considered an elementary counterpart of \$. Generally, the concept of universal problem is interesting and non-trivial only in the context of non-elementary games.

Notice that our definition of interpretation does not insist that an n-ary letter be interpreted as an n-ary game. There are no strong reasons for imposing this requirement. On the other hand, with such a requirement generality would be lost, for we would no longer be able to capture infinitary problems. Some generality would be lost even if we agreed to restrict our considerations to finitary games only. For example, in order to show that all finitary problems of the form $A \vee \neg A$ are computable, we would have to prove the validity of each of the infinitely many formulas: $P \vee \neg P$, $Q(x) \vee \neg Q(x)$, $R(x,y) \vee \neg R(x,y), \ldots$, corresponding to the infinitely many different possible finite arities for A. But in our approach, which allows the 0-ary general letter P to be interpreted

as any (constant or non-constant, finitary or infinitary, elementary or non-elementary) problem, just showing the validity of $P \vee \neg P$ would be sufficient to automatically conclude that all games of the corresponding form are computable no matter what their arities are.

Thus, the game ${}^*L = A(x_1, \ldots, x_n)$ for an n-ary letter L may depend on some hidden variables beyond its explicit variables $x_1, \ldots, x_n$. According to condition 1 of Definition 24.2, however, those hidden variables should remain hidden all the time and never explicitly occur in formulas. This sort of a restriction is necessary to prevent certain unpleasant collisions of variables. For example, if L is a unary letter interpreted as $A(x)$ where $A(x)$ is the predicate $x > y$, y would be a hidden variable. If y was allowed to appear in subformulas of a given formula, then, even though the two (sub)formulas $L(x)$ and $L(y)$ would "look similar", they would represent rather different predicates: $L(x)$ represent $x > y$ while $L(y)$ represent $y > y$ which is nothing but $\bot$, so that certain classically valid principles such as $\exists x L(x) \rightarrow \exists y L(y)$ would fail. Another example of a pair of formulas that look similar but whose interpretations would be far from being similar is $\forall z A(x)$ and $\forall y A(x)$. All unsettling examples of this sort, however, would involve formulas that contain the hidden variable y. Condition 1 amounts to excluding such formulas from the language.

And the purpose of condition 2 of Definition 24.2 is to guarantee that F^* is always defined. The point is that, unlike any other logical operators, the game operations represented by $\forall x$ and $\exists x$ may be undefined for some arguments—in particular, arguments that are not x-unistructural. It can be verified by induction on complexity that, as long as both conditions of Definition 24.2 are satisfied and $\forall x G$ or $\exists x G$ is a subformula of F, the game G^* is (defined and) x-unistructural, so that $\forall x G^*$ and $\exists x G^*$ are defined.

Definition 24.3. A formula of the universal language is said to be *valid* iff every interpretation of it is winnable.

Thus, valid formulas represent valid principles of computability—they can be thought of as schemata of "always computable" problems. Every non-trivial completeness result —result establishing that none of the formulas from (or beyond) a given class is valid— will carry negative information in that it will establish some limitations of computability. The strength of such negative information, however, would generally greatly depend on what kind of interpretations are used in counterexamples when proving non-validity. First and foremost, we do not want those counterexamples to appeal to infinitary or indetermined problems. As this was pointed out in the proof of Proposition 22.1(2), a (positive) infinitary game may fail to be winnable just because it is impossible to ever finish reading all the relevant input information from the valuation tape, which would generally make powerless not only algorithmic strategies but any strategies as well that do not have full actual knowledge of the infinite contents of the valuation tape. Similarly, all indetermined problems are incomputable because they simply have no solutions (even by God), so that indetermined counterexamples will never illustrate any limitations specific to algorithmic methods. The following conjecture immensely strengthens all possible non-trivial completeness results:

Conjecture 24.4. *If a formula F of the universal language is not valid, then F^* is not winnable for some F-admissible interpretation * that interprets all atoms as finitary, determined, strict, unistructural problems.*

Conjecture 25.4 (clause 2) of the following section strengthens the above statement for the exponential-free fragment of the universal language by further restricting the range of * to finite-depth problems. So does Conjecture 25.8 for another natural and rich fragment of the language. It would be interesting to know if such a restriction extends to the full universal language as well. Most likely it does. Very good if so: the more restricted the range of interpretations is, the more general, informative, interesting and strong the corresponding completeness/non-validity results are.

25. The universal logic

Definition 25.1. The *universal logic* is the set of all valid formulas of the universal language.

Here is the naturally arising main open problem in computability logic: *Is the universal logic recursively enumerable and, if yes, does it have a reasonable axiomatization?*

In view of the extraordinary expressive strength of the universal language, solving this big open problem in full generality—if the answer is positive—is a presumably challenging task. A reasonable way to go would be not to try to attack the problem at once, but approach it step-by-step by exploring, in an incremental way, fragments of the universal logic obtained by imposing some natural restrictions on its language.

One of the natural fragments of the universal logic is obtained by forbidding in its language general letters (including \$) and the operators that do not preserve the elementary property of games, which, in view of Theorem 14.1, only leaves the operators $\neg, \wedge, \vee, \rightarrow, \forall, \exists$, so that we get exactly the language of *classical predicate calculus* **CL**. This fragment can be called the *logic of elementary problems* as all of its formulas (and only those formulas) represent elementary problems. A simple analysis of our definitions of the above game operations reveals that, when applied to elementary games (predicates), their semantical meaning is exactly classical, i.e. solvability of an interpretation F^* of a formula F of the above fragment means nothing but its classical truth. Of course, the traditional concept of classical validity and the corresponding soundness proofs for **CL** deal only with the case of finitary predicates and require that every n-ary letter be interpreted as an n-ary predicate. The more general and relaxed approach that we have taken, however, hardly creates any difference: condition 1 of Definition 24.2 guarantees that all of the possible hidden variables on which the interpretation of a given atom may depend will remain totally neutral and harmless, so that **CL** continues to be sound. This fact, together with Gödel's completeness theorem for **CL**, yields the following proposition, establishing that the universal logic is a conservative extension of **CL** so that the latter is a natural syntactic fragment of the former:

Proposition 25.2. $\mathbf{CL} \vdash F$ *iff* F *is valid* (*any formula* F *of the language of* **CL**).

Moreover, if $\mathbf{CL} \nvdash F$, *then* F^* *is not winnable for some* F*-admissible interpretation* * *that interprets all atoms as finitary predicates.*

Another natural fragment of the universal logic, that we can call the *logic of finite-depth problems*, is obtained by forbidding in the universal language the only non-finite-depth operators $!, ?, \Rightarrow$. Let us call this language the *finite-depth language* and its formulas *finite-depth formulas*. Obviously the logic of finite-depth problems is not decidable as it contains classical predicate logic. However, according to Conjecture 25.4 stated below, this logic is recursively enumerable and reasonably axiomatizable. This is a very strong positive statement taking into account that the finite-depth language, even though smaller than the universal language, is still by an order of magnitude more expressive than the language of classical logic, and in fact it is more expressive than the simple union of the languages of classical and additive-multiplicative linear logics.

To axiomatize the logic of finite-depth games, we need some preliminary definitions. In this paragraph "formula" always means "finite-depth formula". Understanding $A \rightarrow B$ as an abbreviation for $\neg A \vee B$, a *positive* (resp. *negative*) occurrence of a subformula is an occurrence that is in the scope of an even (resp. odd) number of occurrences of $\neg$. By an *elementary formula* here we mean a formula that does not contain general letters, $\sqcap, \sqcup, \bigsqcap$ or $\bigsqcup$; a *surface occurrence* of a subformula is an occurrence that is not in the scope of $\sqcap, \sqcup, \bigsqcap$ or $\bigsqcup$; and the *elementarization* of a formula F is the result of replacing in F all positive (resp. negative) surface occurrences of general atoms—including \$—by $\bot$ (resp. $\top$), all (positive or negative) surface occurrences of the form $G_1 \sqcap \cdots \sqcap G_n$ or $\bigsqcap xG$ by $\top$, and all (positive or negative) surface occurrences of the form $G_1 \sqcup \cdots \sqcup G_n$ or $\bigsqcup xG$ by $\bot$. We say that a formula F is *stable* iff its elementarization is a valid formula of classical logic. Otherwise F is *instable*.

Definition 25.3. **FD** is the system in the finite-depth language given by the following rules of inference:

A. $\vec{H} \vdash F$, where F is stable and $\vec{H}$ is a (minimal, finite) set of formulas satisfying the following conditions:

(i) If F has a positive (resp. negative) surface occurrence of a subformula $G_1 \sqcap \cdots \sqcap G_n$ (resp. $G_1 \sqcup \cdots \sqcup G_n$), then, for each $i \in \{1, \ldots, n\}$, $\vec{H}$ contains the result of replacing this occurrence in F by G_i;

(ii) If F has a positive (resp. negative) surface occurrence of a subformula $\bigsqcap xG(x)$ (resp. $\bigsqcup xG(x)$), then $\vec{H}$ contains the result of replacing this occurrence in F by $G(y)$ for some y not occurring in F.

B1. $F' \vdash F$, where F' is the result of replacing in F a negative (resp. positive) surface occurrence of a subformula $G_1 \sqcap \cdots \sqcap G_n$ (resp. $G_1 \sqcup \cdots \sqcup G_n$) by G_i for some $i \in \{1, \ldots, n\}$.

B2. $F' \vdash F$, where F' is the result of replacing in F a negative (resp. positive) surface occurrence of a subformula $\bigsqcap xG(x)$ (resp. $\bigsqcup xG(x)$) by $G(y)$; here y is an arbitrary variable such that neither the above occurrence of $\bigsqcap xG(x)$ (resp. $\bigsqcup xG(x)$)

in F nor any of the free occurrences of x in $G(x)$ are in the scope of $\forall y$, $\exists y$, $\bigsqcap y$ or $\bigsqcup y$.

C. $F' \vdash F$, where F' is the result of replacing in F two—one positive and one negative—surface occurrences of some n-ary general letter by an n-ary non-logical elementary letter that does not occur in F.

D. $F' \vdash F$, where F' is the result of replacing in F a negative surface occurrence of \$ by a general n-ary atom $L(x_1,\ldots,x_n)$ such that the above occurrence of \$ in F is not in the scope of $\forall x_i$ or $\exists x_i$ (any $1 \leqslant i \leqslant n$).

Axioms are not explicitly stated, but note that the set $\vec{H}$ of premises of Rule **A** can be empty, in which case the conclusion of that rule acts as an axiom.

Conjecture 25.4. **FD** $\vdash F$ *iff* F *is valid* (*any finite-depth formula* F). *Moreover*:

(a) *There is an effective procedure that takes an* **FD***-proof of a formula* F *and constructs an EPM that wins* F^* *for every* F*-admissible interpretation* *.

(b) *If* **FD** $\nvdash F$, *then* F^* *is not winnable for some* F*-admissible interpretation* * *that interprets all atoms as finite-depth, finitary, strict, unistructural problems.*

As long as we only consider interpretations that interpret atoms as finite-depth problems, all problems expressible in the language of **FD** will be finite-depth, because all the operators allowed in it preserve the finite-depth property of games. In view of clause (b) of the above conjecture, the completeness of **FD** (and of course its soundness as well) will still be retained in this case. This is what allows us to call **FD** "the logic of finite-depth problems". For the same reason, **FD** is also a logic of perifinite-depth problems. A preliminary version of this logic was studied in [15,16].

Obviously **FD** is recursively enumerable. One can also show that

Claim 25.5. *The* $\forall,\exists$*-free fragment of* **FD** *is decidable.*

This is a nice and perhaps not very obvious/expected fact, taking into account that the above fragment of **FD** is still a first order logic as it contains the quantifier-type operators $\bigsqcap$ and $\bigsqcup$. This fragment is also natural as it gets rid of the only operators of the universal language that produce games with imperfect information. For this reason, it can be called the *logic of perfect-information finite-depth games*. Since Claim 25.5 is of interest only modulo Conjecture 25.4, we let its proof wait till Conjecture 25.4 is officially verified, and hence label it "claim" rather than "proposition".

Below come two other claims, stated without proofs for similar reasons.

An *elementary-base* (resp. *general-base*) *formula* of the universal language is a formula not containing general (resp. elementary) letters. Correspondingly, we will call the restriction of a logic to elementary-base (resp. general-base) formulas the *elementary-base* (resp. *general-base*) *fragment* of that logic.

Claim 25.6. *The elementary-base fragment of* **FD**, *that can be called the* elementary-base logic of finite-depth games, *is exactly*[11] *the logic* **L** *introduced in* [14].

[11] Ignoring the minor differences between the languages of **L** and the elementary-base fragment of **FD**.

In [14], **L** was shown to be sound and complete with respect to a semantics that was technically rather different from our present semantics. [12] The underlying philosophy for that semantics was understanding formulas as physical *tasks* (rather than computational problems). See Section 26 for a discussion of the task intuition behind game semantics.

Claim 25.7. *The general-base,* $\forall, \exists, \$$*-free fragment of* **FD** *is exactly the logic* **ET** *introduced in* [12].

ET is a decidable first-order system defined in [12] syntactically but not axiomatically. It was proven to be sound and complete with a (yet another) semantics in mind that also was technically quite different from our present semantics—first of all because, as noted in Section 3, it only dealt with strict games. **ET**, in turn, was shown to be a conservative extension of the (only) two syntactic fragments of the logic induced by Blass's [5] semantics for which decidability or completeness proofs are known—specifically, the multiplicative propositional fragment and the additive fragment (sequents containing only additive operators). The fact that different semantics, defined in different technical terms with different underlying motivations/philosophies and by different authors, still yield the same relevant fragments of the universal logic, is one of the strong indications that the logic is natural and something indeed "real" stands behind it. Conjecture 25.8 stated shortly reinforces this view.

Another interesting fragment of the universal language, that can be thought of as the language of Heyting's intuitionistic predicate calculus **INT**, is the universal-base fragment of the language whose only logical operators are $\sqcap, \sqcup, \bigsqcap, \bigsqcup, \Rightarrow$ and $\$$, corresponding to the intuitionistic conjunction, disjunction, universal quantifier, existential quantifier, implication and absurd, respectively. Let us call this fragment of the universal language the *intuitionistic language*. The following conjecture, if proven true, would signify a convincing materialization of Kolmogorov's (1932) well-known but rather abstract thesis according to which intuitionistic logic is (or should be) a logic of problems.

Conjecture 25.8. **INT** $\vdash F$ *iff* F *is valid* (*any formula* F *of the intuitionistic language*). *Moreover*:

(a) *There is an effective procedure that takes an* **INT***-proof of a formula* F *and constructs an EPM that wins* F^* *for every* F*-admissible interpretation* *.

(b) *If* **INT** $\nvdash F$, *then* F^* *is not winnable for some* F*-admissible interpretation* * *that interprets all atoms as finite-depth, finitary, strict, unistructural problems.*

Gödel in 1933 suggested a certain 'provability' interpretation for **INT** in a quite abstract sense. That abstract idea was refined and successfully materialized by Artemov [2] for the propositional fragment of **INT**. It would be interesting to understand connections between the computability and provability interpretations of **INT**

[12] The propositional version of **L**, called **TSKp**, was studied in [13].

and, in particular, see if Artemov's proof polynomials have good computability-logic counterparts in terms of EPM-strategies and their combinations.

26. Applications in knowledgebase and resourcebase systems

The main motivation for introducing the formalism and semantics presented in this paper was computability-theoretic. The scope of the significance of this study, however, is not limited to computability theory. In this section we will make a digression from our central line and show how and why our logic is of interest in other, more applied areas of computer science, such as knowledgebase theory or the part of AI that studies planning and action.

The reason for the failure of the principle $F \sqcup \neg F$, as we saw in Section 22, was that the problem represented by this formula may have no algorithmic solution. The reason for the failure of this principle in a knowledgebase system can, however, be much simpler. Such a system may fail to win the game

$$Female(Sandy) \sqcup \neg Female(Sandy)$$

not because there is no algorithmic solution to it (of course there is one), but just because the system simply lacks sufficient knowledge that would allow it to determine the gender of Sandy. On the other hand, any system would be able to "solve" the problem

$$Female(Sandy) \vee \neg Female(Sandy)$$

as this is an automatically won elementary game so that there is nothing to solve at all. Obviously the knowledge expressed by $A \sqcup B$ is generally stronger than the knowledge expressed by $A \vee B$. Yet the language of classical logic fails to capture this difference and—within the framework of the traditional approaches ([20], etc.)—the modal "know that" operator $\Box$—with all the complications it entails (such as, say, the messiness of the semantics and the non-semidecidability of the corresponding logic)—is needed to describe the stronger knowledge $A \sqcup B$, which would translate as $\Box A \vee \Box B$. This simple example demonstrates the main point the author wants to convey here: our universal language can serve not only as a specification language for computational problems, but also as a convenient and comprehensive knowledge specification language, yet without any special knowledge operators in it!

In this context, formulas of our language represent (interactive, multistep, complex) queries. A formula whose main operator is $\sqcup$ or $\bigsqcup$ can be understood as a question asked by the user, and a formula whose main operator is $\sqcap$ or $\bigsqcap$ a question asked by the system.[13] Consider the problem $\bigsqcap d \bigsqcup m Cures(m,d)$, where $Cures(m,d)$ means "medicine m cures disease d". This formula is the following question asked by the system: "For what disease do you want me to find a cure?". The user's response can be "Pneumonia". This move takes us to the position $\bigsqcup m Cures(m, Pneumonia)$. This

[13] The general question/answer intuition behind linear logic has been noticed long ago. See [6].

is now a question asked by the user: "What medicine cures pneumonia?". The system's response can be "Penicillin", thus taking us to the final position *Cures*(*Penicillin*, *Pneumonia*). The system has been successful if penicillin is really a cure for pneumonia.

Multiplicative combinations of formulas may represent queries of a more complex structure—parallel queries where both of the participants may have simultaneous legal questions; $!F$ is a query that can be re-used an arbitrary number of times, etc.

To get a better feel of multiplicatives as query operations, let us consider an example involving $\rightarrow$. As we did not have enough chances to demonstrate the usage of the blind quantifier $\forall$, we will throw it into the example as well. Let us make *Cures* a 3-ary predicate, with $Cures(m, p, d)$ meaning "Medicine m will cure patient p of disease d". And let $Bloodtype(b, p)$ mean "b is the blood type of patient p". Imagine a knowledgebase system that can determine, for each disease, a medicine that cures a patient of that disease, provided that the knowledgebase is told what that patient's blood type is. A selection of the right medicine depends only on the blood type and the disease, so that the system does not need to know who the patient really is. This can be captured by quantifying the variable p with $\forall$ rather than $\sqcap$. The query that the system can solve can then be expressed by

$$\forall p \sqcap d \left(\sqcup b Bloodtype(b, p) \rightarrow \sqcup m \ \ Cures(m, p, d) \right).$$

After the user specifies a disease (without having explicitly specified a patient), the system will request to specify the blood type of the patient and, provided that it receives a correct answer to this counterquery, it can correctly answer the question about what medicine can cure the patient of the specified disease.

A knowledgebase system and its power can be described by some basic set $\{Q_1, \ldots, Q_n\}$ of queries that the system can solve. This may involve atomic elementary formulas expressing factual knowledge, such as $Female(Sandy)$, or non-atomic elementary formulas expressing general knowledge, such as $\forall x \forall y(x + y = y + x)$; it can also include non-elementary formulas such as $\sqcap x \sqcup y(y = x^2)$, expressing the ability of the knowledgebase to compute the square function (expressing the same without additives would take an infinite number of atomic statements: $0^2 = 0$, $1^2 = 1$, $2^2 = 4, \ldots$). These formulas can be called the *informational resources* of the knowledgebase. Since informational resources typically can be used and reused an arbitrary number of times, they should be prefixed with the operator $!$. Finally, since all of these resources can be used in parallel and independently, we can combine them with $\wedge$, thus getting one formula

$$KB = !Q_1 \wedge \cdots \wedge !Q_n,$$

fully characterizing the knowledgebase and with which the knowledgebase system can be identified. Assuming that the system has the appropriate logic built into it and that the whole specific (non-logical) knowledge that the system possesses is the knowledge captured by *KB*, the problem whether the system can solve query A is then equivalent to the problem whether $KB \rightarrow A$ is logically valid.

The notion of validity needs to be reconsidered here though. Validity of F in the sense of Definition 24.3 means existence of a solution to F^* for each (F-admissible) interpretation *. It can be the case, however, that different interpretations require different solutions, so that it is impossible to solve the problem without knowing the interpretation, i.e. the *meaning*, of the atoms involved, as with $Female(Sandy) \sqcup \neg Female(Sandy)$. As long as we assume that KB captures the whole non-logical knowledge of the system, the system would/should be able to determine the gender of Sandy only if this information is explicitly or implicitly contained in KB. Assuming that the system has no extra (not contained in KB) knowledge means nothing but assuming that it has no knowledge of the interpretation of atoms except that that interpretation satisfies KB. To capture this intuition, we need a new concept of validity of a formula F that would mean the possibility to solve F^* without knowing the interpretation *, i.e. to play F^* in a way that would be successful for any possible interpretation *. The formal concept that we call uniform validity (the term "uniform" borrowed from [1]) fits the bill:

Definition 26.1. A formula F of the universal language is said to be *uniformly valid* iff there is an EPM that, for every F-admissible interpretation *, wins F^*.

Then, ability of a knowledgebase KB to solve a query Q means nothing but uniform validity of $KB \rightarrow Q$.

Notice the similarity between Definitions 26.1 and 24.3 after disabbreviating in the latter winnability as existence of a winning machine. In both cases 'machine' is quantified existentially and 'interpretation' universally. The only difference is in the order of the quantification, which is '$\forall$ *interpretation* $\exists$ *machine*' in Definition 24.3 and '$\exists$ *machine* $\forall$ *interpretation*' in Definition 26.1. Our concept of uniform validity is close, in its spirit, to validity in Abramsky and Jagadeesan's [1] sense. We wish to think that the former is more clear both technically and intuitively. Validities in Lorenzen's [18] and a number of other authors' sense who studied game semantics, can also be added to the list of uniform-validity-style concepts. What is common to all those concepts is that validity there does not really mean truth in every particular case, but is rather treated as a basic concept in its own rights. While certainly interesting in many aspects including the above-demonstrated applicability, the uniform-validity-style concepts are mathematically less interesting than the "normal" concepts (valid = always true) of validity, such as the one we introduced in Definition 24.3 or validity in Blass's [5] sense. In particular, replacing validity by uniform validity can dramatically simplify certain completeness proofs. Simplicity is generally good, but in this case it is a symptom of the fact that such completeness proofs are mathematically less informative, the fact that apparently has not been fully recognized by the logic community. Consider the formula $P \sqcup \neg P$. Its non-validity in Blass's [5] sense means existence of indetermined games, which is a very interesting and non-trivial mathematical fact. The non-validity of this formula in our present sense, or in the sense of Japaridze [12], means existence of solvable-in-principle yet algorithmically unsolvable problems, which is a fact that had been remaining unknown to the mankind until a few decades ago. As for the non-uniform-validity of $P \sqcup \neg P$, it comes for free: of course there is no way to choose the right (true) disjunct as long as atoms are treated, using van Benthem's [3] words,

as black boxes that cannot be opened. This kind of a non-validity "proof" for $P \sqcup \neg P$ hardly carries any mathematical information. The concept of validity studied in [14] and with respect to which logic **L** (see Claim 25.6) was shown to be complete, is also a uniform-validity-style concept. For this reason, even though the completeness proof in [14] was everything but trivial and took quite a few pages, the author still does not take too much mathematical pride in it.

Anyway, what is the logic induced by uniform validity and how does it compare with the logic induced by validity (the universal logic)? There are reasons to expect that these two logics are the same:

Conjecture 26.2. *A formula of the universal language is valid if and only if it is uniformly valid.*

While the "if" part of the above statement automatically follows from the definitions, the "only if" part would require a non-trivial proof. Notice that clauses (a) of Conjectures 25.4 and 25.8 in fact already contain the correctness statement for Conjecture 26.2 restricted to the finite-depth and the intuitionistic fragments of the universal language.

In view of the above conjecture, we can automatically transfer all the soundness/completeness statements in the style of Section 25.1 to the context of knowledgebase systems.

This context can be further expanded by generalizing knowledgebases to resourcebases. We have already referred to the conjuncts of KB as "informational resources". The language of our logic can be used to represent physical resources as well. A (physical or informational) resource can be understood as a *task* that is carried out for the agent by its adversary. The term "task" is a synonym of "problem" which we prefer to use in this new context. In the case of informational resources, the task is simply to correctly answer questions. With physical resources, the task may also involve changing the physical world. Elementary atoms can be understood as elementary tasks, such as "Destroy the vampire!", which is considered accomplished (and the corresponding elementary problem solved) if the vampire is eventually destroyed, i.e. the narrative counterpart "The vampire has been destroyed" of the imperative "Destroy the vampire!" becomes true. In general, accomplishing an elementary task T means making T (or its narrative counterpart) true, and when and by whom it was made true does not matter. The only difference between the (informational) elementary task "Sandy is a female" and the (physical) elementary task "Destroy the vampire!" is that the accomplished/failed (won/lost) value of the former is (pre)determined whereas for the latter it is not, so that while there is nothing the agent can do to affect the outcome of the play over "Sandy is a female", the outcome of the elementary game "Destroy the vampire" can still be manageable. Our formal semantics, however, does not make any distinction between informational and physical tasks and, as a simple example at the end of this section may convince the reader, there is no real need in making such a distinction.

There is very natural task/resource intuition behind our game operations, e.g.,

Destroy the vampire $\sqcap$ *Kill the werewolf*

is the task which is accomplished—i.e. the corresponding game won—if the environment made the move *left* (requested to destroy the vampire) and the vampire was really destroyed, or the move was *right* and the werewolf was really killed. Notice that the agent will have to carry out only one of the two subtasks, so accomplishing the task expressed by the above $\sqcap$-conjunction is easier than accomplishing the task

$$\textit{Destroy the vampire} \ \wedge \ \textit{Kill the werewolf},$$

which obligates the agent to carry out both of the two subtasks, for which he may not have sufficient resources, even if he can successfully carry out any particular one of those two.

The operation $\rightarrow$ now becomes an adequate formalization of our intuition of reducing one task to another. To get a feel of $\rightarrow$ as a task reduction operation, it would be sufficient to look at the task

$$\textit{Give me a wooden stake} \ \sqcap \ \textit{Give me a siver bullet}$$
$$\rightarrow \ \textit{Destroy the vampire} \ \sqcap \ \textit{Kill the werewolf},$$

which describes the situation of an agent who has a mallet and a gun as well as sufficient time, energy and bravery to chase any one of the two monsters, and only needs a wooden stake and/or silver bullet to complete the noble mission *Destroy the vampire* $\sqcap$ *Kill the werewolf*.

With this task/resource intuition in mind, our logic can be used as a logic of resource-oriented planning and action. The resources—both informational and physical—that the agent possesses can be $\wedge$-combined in the *resourcebase RB* of the agent. Then the question of possibility to successfully carry out the goal task G reduces to the question of uniform validity of the formula $RB \rightarrow G$.

The conjuncts of RB that represent physical resources, unlike those representing informational resources, generally would not be prefixed with $!$, as typical physical resources are not reusable. For instance, (the task performed by) a ballistic missile can be described by $\bigsqcap tHit(t)$, expressing its ability to hit any one target t specified by the user. Having a finite number n of such resources can be expressed by taking the $\wedge$-conjunction of n identical conjuncts $\bigsqcap tHit(t)$. However, including $!\bigsqcap tHit(t)$ in the resourcebase would mean having an infinite number of missiles!

To see the potential of our logic as a logic of planning and action, let us consider the following simple planning problem borrowed from [14]:

There are several sorts of antifreeze coolant available to the agent, and his goal is to

0. Fill the radiator of the car with a safe sort of coolant.

This is what the agent knows:

1. A sort of coolant is safe iff it does not contain acid.

2. A sort of coolant contains acid iff the litmus paper turns red when it is used to test the coolant.

3. At least one of the two sorts of coolant: c_1, c_2 is safe.

This is what the agent has:

4. A piece of litmus paper which he can use to test any one particular sort of coolant, and he also can

5. Fill the radiator using any one particular sort of coolant.

Can the agent accomplish the goal described in item 0 if all the informational and physical resources he has are the ones described in items 1–5, and if yes, how? Let us fix the following four elementary tasks: $A(x)$—"coolant x contains acid"; $S(x)$—"coolant x is safe"; $R(x)$—"the litmus paper turns red when used for testing coolant x"; $F(x)$—"fill the radiator with coolant x". Then, writing $X \leftrightarrow Y$ to abbreviate $(X \rightarrow Y) \wedge (Y \rightarrow X)$, the tasks/resources expressed in items 0–5 can be specified as follows:

0. $\exists x(S(x) \wedge F(x))$.
1. $\forall x(S(x) \leftrightarrow \neg A(x))$.
2. $\forall x(A(x) \leftrightarrow R(x))$.
3. $S(c_1) \vee S(c_2)$.
4. $\sqcap x(R(x) \sqcup \neg R(x))$.
5. $\sqcap x F(x)$.

To accomplish the goal task (0) having resources (1)–(5) means nothing but to accomplish the conditional task $RB \rightarrow (0)$ unconditionally, where $RB = (1) \wedge (2) \wedge (3) \wedge (4) \wedge (5)$. Thus RB combines informational (1)–(3) and physical (4)–(5) resources of the agent together. There is no need to formally distinguish between informational and physical resources and, in fact, not having to distinguish between them is very convenient as, after all, it is not always clear where to draw the line. For example, we might as well have considered (4) an informational resource because the accomplished/failed values of its atoms are predetermined. Unlike typical informational resources, however, this one cannot be prefixed with $!$ as the litmus paper is not reusable. As for (1)–(3), they are elementary informational resources and prefixing them with $!$ is unnecessary for, as we remember (Claim 22.8), every elementary problem B is equivalent to $!B$.

Here is a strategy for carrying out the above task. At first, the agent specifies x as (the value of) c_1 in (4). The meaning of this action is to use the litmus paper for testing coolant c_1. The environment should react by making one of the moves *left*, *right* in (4). This step corresponds to observing, by the agent, whether the litmus paper turns red or not. The next action is contingent on what the reaction of the environment to the previous move was. If the reaction was *left*, then the agent specifies x in (5) as c_2. This means having the radiator filled with c_2. And if the reaction was *right*, then the agent specifies x in (5) as c_1, which means having the radiator filled with c_1. It can be seen that this strategy guarantees success: the radiator will be filled with safe coolant as long as none of the agent's resources (1)–(5) fail to do their job. This strategy will be successful no matter what the meanings of A, S, R, F, c_1, c_2 really are. Thus, the goal task (0) can be accomplished by an agent who does not have any extra physical or informational resources except those described in RB, which means nothing but uniform validity of $RB \rightarrow (0)$.

Notice that both *RB* (with c_1, c_2 understood as free variables) and (0) are specified in the language of **FD**—in fact, in its elementary-base fragment. By the soundness and completeness conjecture for **FD**, in conjunction with Conjecture 26.2, $RB \rightarrow (0)$ is uniformly valid if and only if $\mathbf{FD} \vdash RB \rightarrow (0)$. So, had our ad hoc methods failed to find an answer, the existence of a successful strategy could have been established by showing that the task we are considering is provable in **FD**. Of course, just knowing that a task-accomplishing strategy exists, would have very little practical value unless we could constructively find such a strategy. No problem: according to clause (a) of Conjecture 25.4, such a strategy can be effectively constructed from an **FD**-proof of the task. In general, uniform validity of a formula of the universal language means the possibility to accomplish the corresponding task by an agent who has nothing but its intelligence (no physical or non-logical informational resources, that is) for carrying out tasks.

As it was shown in [14], this sort of resourcebase planning systems are immune to the notorious *frame problem* (see [21]) and neutralize the *knowledge preconditions problem* first recognized in [19]. This greatly increases the appeal of the universal logic as a logic of planning and action.

27. From syntax to semantics or vice versa?

The normal way to introduce a new logic is to proceed from a semantics to a syntax (by syntax here we mean axiomatization or any other, "similar", sort of description): At the beginning, (1) a philosophy, motivation and intuitions—i.e. an informal semantics—should be presented; then (2) a formal semantics should be elaborated that adequately captures those intuitions, and only after that (3) we can ask the question regarding what the corresponding logic (syntax) is. The way classical logic evolved matches this pattern. And this is exactly the scheme we have followed in this paper. On the other hand, looking back at the history of intuitionistic and linear logics, we can see that both of them jumped from step (1) directly to step (3). That is, a particular deductive system was proposed (step (3)) based on some general intuitions (step (1)) without any exact formal semantics, so that, in the absence of a clear concept of truth or validity, the question regarding whether the proposed system was complete and hence "right", could not even be meaningfully asked. Only retroactively, a number of authors started looking for a missing formal semantics that would justify the proposed deductive system. Technically, it is always possible to find some sort of a formal semantics that matches a given target syntactic construction, but the whole question is how natural and meaningful such a semantics is and whether and how adequately it captures the original intuitions and motivations (step (1)) underlying the logic. Unless the target construction, by good luck, really *is* "the right logic", the chances of a decisive success when following the odd scheme 'from syntax to semantics' could be rather slim.

As this was mentioned, one of the main intuitions associated with intuitionistic logic, apparently equivalent to the more original constructivistic intuitions behind it, is that this is a logic of problems. If our Conjecture 25.8 is true, we deal with what we called a lucky case: **INT**, even though introduced syntactically, indeed turns out to be sound

and complete as a logic of problems. The restriction of our concept of computability to the language of **INT** could be perceived as "intuitionistic truth", as it naturally captures the computational-problem intuition of intuitionistic logic. Unlike Lorenzen's concept of game-semantical validity with its notorious fate, no one can accuse it of being just a target-adjusted, artificially selected technical concept lacking real meaning.

One of the main motivations and intuitions associated with linear logic is that this is a logic of resources. This resource philosophy, however, had never really been formalized as a mathematically strict and intuitively convincing resource semantics. The canonical version of linear logic, just like **INT**, was introduced syntactically rather than semantically—essentially by starting from classical sequent calculus and throwing out the rules that seemed unacceptable from the intuitive, naive resource point of view. It well could be the case that in this process some innocent, deeply hidden principles got victimized as well. Indeed, this is exactly what happened. As this was noted in Section 22, Girard's [10] axiomatization for linear logic is incomplete with respect to our semantics, which probably means that it is incomplete as a resource logic. It has already been pointed out that resources are synonymic (or symmetric) to problems/tasks: what is a problem for one player to solve, is a resource for the other player to use, and vice versa. Hence, our semantics of computational problems is, at the same time, a semantics of computational resources. This claim has been amply supported by our discussions and examples. Moreover, in Section 26 we saw how to extend the context of computational resources to informational and physical resources as well. Just as in the case of intuitionistic logic, the restriction of our concept of computability to the language of linear logic is what could be called "linear-logic truth", understanding linear logic in a broad sense and not identifying it with Girard's axiomatic version.

Beginning from Lorenzen [18], many different sorts of game semantics have been introduced in the literature, pretty often in the course of following the scheme 'from syntax to semantics' in an attempt to justify some already given target syntactic constructions. Among the least (if at all) target-driven and hence most natural and appealing game semantics studied so far is the semantics introduced by Blass [5] for the language of linear logic, already mentioned many times throughout this paper. In fact, the game operations through which Blass interprets the linear-logic connectives in [5], were known to him long time before linear logic was invented, when studying ordering relations on indetermined games in the early 1970s [4].

As Blass's semantics is a close relative of our present game semantics, it is worthwhile to briefly point out some differences, with an accent on the advantages of our approach. Unlike our games, Blass's games, just as the games studied in [12], are strict. To maintain strictness, some additional rules govern multiplicative and exponential combinations of games, such as, say, not allowing $\bot$ to make a move in a $\wedge$-conjunct as long as $\top$ has legal moves in the other conjunct. As we argued in Section 3, strict multiplicatives do not capture the intuition of parallel combinations of games as adequately as free multiplicatives do, where plays in the two components can proceed totally independently from each other. The same can be said about Blass's strict versus our free exponentials. Next, for certain technical reasons—in particular, to avoid a trivialization of the logic—atoms in Blass's semantics have to be interpreted as

indetermined and hence infinite-depth games. This precludes the logic of games from being perceived as a generalization or refinement of classical logic or a reasonable alternative to it, for it is not clear how to interpret classical predicates such as $x > y$ as infinite-depth—let alone indetermined—games. Finally, the requirement of effectiveness for $\top$'s strategies is absent in Blass's approach as it would be redundant there, while this requirement is the whole gist of our approach and, along with the possibility to interpret atoms as elementary or (peri)finite-depth games, turns the logic of games into a comprehensive logic of computability—into something that is no longer "just a game". It would be accurate to say that truth in Blass's [5] sense means positiveness rather than computability.

Despite these differences, Blass's [5] game semantics is the nearest precursor of our present semantics among the semantics studied by other authors. Unfortunately, the former never received a complete treatment, and was abandoned half-way after discovering that it validates certain principles underivable in the canonical version of linear logic or even Affine logic. New efforts [1,11] were mainly directed towards modifying Blass's semantics so that to bring the corresponding logic closer to linear logic. Such efforts achieved a certain degree of technical success, but signified a departure from some of Blass's most natural intuitions. Those modifications are less closely related to our present game semantics than Blass's original semantics.

28. Why study computability logic?

Setting the utilitarian arguments of Section 26 aside, the most obvious answer to the question "Why study the logic of computability?", reflecting the original and primary motivation of the author, was given in Section 1: The property of computability is at least as interesting as the property of truth but, while we have the perfectly clear and well-studied logic of truth—which is classical logic—the logic of computability has never received the treatment it naturally deserves. Anyone critical or skeptical regarding the approach initiated in this paper, would either have to argue that computability is not worth studying, or argue that the two basic concepts around which the approach evolves—those of static games and their winnability—are not really about computability. In the latter case, as a minimum, the skeptic would be expected to make a good case against Theses 4.4 and 18.2.

As it turns out, classical truth is nothing but a special case of computability—in particular, computability restricted to elementary problems (predicates), so that computability logic is a natural conservative extension, generalization and refinement of classical logic and, for this reason, it could come to replace the latter in virtually every aspect of its applications. Whatever we can say or do by the means that classical logic offers, we can automatically also say (without changing the meaning or even the notation) and do with our universal logic. At the same time, the universal logic enriches classical logic by adding very strong constructive elements to its expressive power. This gives us good reasons to try to base applied theories, such as, say, Peano arithmetic, on the universal logic instead of just classical logic. In the ideal case, we would use a sound and complete axiomatization of the universal logic. But even if such

an axiomatization is not found, we could as well use any sound but not necessarily complete axiomatic system, such as linear logic, to whose axioms and rules could be added those of **FD** and **INT** (provided that our conjectures regarding the soundness of these logics are correct, of course). This way we would get a much more expressive, informative and constructive version of the applied theory than its classical-logic-based version. Of course, properly rewriting a classical-logic-based applied theory could be creative work. A standard scheme to follow would be to start with the existing set of non-logical axioms of the theory and then try to replace in those axioms, whenever possible, the non-constructive classical (blind) quantifiers by their constructive counterparts $\sqcap$ and $\sqcup$, as well as (less importantly) $\wedge, \vee$ with the more constructive and informative $\sqcap, \sqcup$. Moreover, it would not be necessary to really replace the old axioms: at the beginning we could leave them untouched for safety, to ensure that no old information is lost or weakened, and simply add to them, in parallel, their constructivized versions, or some totally new axioms or computability-preserving rules specific to that theory; only later, once ensured that an old axiom is no longer independent, we would want to delete it altogether. The process of constructivization of $\forall, \exists$ sometimes may force us to replace $\rightarrow$ with $\Rightarrow$. This would be an option that we would not hesitate to take, because $\Rightarrow$, although not as good as $\rightarrow$, is still reasonably constructive, and $\rightarrow$ is worth being "downgraded" to $\Rightarrow$ for the sake of constructivizing the classical quantifiers. For example, one of the possible ways to constructivize the axiom (scheme) of induction

$$\big(F(0) \wedge \forall x(F(x) \rightarrow F(x+1))\big) \rightarrow \forall x F(x)$$

of Peano arithmetic would be to rewrite it as

$$\big(F(0) \wedge \sqcap x(F(x) \rightarrow F(x+1))\big) \Rightarrow \sqcap x F(x),$$

which can be shown to be always computable. Here we have turned the two occurrences of $\forall$ into $\sqcap$, but the price we paid was replacing the main connective $\rightarrow$ by $\Rightarrow$: one can show that with $\rightarrow$, the formula would no longer represent always-computable problems. In case we want to express the axiom of induction purely in the intuitionistic language, the above formula can be safely rewritten as

$$\big(F(0) \sqcap \sqcap x(F(x) \Rightarrow F(x+1))\big) \Rightarrow \sqcap x F(x)$$

as well. Another possibility—in case we only want to deal with the exponential-free fragment of the universal logic, that is, the fragment **FD**—would be to replace the axiom of induction by the rule of induction

$$F(0),\ \sqcap x(F(x) \rightarrow F(x+1)) \vdash \sqcap x F(x),$$

which can be shown to preserve the property of computability.

While provability of a formula $\forall x \exists y P(x, y)$ in a sound theory merely signifies that for every n there is m with $P(n, m)$, proving the constructive version $\sqcap x \sqcup y P(x, y)$ of that formula would imply something much stronger: it would mean that, for every n, an m with $P(n, m)$ not only just exists, but can be algorithmically found. This would be guaranteed by the very semantics of that formula rather than some (usually not very

clear) "constructive" syntactic features of the logic that we used in its proof. The only requirement that such a logic would need to satisfy is soundness with respect to our semantics which, as long as all non-logical axioms of the theory are true in our sense (i.e. computable), would automatically take care of everything else. Moreover, in view of clauses (a) of Conjectures 25.4 and 25.8, whose statements most likely extend to the rest of the universal logic as well, after proving $\sqcap x \sqcup y P(x, y)$ we would not only know that an algorithm that computes m from n exists, but we would also be able to actually construct such an algorithm from the proof of $\sqcap x \sqcup y P(x, y)$. Is not this exactly what the constructivists and intuitionists have been calling for?! And, unlike many other attempts (mostly based on some vague philosophical considerations reflected in syntax rather than a coherent and convincing constructivist semantics) to constructivize theories, such as basing arithmetic on **INT** instead of classical logic, yielding loss of information and hence dissatisfaction of those who see nothing wrong with classical logic, our approach could fit everyone's needs and taste: If you are open minded, take advantage of the full expressive power of the universal logic; and if you are constructivistic minded, just identify the collection of operators (or their combinations) whose meanings seem constructive enough to you, then simply mechanically disregard all the theorems of the universal-logic-based applied theory that contain the "bad" operators, and put an end to those fruitless scholastic fights regarding what deductive methods and principles should be considered acceptable and what should not.

The above discussion, as well as all of the examples of Section 26, dealt with systems where new logical operators are applied to old non-logical atoms—predicates. Now we should remember that our logic also has another way to enrich traditional applied systems, which is adding to their vocabularies atoms that represent totally new and higher-level sorts of objects—non-elementary, non-finite-depth or even non-perifinite-depth problems. Only the future can fully reveal the whole potential and utility of this virtue of computability logic.

Index

References

[1] S. Abramsky, R. Jagadeesan, Games and full completeness for multiplicative linear logic, J. Symbolic Logic 59 (2) (1994) 543–574.
[2] S. Artemov, Explicit provability and constructive semantics, Bull. Symbolic Logic 7 (1) (2001) 1–36.
[3] J. van Benthem, Logic in games, ILLC, preprint, University of Amsterdam, 2001.
[4] A. Blass, Degrees of indeterminacy of games, Fund. Math. 77 (1972) 151–166.
[5] A. Blass, A game semantics for linear logic, Ann. Pure Appl. Logic 56 (1992) 183–220.
[6] A. Blass, Questions and answers—a category arising in linear logic, complexity theory, and set theory, in: J.Y. Girard, Y. Lafont, L. Regnier (Eds.), Advances in Linear Logic (Ithaca, NY, 1993), London Mathematical Society Lecture Note Series, Vol. 222, Cambridge University Press, Cambridge, 1995, pp. 61–81.
[7] A.K. Chandra, D.C. Kozen, L.J. Stockmeyer, Alternation, J. Assoc. Comput. Mach. 28 (1) (1981) 114–133.
[8] J.H. Conway, On Numbers and Games, A K Peters Ltd., Natic, MA, 2001.
[9] D. Gale, F.M. Stewart, Infinite games with perfect information, in: H.W. Kuhn, A.W. Tucker (Eds.), Contributions to the Theory of Games, Vol. 2; Ann. Math. Stud. 28 (1953) 245–266.
[10] J.Y. Girard, Linear logic, Theoret. Comput. Sci. 50 (1) (1987) 1–102.
[11] J.M.E. Hyland, C.-H.L. Ong, Fair games and full completeness for multiplicative linear logic without the MIX-rule, preprint, 1993.
[12] G. Japaridze, A constructive game semantics for the language of linear logic, Ann. Pure Appl. Logic 85 (2) (1997) 87–156.
[13] G. Japaridze, The propositional logic of elementary tasks, Notre Dame J. Formal Logic 41 (2) (2000) 171–183.
[14] G. Japaridze, The logic of tasks, Ann. Pure Appl. Logic 117 (2002) 263–295.
[15] G. Japaridze, Preliminary results on the basic propositional logic of racefree games, Bull. Georgian Acad. Sci. 165 (1) (2002) 26–29.
[16] G. Japaridze, Preliminary results on the basic predicate logic of racefree games, Bull. Georgian Acad. Sci. 165 (2) (2002) 256–259.
[17] A. Joyal, Free bicomplete categories, Math. Rep. Acad. Sci. Roy. Soc. Canada 17 (5) (1995) 219–224.
[18] P. Lorenzen, Ein dialogisches Konstruktivitätskriterium, in: Mathematical Institute of the Polish Academy of Sciences (Ed.), Infinitistic Methods, PWN, Proc. Symp. Foundations of Mathematics, Warsaw, 1961, pp. 193–200.
[19] J. McCarty, P. Hayes, Some philosophical problems from the standpoint of Artificial Intelligence, in: B. Meltzer (Ed.), Machine Intelligence, Vol. 4, Edinburgh University Press, Edinburgh, 1969, pp. 463–502.
[20] R. Moore, A formal theory of knowledge and action, in: J.R. Hobbs, R.C. Moore (Eds.), Formal Theories of Commonsense Worlds, Ablex, Norwood, NJ, 1985, pp. 319–358.
[21] S. Russel, P. Norwig, Artificial Intelligence: A Modern Approach, Prentice-Hall, Englewood Cliffs, NJ, 1995.