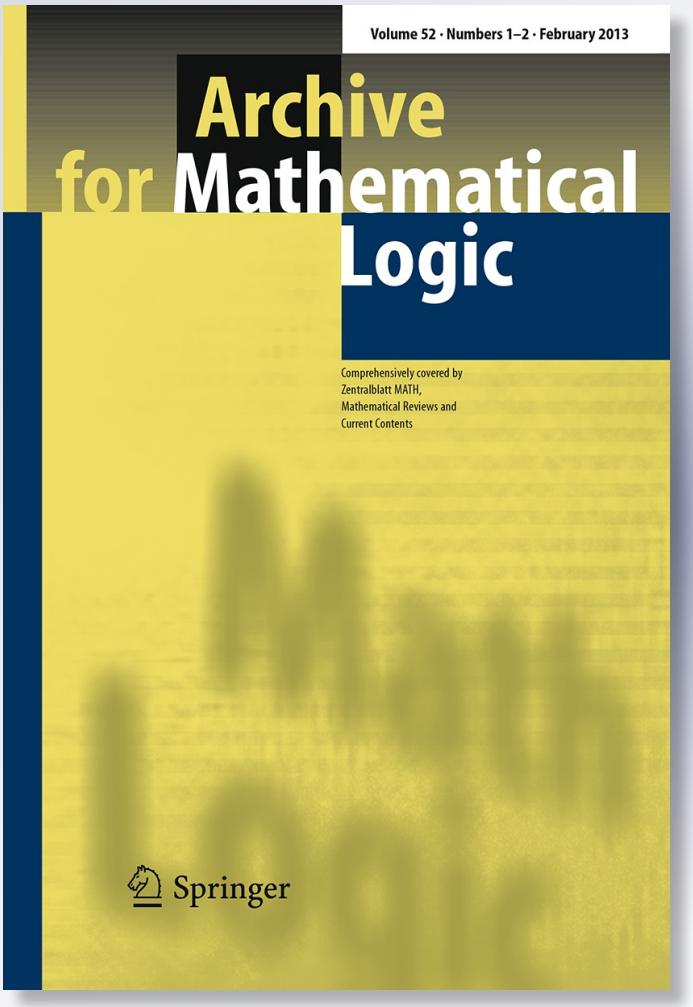
The taming of recurrences in computability logic through cirquent calculus, Part I

Giorgi Japaridze

Archive for Mathematical Logic

ISSN 0933-5846
Volume 52
Combined 1-2

Arch. Math. Logic (2013) 52:173-212
DOI 10.1007/s00153-012-0313-8



Springer

Arch. Math. Logic (2013) 52:173–212
DOI 10.1007/s00153-012-0313-8

Mathematical Logic

The taming of recurrences in computability logic through cirquent calculus, Part I

Giorgi Japaridze

Received: 11 June 2011 / Accepted: 9 November 2012 / Published online: 23 November 2012

Abstract This paper constructs a cirquent calculus system and proves its soundness and completeness with respect to the semantics of computability logic. The logical vocabulary of the system consists of negation $\neg$, parallel conjunction $\wedge$, parallel disjunction $\vee$, branching recurrence $\curlywedge$, and branching corecurrence $\curlyvee$. The article is published in two parts, with (the present) Part I containing preliminaries and a soundness proof, and (the forthcoming) Part II containing a completeness proof.

Keywords Computability logic · Cirquent calculus · Interactive computation · Game semantics · Resource semantics

Mathematics Subject Classification Primary 03B47; Secondary 03B70 · 68Q10 · 68T27 · 68T15

1 Introduction

Computability logic (CoL) is a project for redeveloping logic as a formal theory of computability. In much the same way classical logic's objects of study are predicates and their truth conditions, CoL talks about computational problems and their algorithmic solvability. Computational problems, in turn, understood in the most general—interactive—sense, are defined as games played by a machine against its environment,

Supported by 2010 Summer Research Fellowship from Villanova University.

G. Japaridze
School of Computer Science and Technology, Shandong University, Jinan, China

G. Japaridze (✉)
Department of Computing Sciences, Villanova University, Villanova, PA, USA
e-mail: giorgi.japaridze@villanova.edu

with computability meaning existence of a machine that always wins. Among the main pursuits of CoL is to provide a systematic, universal-utility tool for telling what can be computed and how.

1.1 A brief informal look at the language and semantics of CoL

The approach of CoL induces a rich collection of logical operators, standing for various natural and basic operations on problems/games. An incomplete—in fact, open-ended and still expanding—list of those includes: *negation* $\neg$; *parallel, choice, sequential* and *toggling conjunctions* $\wedge, \sqcap, \triangle, \curlywedge$ together with corresponding *disjunctions* $\vee, \sqcup, \triangledown, \curlyvee$ and *quantifiers* $\bigwedge x, \bigvee x, \bigsqcap x, \bigsqcup x, \triangle x, \triangledown x, \bigcurlywedge x, \bigcurlyvee x$; *branching*, *parallel*, *sequential* and *toggling recurrences* $\mathbin{!}, \mathbin{\curlywedge!}, \mathbin{\triangle!}, \mathbin{\curlyvee!}$ together with their dual *corecurrences* $?, \curlyvee?, \triangledown?, \curlyvee?$.

In a quick intuitive tour of this zoo of operations, $\neg$ can be characterized as a role switch operation: $\neg A$ is the same from the point of a given player as what A is from the point of view of the other player. That is, the machine's moves and wins become those of the environment, and vice versa. For instance, if *Chess* is the game of chess as seen by the white player, then $\neg$*Chess* is the same game as seen by the black player.

Next, $A \wedge B$ and $A \vee B$ are games playing which means playing both A and B in parallel. In $A \wedge B$, the machine is considered to be the winner if it wins in both components, while in $A \vee B$ winning in just one component is sufficient. In contrast, $A \sqcap B$ (resp. $A \sqcup B$) is a game where the environment (resp. machine) has to choose, at the very beginning, one of the two components, after which the play continues according to the rules of the chosen component. To appreciate the difference, compare $\neg Chess \vee Chess$ and $\neg Chess \sqcup Chess$. The former is a two-board game, where the machine plays black on the left board and white on the right board. It is very easily won by the machine by just mimicking on either board the moves made by its adversary on the other board. On the other hand, $\neg Chess \sqcup Chess$ is not at all easy to win. Here the machine has to choose between playing black or white, after which the game continues as the chosen one-board game. Generally, the principle $\neg P \vee P$ is valid in CoL (in the sense of being "always winnable" by a machine) while $\neg P \sqcup P$ is not.

The combination $A \triangle B$ (resp. $A \triangledown B$) is a game that starts as an ordinary play of A. It will also end as A unless, at some point, the environment (resp. machine) decides to make a switch to the second component, in which case the game restarts, continues and ends as B. As for $A \curlywedge B$ (resp. $A \curlyvee B$), here the environment (resp. machine) is allowed to make a switch back and forth between the components any finite number of times.

All of the above four (parallel, choice, sequential and toggling) sorts of conjunction and disjunction naturally extend to corresponding universal and existential quantifiers. Namely, with the universe of discourse being the set of natural numbers, $\bigwedge x A(x)$ can be defined as $A(0) \wedge A(1) \wedge A(2) \wedge \ldots$, $\bigvee x A(x)$ as $A(0) \vee A(1) \vee A(2) \vee \ldots$, $\bigsqcap x A(x)$ as $A(0) \sqcap A(1) \sqcap A(2) \sqcap \ldots$, and so on. To get a feel for the associated computational intuitions, consider a function $f(x)$. CoL sees standard propositions such as $f(3) = 81$ as special, moveless sorts of games, automatically won by the machine when true and lost when false. If so, the meaning of $\bigwedge x \bigvee y(f(x) = y)$ can be seen to be exactly

classical (here with $\wedge = \forall$ and $\vee = \exists$). Namely, this is a moveless game won by the machine if and only if the function $f(x)$ is total. In contrast, $\sqcap x \sqcup y(f(x) = y)$ is a two-move game. The first move is by the environment, consisting in choosing a particular value m for x and intuitively amounting to asking the question "what is the value of $f(m)$?". The second move is by the machine, which should choose a value n for y. This amounts to answering/claiming that $f(m) = n$. The machine wins if and only if such a claim is true. We thus see that $\sqcap x \sqcup y(f(x) = y)$ in fact expresses the problem of computing function $f(x)$. Namely, the machine has a(n algorithmic) winning strategy in this game if and only if $f(x)$ is (total and) computable in the standard sense. In a similar fashion, where $p(x)$ is a predicate, $\sqcap x(\neg p(x) \sqcup p(x))$ can be seen to express the problem of deciding $p(x)$, $\sqcap x(\neg p(x) \triangledown p(x))$ as the problem of semideciding (recursively enumerating) $p(x)$, and $\sqcap x(\neg p(x) \curlyvee p(x))$ as the problem of recursively approximating $p(x)$.

An infinite variety of other relations and operations on computational problems, only very few of which have established names in the literature, can be systematically expressed and studied using the formalism of CoL. This includes various sorts of *reduction* relations or operations, such as mapping (many-to-one) reduction or Turing reduction. Expressions capturing reduction will typically involve the operator $\rightarrow$ (possibly in combination with some other operators), defined by $A \rightarrow B = \neg A \vee B$. To see why the game/problem $A \rightarrow B$ is indeed about reducing B to A, note that, in it, from the machine's prospective, the antecedent A can be viewed as a *computational resource*. Resources are symmetric to problems: what is a computational problem for one player to solve, is a computational resource that the other player can use. Since the roles of the players are interchanged in negated games, A in the antecedent of $A \rightarrow B$ is a resource rather than a problem for the machine. During a play of $A \rightarrow B$, the goal of the machine is to successfully solve (win) B as long as the environment successfully solves (wins) A. The effect is that the environment, in fact, provides an oracle for A, which can be used by the machine in solving B.

What is common to all members of the family of recurrence operations is that, when applied to A, they turn it into a game playing which means repeatedly playing A. In terms of resources, recurrence operations generate multiple "copies" of A, thus making A a reusable/recyclable resource. The difference between the various sorts of recurrences is how "reusage" is exactly understood. To get an intuitive feel for recurrence operations, here we compare three sorts of them: $⍋$, $⅄$ and $⍊$.

Imagine a computer that has a program successfully playing $Chess$. The resource that such a computer provides is obviously something stronger than just $Chess$, for it permits to play $Chess$ as many times as the user wishes, whereas $Chess$, as such, only assumes a single play. Even the simplest operating system would allow to start a session of $Chess$, then—after finishing or abandoning and destroying it—start a new play again, and so on. The game that such a system plays—i.e. the resource that it supports/provides—is nothing but the sequential recurrence $⍋Chess$, which assumes an unbounded number of plays of $Chess$ in a sequential fashion and which can be defined as the infinite sequential conjunction $Chess \vartriangle Chess \vartriangle Chess \vartriangle \ldots$. A more advanced operating system, however, would not require to destroy the old sessions before starting new ones; rather, it would allow to run as many parallel sessions as the user needs. This is what is captured by the parallel recurrence $⅄Chess$, defined as the

infinite parallel conjunction $Chess \wedge Chess \wedge Chess \wedge \ldots$. As a resource, $⅄Chess$ is obviously stronger than $⅄Chess$ as it gives the user greater flexibility. But $⅄$ is still not the strongest form of reusage. A really good operating system would not only allow the user to start new sessions of $Chess$ without destroying old ones; it would also make it possible to branch/replicate each particular stage of each particular session, i.e. create any number of "copies" of any already reached position of the multiple parallel plays of $Chess$, thus giving the user the possibility to try different continuations from the same position. What corresponds to this intuition is the branching recurrence $⅁Chess$.

So, the user of the resource $⅁A$ does not have to restart A from the very beginning every time it wants to reuse it; rather, it is allowed to backtrack to any of the previous—not necessarily starting—positions and try a new continuation from there, thus depriving the adversary of the possibility to reconsider the moves it has already made in that position. This is in fact the type of reusage every purely software resource allows or would allow in the presence of an advanced operating system and unlimited memory: one can start running process A; then fork it at any stage thus creating two threads that have a common past but possibly diverging futures (with the possibility to treat one of the threads as a "backup copy" and preserve it for backtracking purposes); then further fork any of the branches at any time; and so on. The less flexible type of reusage of A assumed by $⅄A$, on the other hand, is closer to what infinitely many autonomous physical resources would naturally offer, such as an unlimited number of independently acting robots each performing task A, or an unlimited number of computers with limited memories, each one only capable of and responsible for running a single thread of process A. Here the effect of replicating/forking an advanced stage of A cannot be achieved unless, by good luck, there are two identical copies of the stage, meaning that the corresponding two robots or computers have so far acted in precisely the same ways. As for $⅄A$, it models the task performed by a single reusable physical resource—the resource that can perform task A over and over again any number of times.

Most interesting and important of all recurrences is branching recurrence $⅁$, on which the present paper is going to be focused. As noted, $⅁$ is the strongest form of recurrence in that it allows to use and re-use its argument (as a computational resource) in the strongest algorithmic sense possible. This immediately translates into a well-justified claim that the compound operation $⅁A \rightarrow B$, abbreviated as $A \circ\!\!- B$, captures our most general intuition of algorithmically reducing B to A. The well-known concept of Turing reduction has the same claims. But the latter is defined only for traditional sorts of problems, such as the problem of computing a function or the problem of deciding a predicate. $A \circ\!\!- B$, on the other hand, is meaningful for all interactive computational problems. As expected, $A \circ\!\!- B$ turns out to be a conservative generalization of Turing reduction in the sense that, when A and B are traditional sorts of problems, B is Turing reducible to A if and only if there is a machine that always wins the game $A \circ\!\!- B$. As for the logical behavior of this generalized Turing reduction operation, the paper [10] showed that the set of the principles validated by $\circ\!\!-$ is precisely described by (the implicative fragment of) Heyting's intuitionistic calculus, with $\circ\!\!-$ understood as intuitionistic implication. This result was further extended in [13] to the principles additionally involving $\sqcap$ and $\sqcup$, with the latter understood as intuitionistic conjunction and disjunction, respectively. This can be viewed as a corroboration of Kolmogorov's

[28] well-known yet rather abstract thesis, according to which intuitionistic logic is a "logic of problems".

All in all, the logical behavior of $⫰$ is reminiscent of Girard's [3] storage operator ! and (especially) Blass's [2] repetition operator R, yet different from either. For instance, as will be seen later from Section 5.6, the principle

$$⫯⫰P \rightarrow ⫰⫯P$$

($⫯$ means $\neg⫰\neg$) is valid in CoL while linear of affine logics do not prove it with $⫰$, $⫯$ understood as !, ? and $\rightarrow$ understood as linear implication; on the other hand, as shown in [23], the principle

$$P \wedge ⫰(P \rightarrow P \wedge P) \wedge ⫰(P \vee P \rightarrow P) \rightarrow ⫰P$$

is not valid in CoL (nor provable in affine logic) while its counterpart is validated by Blass's semantics.

1.2 CoL versus other logical traditions

As noted, CoL sees classical propositions and predicates (the latter being nothing but generalized propositions) as special sorts of games, automatically won or lost depending on whether they are true or false. Such moveless games—problems of zero degree of interactivity—are termed *elementary*. As a result, classical logic re-emerges as a modest conservative fragment of the otherwise much more expressive CoL. Namely, the former is nothing but the latter restricted to elementary games and the vocabulary $\{\neg, \wedge, \vee, \bigwedge, \bigvee\}$. The game-semantical meanings of these five operations turn out to be conservative generalizations of the corresponding classical connectives and quantifiers, naturally and fully coinciding with the latter when applied to propositions and predicates, i.e. elementary games.

A number of non-classical logics and/or their variations also re-emerge as special fragments of CoL. Those include intuitionistic logic (cf. [10,12,13,17,31]), linear logic (cf. [16]) and independence-friendly logic (cf. [20]). CoL with its game semantics thus acts as a unifying framework for various, sometimes seemingly incompatible or even antagonistic philosophical traditions in logic. Accommodating and reconciling this sort of diversity is possible due to the fact that, as [21] puts it, "*CoL gives Caesar what belongs to Caesar and God what belongs to God*". For instance, CoL settles the fruitless controversy around the law of excluded middle between classical and intuitionistic logics by simply pointing out that the meaning associated with disjunction in classical logic is $\vee$ while in intuitionistic logic it is (or should be) $\sqcup$ instead, so that $\neg A \vee A$ is indeed valid just as it is in classical logic, and $\neg A \sqcup A$ is indeed invalid just as it is in intuitionistic or other constructive logics. Next, the differences between classical and linear logics (the latter understood in a generous sense and not necessarily identified with Girard's [3] canonical version of it) are explained by the fact that the two deal with different sorts of "games": classical logic exclusively deals with elementary (moveless) games, while linear logic with not-necessarily-elementary ones.

CoL typically insists on having two different sorts of atoms in its language: $p, q, \ldots$ ranging over elementary games, and $P, Q, \ldots$ ranging over all games.[1] As a result, (for instance) the principle $p \to p \wedge p$ goes through just as it does in classical logic, and the principle $P \to P \wedge P$ fails just as it does in linear logic. As for independence-friendly logic, its expressive power (and far beyond) is achieved through generalizing the syntax of formulas to the more flexible syntax of so called cirquents—a generalization which, as will be seen shortly, is naturally and independently called for in CoL.

Non-classical logics have often been constructed syntactically rather than semantically, essentially by taking an axiomatization of classical logic and deleting or modifying axioms that are otherwise inconsistent with the intuitions and philosophy underlying the non-classical approach. CoL finds this way of developing new logics less than satisfactory, warning that it may result in throwing out the baby with the bath water. Namely, there is no guarantee that, together with the clearly offending principles such as excluded middle in intuitionistic logic or contraction in linear logic, some other, deeply hidden innocent principles will not be automatically expelled as well. The earlier mentioned $⫯⫰P \to ⫰⫯P$ is among such "innocent victims". In CoL, the starting point is semantics rather than syntax, with the function of the latter seen to be acting as a faithful servant to the former rather than vice versa, for it is semantics that provides a bridge between logic and the real, outside word, thus making the former a meaningful and useful tool for navigating the latter. One should explicate the philosophy and intuitions underlying a logic—its *informal semantics*, that is—through an adequate *formal semantics* (rather than try to do so directly through an "adequate syntax"), and only after that start looking for a corresponding syntax/axiomatization, accompanying any adequacy claims for such a syntax with rigorous soundness and completeness proofs. In comparing the semantics-based approach of CoL with the essentially syntax-driven approaches of intuitionistic or linear logics, [16] tries to make a point about the circularity of the latter through the following sarcasm:

> The reason for the failure of $A \sqcup \neg A$ in CoL is not that this principle … is not included in its axioms. Rather, the failure of this principle is exactly the reason why this principle, or anything else entailing it, would not be among the axioms of a sound system for CoL.

1.3 Utility

While at this point the ambitious and long-term CoL project still remains in its infancy, a wide range of applications, mainly in computer science, are already in sight. The applicability of CoL is related to the fact that it provides a systematic answer to not only the question "*What* can be computed?", but also "*How* can be computed?". Namely, all known axiomatizations of (various fragments of) CoL enjoy the so called *uniform-constructive soundness* property, according to which every proof of a valid

[1] This however is not the case for the system **CL15** dealt with in the present paper, whose formal language only has the second sort of atoms.

formula F can be effectively—in fact, efficiently—translated into an algorithmic—in fact, efficient—solution for F (for the problem represented by F, that is) regardless of how the non-logical atoms of F are interpreted. This phenomenon further extends from proofs to derivations: given a derivation of F from some set $\vec{F}$ of formulas, one can effectively—in fact, efficiently—extract a solution S for F from any set $\vec{S}$ of solutions for the elements of $\vec{F}$; furthermore, if all solutions in $\vec{S}$ are efficient, so is S; and, as in the preceding case, such a solution S or its extraction do not depend on the actual meanings associated with the atoms of F, $\vec{F}$. To summarize, CoL is a problem-solving formal tool, allowing us to systematically find solutions for new problems from already known solutions for old problems.

Other than theory of interactive computation and interactive algorithms, the actual or potential application areas for CoL include knowledge base systems [16,33], systems for resource-oriented planning and action [16], logic programming [29,30] and declarative programming languages [21], implicit computational complexity [21,26], constructive applied theories [18,21,26,27]. Discussing those, even briefly, could take us too far. Here we shall only point out that, as expected, in CoL-based applied systems, such as CoL-based axiomatic theories of (Peano) arithmetic developed in [18,21,26,27], every formula represents a(n interactive) computational problem, every theorem represents a problem with an algorithmic solution, and every proof efficiently encodes such a solution. Furthermore, by varying the underlying set of non-logical axioms (usually only induction), one can obtain elegant and amazingly simple systems sound and (representationally) complete with respect to various classes of computational complexity, such as polynomial time computability[2] [21], polynomial space computability [26], elementary recursive computability [26], primitive recursive computability [26], provably recursive computability [27], and so on. Such systems can be viewed as programming languages where programming reduces to proof-search, and where the generally undecidable problem of whether a program meets its specification is fully neutralized because every proof automatically also serves as verification of the correctness of the program extracted from it. In a more ambitious and, at this point, somewhat fantastic perspective, developing reasonable theorem-provers would turn CoL-based applied systems into declarative programming languages in an extreme sense, where human "programming" reduces merely to specifying the goal, with the rest of the job—finding a proof of the goal formula and extracting a program from it—delegated to a CoL-based compiler.

1.4 On the present contribution

Since CoL evolves by the scheme "*from semantics to syntax*", among its main pursuits at this early stage of development is finding sound and complete axiomatizations for various fragments of it. Recent years ([6–15,17–20,24,31,35], etc.) have seen rapid and sustained progress in this direction, at both the propositional and the first-order levels, including axiomatizations for the rather expressive first-order fragments of

[2] Meaning that every proof in such a system encodes not merely an algorithmic solution, but a polynomial time solution, and vice versa: to every polynomial time algorithm corresponds a proof in the system.

CoL on which the above mentioned systems of arithmetic from [18,21,26,27] are based. All fragments axiomatized so far, however, have been recurrence-free,[3] and finding syntactic descriptions of the logic induced by $\barwedge$ (the most important of all recurrence operations) has been remaining among the greatest challenges in the entire CoL enterprise since its inception.

The present paper signifies a long-awaited breakthrough in overcoming that challenge. It constructs a sound and complete axiomatization **CL15** of the basic logic of branching recurrence—namely, the one in the signature $\{\neg, \wedge, \vee, \barwedge, \barwedge\}$. By the standards of CoL, this is a relatively modest fragment, of course. But taming it is a necessary first step, providing a platform for launching attacks on further, incrementally more expressive recurrence-containing fragments. This article is published in two parts, with (the present) Part I containing preliminaries and a soundness proof, and (the forthcoming) Part II [25] containing a completeness proof.

CL15 is a system built in *cirquent calculus*. The latter is a new proof-theoretic approach introduced in [9] and further developed in [14,20,34,35]. It manipulates graph-style constructs termed *cirquents*, as opposed to the traditional tree-style objects such as formulas (Frege, Hilbert), sequents (Gentzen), hypersequents (Avron [1], Pottinger [32]) or structures (Guglielmi [4]). Cirquents come in a variety of forms, and what is characteristic to all of them, making them different from the traditional objects of syntactic manipulation, is allowing to explicitly account for presence or absence of *shared* subcomponents between different components. Among the advantages of cirquent calculus are higher expressiveness, flexibility and efficiency. Due to the first two, cirquent calculus also appears to be the only suitable systematic deductive framework for CoL. Attempts to axiomatize even the simplest $(\neg, \wedge, \vee)$ fragment of CoL in any of the above-mentioned "traditional" frameworks have failed hopelessly, for apparently inherent reasons.

From the technical point of view, the present paper is self-contained in that it includes all relevant definitions. For detailed elaborations on the associated motivations, explanations and illustrations, if necessary, the reader may additionally see the first 10 sections of [16], which provide a tutorial-style introduction to CoL.

2 Basic concepts

The present section provides a quick account on the basic relevant concepts of CoL, and some basic notational conventions that the rest of the paper will rely on. The account is formal/technical and, as mentioned, a reader wishing to get deeper insights, may want to consult [16].

2.1 Constant games

As we already know, CoL is a formal theory of interactive computational problems, and understands the latter as games between two players: *machine* and *environment*.

[3] The so called intuitionistic fragment of CoL, studied in [10,12,13,31], is the only exception. There, however, the usage of $\barwedge$ is limited to the very special form/context $\barwedge E \rightarrow F$.

The symbolic names for these two players are $\top$ and $\bot$, respectively. $\top$ is a deterministic mechanical device (thus) only capable of following algorithmic strategies, whereas there are no restrictions on the behavior of $\bot$. The letter

$$\wp$$

is always a variable ranging over $\{\top, \bot\}$, with

$$\neg\wp$$

meaning $\wp$'s adversary, i.e. the player which is not $\wp$.

We agree that a **move** means any finite string over the standard keyboard alphabet. A **labeled move** (**labmove**) is a move prefixed with $\top$ or $\bot$, with its prefix (**label**) indicating which player has made the move. A **run** is a (finite or infinite) sequence of labmoves, and a **position** is a finite run. Runs will be usually delimited by "$\langle$" and "$\rangle$", with $\langle\rangle$ thus denoting the **empty run**. When Γ is a run, by

$$\neg\Gamma$$

we mean the same run but with each label $\wp$ changed to its opposite $\neg\wp$.

The following is a formal definition of the concept of a constant game, combined with some less formal conventions regarding the usage of certain terminology.

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

1. $\mathbf{Lr}^A$ is a set of runs satisfying the condition that a finite or infinite run is in $\mathbf{Lr}^A$ iff all of its nonempty finite—not necessarily proper—initial segments are in $\mathbf{Lr}^A$ (notice that this implies $\langle\rangle \in \mathbf{Lr}^A$). The elements of $\mathbf{Lr}^A$ are said to be **legal runs** of A, and all other runs are said to be **illegal runs** of A. We say that α is a **legal move** for $\wp$ in a position Φ of A iff $\langle\Phi, \wp\alpha\rangle \in \mathbf{Lr}^A$; otherwise α is an **illegal move**. When the last move of the shortest illegal initial segment of Γ is $\wp$-labeled, we say that Γ is a **$\wp$-illegal run** of A.
2. $\mathbf{Wn}^A$ is a function that sends every run Γ to one of the players $\top$ or $\bot$, satisfying the condition that if Γ is a $\wp$-illegal run of A, then $\mathbf{Wn}^A\langle\Gamma\rangle = \neg\wp$.[4] When $\mathbf{Wn}^A\langle\Gamma\rangle = \wp$, we say that Γ is a **$\wp$-won** (or **won by** $\wp$) run of A; otherwise Γ is **lost by** $\wp$. Thus, an illegal run is always lost by the player who has made the first illegal move in it.

It is clear from the above definition that, when defining the **Wn** component of a particular constant game A, it is sufficient to specify what *legal runs* are won by $\top$. Such a definition will then uniquely extend to all—including illegal—runs. We will implicitly rely on this observation in the sequel.

[4] We write $\mathbf{Wn}^A\langle\Gamma\rangle$ for $\mathbf{Wn}^A(\Gamma)$.

2.2 Game operations

Throughout this paper, a **bitstring** means a finite or infinite sequence of bits 0, 1. For bitstrings x and y, we write

$$x \preceq y$$

to mean that x is a (not necessarily proper) initial segment—i.e. prefix—of y.

Notation 2.2 Let Θ be a run.

1. Where α is a move, we will be using the notation

$$\Theta^{\alpha}$$

to mean the result of deleting from Θ all moves (together with their labels) except those that look like $\alpha\beta$ for some move β, and then further deleting the prefix "α" from such moves. For instance, $\langle \top 0.\beta,\ \bot 1.\gamma,\ \bot 0.\delta\rangle^{0.} = \langle \top\beta,\ \bot\delta\rangle$.

2. Where x is an infinite bitstring, we will be using the notation

$$\Theta^{\preceq x}$$

to mean the result of deleting from Θ all moves (together with their labels) except those that look like $u.\beta$ for some move β and some finite initial segment u of x, and then further deleting the prefix "u." from such moves. For instance, $\langle \top 00.\alpha,\ \bot 001.\beta,\ \bot 0.\delta\rangle^{\preceq 000\cdots} = \langle \top\alpha,\ \bot\delta\rangle$.

Definition 2.3 Below $A,\ A_0,\ A_1$ are arbitrary constant games, α ranges over moves, i ranges over $\{0, 1\}$, w ranges over finite bitstrings, x ranges over infinite bitstrings, Γ ranges over all runs, and Ω ranges over all legal runs of the game that is being defined.

1. $\neg A$ (**negation**) is defined by:
   (i) $\Gamma \in \mathbf{Lr}^{\neg A}$ iff $\neg\Gamma \in \mathbf{Lr}^{A}$.
   (ii) $\mathbf{Wn}^{\neg A}\langle\Omega\rangle = \top$ iff $\mathbf{Wn}^{A}\langle\neg\Omega\rangle = \bot$.

2. $A_0 \wedge A_1$ (**parallel conjunction**) is defined by:
   (i) $\Gamma \in \mathbf{Lr}^{A_0 \wedge A_1}$ iff every move of Γ is $i.\alpha$ for some i, α and, for both i, $\Gamma^{i.} \in \mathbf{Lr}^{A_i}$.
   (ii) $\mathbf{Wn}^{A_0 \wedge A_1}\langle\Omega\rangle = \top$ iff, for both i, $\mathbf{Wn}^{A_i}\langle\Omega^{i.}\rangle = \top$.

3. $A_0 \vee A_1$ (**parallel disjunction**) is defined by:
   (i) $\Gamma \in \mathbf{Lr}^{A_0 \vee A_1}$ iff every move of Γ is $i.\alpha$ for some i, α and, for both i, $\Gamma^{i.} \in \mathbf{Lr}^{A_i}$.
   (ii) $\mathbf{Wn}^{A_0 \vee A_1}\langle\Omega\rangle = \top$ iff, for some i, $\mathbf{Wn}^{A_i}\langle\Omega^{i.}\rangle = \top$.

4. $⍀A$ (**branching recurrence**) is defined by:

   (i) $\Gamma \in \mathbf{Lr}^{⍀A}$ iff every move of Γ is $w.\alpha$ for some w, α and, for all x, $\Gamma^{\preceq x} \in \mathbf{Lr}^{A}$.

   (ii) $\mathbf{Wn}^{⍀A}\langle\Omega\rangle = \top$ iff, for all x, $\mathbf{Wn}^{A}\langle\Omega^{\preceq x}\rangle = \top$.

5. $⍰A$ (**branching corecurrence**) is defined by:

   (i) $\Gamma \in \mathbf{Lr}^{⍰A}$ iff every move of Γ is $w.\alpha$ for some w, α and, for all x, $\Gamma^{\preceq x} \in \mathbf{Lr}^{A}$.

   (ii) $\mathbf{Wn}^{⍰A}\langle\Omega\rangle = \top$ iff, for some x, $\mathbf{Wn}^{A}\langle\Omega^{\preceq x}\rangle = \top$.

Intuitively, as noted in Sect. 1.1, $\neg$ is a role switch operation: it turns $\top$'s (legal) runs and wins into those of $\bot$, and vice versa.

Next, $A \wedge B$ and $A \vee B$ are parallel plays in the two components (two "boards"). The intuitive meaning of a move $0.\alpha$ (resp. $1.\alpha$) by either player is making the move α in the A (resp. B) component of the game. So, when Γ is a legal run of either play, $\Gamma^{0.}$ can and will be seen as the run that took place in A, and $\Gamma^{1.}$ as the run that took place in B. In order to win $A \wedge B$, $\top$ needs to win in both components, while for winning $A \vee B$ winning in just one of the components is sufficient.

Next, $⍀A$ and $⍰A$ can be seen as parallel plays of a continuum of "copies", or "**threads**", of A.[5] Each thread is denoted by an infinite bitstring and vice versa: every infinite bitstring denotes a thread. The meaning of a move $w.\alpha$, where w is a finite bitstring, is making the move α simultaneously in all threads (whose names are) of the form wy. Correspondingly, when Γ is a legal run of $⍀A$ or $⍰A$ and x is an infinite bitstring, $\Gamma^{\preceq x}$ represents the run of A that took place in thread x. In order to win $⍀A$, $\top$ needs to win in all threads, while for winning $⍰A$ winning in just one thread is sufficient.

A correspondence between the above intuitive characterization of $⍀$ and the characterization of this operation provided in Sect. 1.1 may not be obvious. The point is that two versions of $⍀$ have been studied in the earlier literature on CoL. The old, "canonical" version, called *tight*, was defined in [5,16], while the newer version, called *loose*, was introduced only very recently in [22]. It is the definition of the tight rather than the loose version that directly materializes the intuitions presented in Sect. 1.1. On the other hand, Definition 2.3 and the rest of this paper exclusively deal with the loose version. There is nothing to be confused about here: all results of this paper automatically extend to the tight version as well because, as shown in [22], the two versions are equivalent in all relevant respects, including (but not limited to) equivalence in the sense of validating identical principles.

Later we will seldom rely on the strict definitions of the operations $\neg, \wedge, \vee, ⍀, ⍰$ when analyzing games. Rather, based on the above-described intuitions, we will typically use a rather relaxed informal or semiformal language and say, for instance,

[5] Nothing to worry about: "playing a continuum of copies" does not destroy the "finitary" or "playable" character of our games. Every move or position is still a finite object, and every infinite run is still an ω-sequence of (lab)moves.

"$\top$ made the move α in the A component of $A \wedge B$" instead of "$\top$ made the move $0.\alpha$". In either case, instead of saying "$\top$ made the move γ", we can simply say "the labmove $\top\gamma$ was made". And so on.

Note the perfect symmetry between $\wedge$ and $\vee$, as well as between $\barwedge$ and $\barvee$: the definition of either operation of a pair can be obtained from the definition of its dual by simply interchanging $\top$ with $\bot$. With this observation, the following fact is easy to verify:

Fact 2.4 *For any constant games A and B, we have:*

$$\neg\neg A = A;$$
$$\neg(A \wedge B) = \neg A \vee \neg B; \quad \neg(A \vee B) = \neg A \wedge \neg B;$$
$$\neg\barwedge A = \barvee\neg A; \quad \neg\barvee A = \barwedge\neg A.$$

2.3 Games in general

Constant games can be seen as generalized propositions: while the propositions of classical logic are just elements of $\{\top, \bot\}$, constant games are functions from runs to $\{\top, \bot\}$. As we are going to see, our concept of a (simply) game generalizes that of a constant game in the same sense as the classical concept of a predicate generalizes that of a proposition.

We fix a countably infinite set of expressions called **variables**, and another countably infinite set of expressions called **constants**: $\{0, 1, 2, \ldots\}$. Constants are thus decimal numerals, which we shall typically identify with the corresponding natural numbers.

By a **valuation** we mean a mapping e that sends each variable x to a constant $e(x)$. In these terms, a classical predicate p can be understood as a function that sends each valuation e to a proposition, i.e., to a constant predicate. Similarly, what we call a game is a function that sends valuations to constant games:

Definition 2.5 A **game** is a function A from valuations to constant games. We write $e[A]$ (rather than $A(e)$) to denote the constant game returned by A on valuation e. Such a constant game $e[A]$ is said to be an **instance** of A.

Just as this is the case with propositions versus predicates, constant games in the sense of Definition 2.1 will be thought of as special, constant cases of games in the sense of Definition 2.5. In particular, each constant game A' is the game A such that, for every valuation e, $e[A] = A'$. From now on we will no longer distinguish between such A and A', so that, if A is a constant game, it is its own instance, with $A = e[A]$ for every e.

We say that a game A is **unary** iff there is a variable x such that, for any two valuations e_1 and e_2 that agree on x, we have $e_1[A] = e_2[A]$.

Just as the Boolean operations straightforwardly extend from propositions to all predicates, our operations $\neg, \wedge, \vee, \barwedge, \barvee$ extend from constant games to all games. This is done by simply stipulating that $e[\ldots]$ commutes with all of those operations: $\neg A$ is the game such that, for every valuation e, $e[\neg A] = \neg e[A]$; $A \wedge B$ is the game such that, for every valuation e, $e[A \wedge B] = e[A] \wedge e[B]$; etc.

2.4 Static games

While the operations of Sect. 2.2—as well as all other operations studied in CoL—are meaningful for all games, CoL restricts its attention (more specifically, possible interpretations of the atoms of its formal language) to a special yet very wide subclass of games termed "static". Intuitively, static games are interactive tasks where the relative speeds of the players are irrelevant, as it never hurts a player to postpone making moves. In other words, these are games that are contests of intellect rather than contests of speed. Below comes a formal definition of this concept.

For either player $\wp$, we say that a run Υ is a **$\wp$-delay** of a run Γ iff:

- for both players $\wp' \in \{\top, \bot\}$, the subsequence of $\wp'$-labeled moves of Υ is the same as that of Γ, and
- for any $n, k \geq 1$, if the n'th $\wp$-labeled move is made later than (is to the right of) the k'th $\neg\wp$-labeled move in Γ, then so is it in Υ.

The above conditions mean that in Υ each player has made the same sequence of moves as in Γ, only, in Υ, $\wp$ might have been acting with some delay. For instance, of the two runs $\langle \bot\alpha, \top\beta, \bot\delta\rangle$ and $\langle \bot\alpha, \bot\delta, \top\beta\rangle$, the latter is a $\top$-delay of the former while the former is is a $\bot$-delay of the latter.

Let us say that a run is **$\wp$-legal** iff it is not $\wp$-illegal. That is, a $\wp$-legal run is either simply legal, or the player responsible for (first) making it illegal is $\neg\wp$ rather than $\wp$.

Now, we say that a constant game A is **static** iff, whenever a run Υ is a $\wp$-delay of a run Γ, we have:

- if Γ is a $\wp$-legal run of A, then so is Υ;[6]
- if Γ is a $\wp$-won run of A, then so is Υ.

Next, a not-necessarily-constant game is **static** iff so are all of its instances.

It is known [5,22] that the class of static games is closed under the operations $\neg$, $\wedge$, $\vee$, $\barwedge$, $\barvee$, as well as any other operations studied in CoL. Other than being comprehensive (in a sense including "everything that we may ever want to talk about"), this class is very natural and robust from various aspects, one of which is explained later in Remark 2.6. A central thesis on which CoL philosophically relies is that static games are adequate formal counterparts of our broadest intuition of "pure", speed-independent interactive computational problems/tasks.

2.5 Strategies

CoL understands $\top$'s effective strategies as interactive machines. Two versions of such machines were introduced in [5], called *hard-play machine* (**HPM**) and *easy-play machine* (**EPM**). A third kind, called *block-move EPM* (**BMEPM**), was introduced in [18]. All three are sorts of Turing machines with an additional capability

[6] In some papers on CoL, the concept of static games is defined without this (first) condition. In such cases, however, the existence of an always-illegal move ♠ is stipulated in the definition of games. The first condition of our present definition of static games turns out to be simply derivable from that stipulation. This and a couple of other minor technical differences between our present formulations from those given in other pieces of literature on CoL only signify presentational and by no means conceptual variations.

of making moves. Together with the ordinary read/write **work tape**, such machines have two additional tapes, called the **run tape** and the **valuation tape**, both read-only. The run tape serves as a dynamic input, at any time ("**clock cycle**", "**computation step**") spelling the current position, i.e. the sequence of the (lab)moves made by the two players so far: every time one of the players makes a move, that move—with the corresponding label—is automatically appended to the content of this tape. As for the valuation tape, it serves as a static input, spelling some valuation e by listing constants in the lexicographic order of the corresponding variables. Its content remains fixed throughout the work of the machine.

In the HPM model, the machine can make at most one move on a clock cycle but there is no restriction on the frequency of environment's moves, so, during a given cycle, any finite number of environment's moves can be nondeterministically appended to the content of the run tape. In the EPM model, either player can make at most one move on a given clock cycle, but the environment can move only when the machine explicitly allows it to do so. We refer to this sort of an action by the machine as **granting permission**. An BMEPM only differs from an EPM in that either player can make any finite number of moves—rather than only one—at once (the machine whenever it wants, the environment only when permission is granted).

Where $\mathcal{M}$ is an HPM, EPM or BMEPM, a *configuration* of $\mathcal{M}$ is defined in the standard way: this is a full description of the ("current") state of the machine, the contents of its three tapes, and the locations of the corresponding three scanning heads. The *initial configuration* on a valuation e is the configuration where $\mathcal{M}$ is in its start state, the work and run tapes are empty, and the valuation tape spells e. A configuration C' is said to be a *successor* of a configuration C if C' can legally follow C in the standard sense, based on the transition function (which we assume to be deterministic) of the machine and accounting for the possibility of the above-described nondeterministic updates of the content of the run tape. For a valuation e, an **e-computation branch** of $\mathcal{M}$ is a sequence of configurations of $\mathcal{M}$ where the first configuration is the initial configuration on e, and each other configuration is a successor of the previous one. Thus, the set of all computation branches captures all possible scenarios corresponding to different behaviors by $\bot$. Each e-computation branch B of $\mathcal{M}$ incrementally spells—in the obvious sense—a run Γ on the run tape, which we call the **run spelled by** B. We will subsequently refer to any such Γ as a **run generated by** $\mathcal{M}$ on e.

When $\mathcal{M}$ is an EPM or BMEPM and B is a computation branch of $\mathcal{M}$, we say that B is **fair** iff, in it, permission has been granted by $\mathcal{M}$ infinitely many times.

In these terms, an **algorithmic solution** ($\top$**'s winning strategy**) for a given game A is understood as an HPM, EPM or BMEPM $\mathcal{M}$ such that, for every valuation e, whenever B is an e-computation branch of $\mathcal{M}$ and Γ is the run spelled by B, Γ is a $\top$-won run of $e[A]$; if here $\mathcal{M}$ is an EPM or BMEPM, an additional requirement is that B should be fair unless Γ is a $\bot$-illegal run of $e[A]$. When the above is the case, we say that $\mathcal{M}$ **wins**, or **solves**, or **computes** A, and that A is a **computable** game.

Remark 2.6 In the above outline, we described HPMs, EPMs and BMEPMs in a relaxed fashion, without being specific about technical details such as, say, how,

exactly, moves are made by the machine,[7] what happens (in the case of HPM) if both players move during the same cycle,[8] how permission is exactly granted by an EPM or BMEPM,[9] etc. These details are irrelevant and can be filled arbitrarily because, as in the case of ordinary Turing machines, all reasonable design choices yield equivalent (in computing power) models for static games. Furthermore, according to Theorem 17.2 of [5] and Proposition 4.1 of [18], all three models (HPM, EPM and BMEPM) yield the same class of computable static games. And this is so in the following strong, constructive sense: there is an effective procedure for converting any machine $\mathcal{M}$ of any of the three sorts into a machine $\mathcal{M}'$ of any of the other two sorts such that $\mathcal{M}'$ wins every static game that $\mathcal{M}$ wins.

Since we exclusively deal with static games, the three models are thus equivalent in all relevant respects. Therefore, in what follows, we may simply say "a **machine** $\mathcal{M}$" without being specific about whether $\mathcal{M}$ is meant to be an HPM, EPM or BMEPM.

2.6 Formulas and their semantics

We fix a some nonempty collection of (nonlogical) **atoms** and use the letters P, Q as metavariables for them. Throughout this paper, unless otherwise specified, a **formula** means one constructed from atoms in the standard way using the unary connectives $\neg$, $\sqcap$, $\sqcup$ and binary connectives $\wedge$, $\vee$. If we write $F \rightarrow G$, it is to be understood as an abbreviation of $\neg F \vee G$. Furthermore, officially all formulas are required to be written in negation normal form. That is, $\neg$ is only allowed to be applied to atoms. $\neg\neg F$ is to be understood as F, $\neg(F \wedge G)$ as $\neg F \vee \neg G$, $\neg(F \vee G)$ as $\neg F \wedge \neg G$, $\neg\sqcap F$ as $\sqcup\neg F$, and $\neg\sqcup F$ as $\sqcap\neg F$. In view of Fact 2.4, this restriction does not yield any loss of expressive power. As always, a **literal** means P or $\neg P$, where P is an atom.

An **interpretation** is a function * that sends every atom P to a static game P^*. This function extends to all formulas by seeing the logical connectives as the same-name game operations. That is, $(\neg E)^* = \neg(E^*)$, $(E \wedge F)^* = E^* \wedge F^*$, etc. When $F^* = A$, we say that * **interprets** F **as** A.

Definition 2.7 We say that a formula F is:

- **uniformly valid** iff there is a machine $\mathcal{M}$, called a **uniform solution** of F, such that, for every interpretation *, $\mathcal{M}$ wins F^*;
- **multiformly valid** iff, for every interpretation *, there is a machine that wins F^*.

As will be seen later, the two concepts of validity are extensionally equivalent (characterize the same classes of formulas), so we may sometimes simply say "**valid**" without being specific about whether we mean uniform or multiform validity. The main goal of the present paper is to axiomatize the set of valid formulas.

[7] Perhaps this is done by constructing the moves on the work tape, delimiting their beginnings and ends by some special symbols, and then entering one of the specially designated "*move states*".

[8] An arrangement here can be that the machine's move will appear after the environment's move(s).

[9] A natural arrangement would be that permission is granted through entering one of the specially designated "*permission states*".

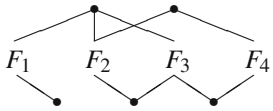
3 Cirquents

Definition 3.1 A **cirquent** (in this paper) is a triple $C = (\vec{F}, \vec{U}, \vec{O})$ where:

1. $\vec{F}$ is a nonempty finite sequence of formulas, whose elements are said to be the **oformulas** of C. Here the prefix "o" is for "occurrence", and is used to mean a formula together with a particular occurrence of it in $\vec{F}$. So, for instance, if $\vec{F} = \langle E, F, E\rangle$, then the cirquent has three oformulas even if only two formulas.
2. Both $\vec{U}$ and $\vec{O}$ are nonempty finite sequences of nonempty sets of oformulas of C. The elements of $\vec{U}$ are said to be the **undergroups** of C, and the elements of $\vec{O}$ are said to be the **overgroups** of C. As in the case of oformulas, it is possible that two undergroups or two overgroups are identical as sets (have identical **contents**), yet they count as different undergroups or overgroups because they occur at different places in the sequence $\vec{U}$ or $\vec{O}$. Simply "**group**" will be used as a common name for undergroups and overgroups.
3. Additionally, every oformula is required to be in (the content of) at least one undergroup and at least one overgroup.

While oformulas are not the same as formulas, we may often identify an oformula with the corresponding formula and, for instance, say "the oformula E" if it is clear from the context which of possibly many occurrences of E is meant. Similarly, we may not always be very careful about differentiating between undergroups (resp. overgroups) and their contents.

We represent cirquents using diagrams such as the one shown below:



This diagram represents the cirquent with four oformulas (in the order of their occurrences) F_1, F_2, F_3, F_4, three undergroups $\{F_1\}$, $\{F_2, F_3\}$, $\{F_3, F_4\}$ and two overgroups $\{F_1, F_2, F_3\}$, $\{F_2, F_4\}$. We typically do not terminologically differentiate between cirquents and diagrams: for us, a diagram *is* (rather than *represents*) a cirquent, and a cirquent *is* a diagram. Each group is represented by (and identified with) a $\bullet$, where the **arcs** (lines connecting the $\bullet$ with oformulas) are pointing to the oformulas that the given group contains.

4 The rules of CL15

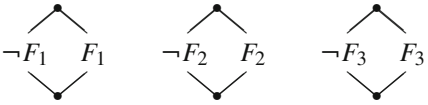
We explain the inference rules of our system **CL15** in a relaxed fashion, in terms of deleting arcs, swapping oformulas, etc. Such explanations are rather clear, and translating them into rigorous formulations in the style and terms of Definition 3.1, while possible, is hardly necessary.

4.1 Axiom

Axiom is a "rule" with no premises. It introduces (its conclusion is) the cirquent

$$(\langle \neg F_1, F_1, \ldots, \neg F_n, F_n\rangle,\ \langle\{\neg F_1, F_1\}, \ldots, \{\neg F_n, F_n\}\rangle,\\ \langle\{\neg F_1, F_1\}, \ldots, \{\neg F_n, F_n\}\rangle),$$

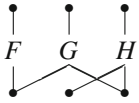
where n is any positive integer, and $F_1, \ldots, F_n$ are any formulas. Such a cirquent looks like an array of n "diamonds", as shown below for the case of $n = 3$:



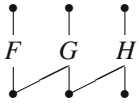
4.2 Exchange

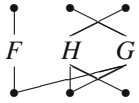
This and all of the remaining rules take a single premise. The Exchange rule comes in three flavors: **Undergroup Exchange**, **Oformula Exchange** and **Overgroup Exchange**. Each one allows us to swap any two adjacent objects (undergroups, oformulas or overgroups) of a cirquent, otherwise preserving all oformulas, groups and arcs.

Below we see three examples. In each case, the upper cirquent is the premise and the lower cirquent is the conclusion of an application of the rule. Between the two cirquents—here and later—is placed the name of the rule by which the conclusion follows from the premise.

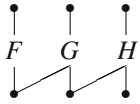


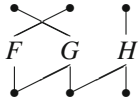
Undergroup Exchange



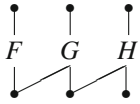


Oformula Exchange





Overgroup Exchange

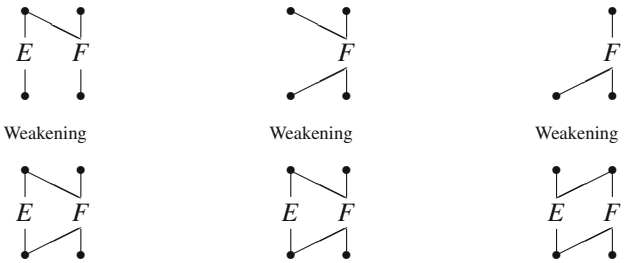


Note that the presence of Exchange essentially allows us to treat all three components $(\vec{F}, \vec{U}, \vec{O})$ of a cirquent as multisets rather than sequences.

4.3 Weakening

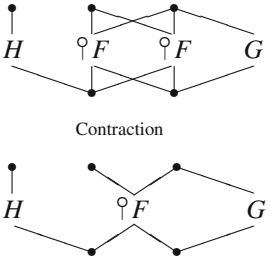
The premise of this rule is obtained from the conclusion by deleting an arc between some undergroup U with ≥ 2 elements and some oformula F; if U was the only undergroup containing F, then F should also be deleted (to satisfy Condition 3 of

Definition 3.1), together with all arcs between F and overgroups; if such a deletion makes some overgroups empty, then they should also be deleted (to satisfy Condition 2 of Definition 3.1). Below are three examples:



4.4 Contraction

The premise of this rule is obtained from the conclusion through replacing an oformula $?F$ by two adjacent oformulas $?F$, $?F$, and including them in exactly the same undergroups and overgroups in which the original oformula was contained. Example:

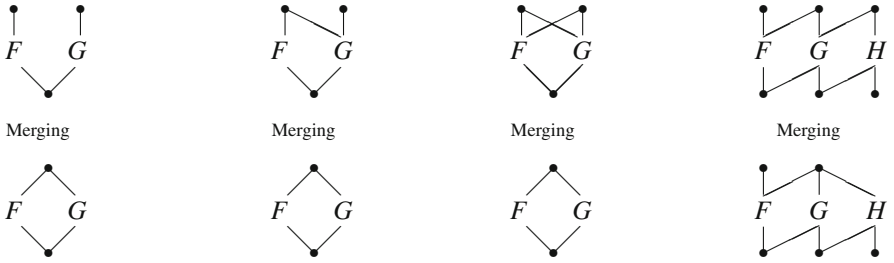


4.5 Duplication

This rule comes in two versions: **Undergroup Duplication** and **Overgroup Duplication**. The conclusion of Undergroup Duplication is the result of replacing, in the premise, some undergroup U with two adjacent undergroups whose contents are identical to that of U. Similarly for Overgroup Duplication. Examples:

4.6 Merging

In the top-down view, this rule merges any two adjacent overgroups, as illustrated below.



4.7 Disjunction introduction

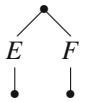
The premise of this rule is obtained from the conclusion by replacing an oformula $F \vee G$ by two adjacent oformulas F, G, and including both of them in exactly the same undergroups and overgroups in which the original oformula was contained, as illustrated below:

4.8 Conjunction introduction

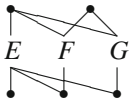
The premise of this rule is obtained from the conclusion by applying the following two steps:

- Replace an oformula $F \wedge G$ by two adjacent oformulas F, G, and include both of them in exactly the same undergroups and overgroups in which the original oformula was contained.
- Replace each undergroup U originally containing the oformula $F \wedge G$ (and now containing F, G instead) by the two adjacent undergroups $U - \{G\}$ and $U - \{F\}$.

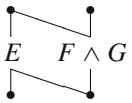
Below we see three examples.

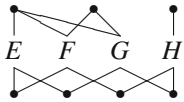


Conjunction Introduction



Conjunction Introduction

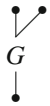




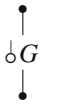
Conjunction Introduction

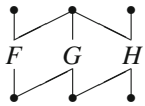
4.9 Recurrence introduction

The premise of this rule is obtained from the conclusion through replacing an oformula $⍜F$ by F (while preserving all arcs), and inserting, anywhere in the cirquent, a new overgroup that contains F as its only oformula. Examples:

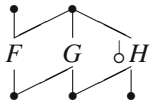


Recurrence Introduction



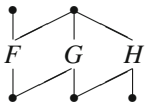


Recurrence Introduction

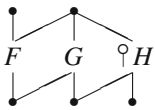


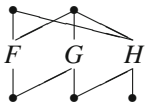
4.10 Corecurrence introduction

The premise of this rule is obtained from the conclusion through replacing an oformula $⍥F$ by F, and including F in any (possibly zero) number of the already existing overgroups in addition to those in which the original oformula $⍥F$ was already present. Examples:

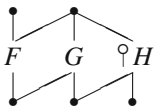


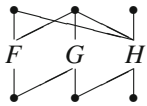
Corecurrence Introduction



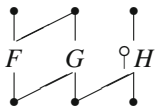


Corecurrence Introduction





Corecurrence Introduction



5 Some taste of CL15

A **CL15-proof** (or simply a **proof**) of a cirquent C is a sequence of cirquents ending in C such that the first cirquent is an axiom, and every subsequent cirquent follows from the immediately preceding cirquent by one of the rules of **CL15**.

For any formula F, we let

$$F^{\clubsuit}$$

denote the cirquent $(\langle F\rangle, \langle\{F\}\rangle, \langle\{F\}\rangle)$, i.e. the cirquent

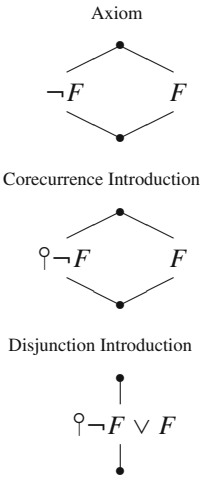
Correspondingly, by a **proof** of a formula F we mean one of the cirquent $F^{\clubsuit}$.

A formula or cirquent X is said to be **provable** (symbolically **CL15** $\vdash X$) if and only if it has a proof. As expected, $\nvdash$ means "not provable".

The following subsections of this section contain proofs of several formulas, serving the purpose of helping the reader to get a better feel for the system.

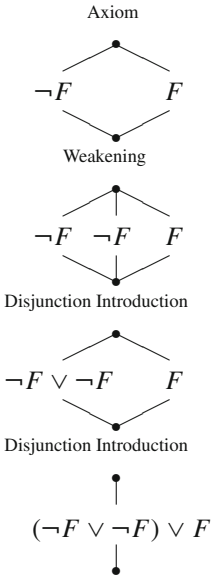
5.1 First example

The following is a proof of $⫰F \rightarrow F$, which, according to our conventions from Sect. 2.6, is an abbreviation of $⫯\neg F \vee F$:



5.2 Second example

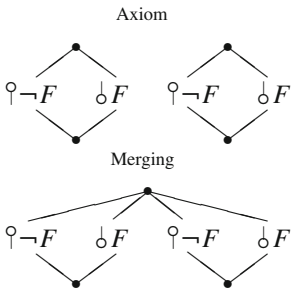
The present example shows a proof of the recurrence-free formula $F \wedge F \to F$:



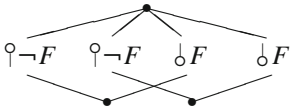
At the same time, it is easy to see that the converse $F \to F \wedge F$ of the above formula has no proof. However, the latter becomes provable with $\mathbin{!}F$ instead of F, as seen from the following example.

5.3 Third example

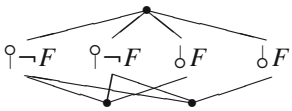
Below is a proof of $\mathbin{!}F \to \mathbin{!}F \wedge \mathbin{!}F$. In informal terms, the meaning of the principle expressed by this formula can be characterized by saying that solving two copies of a problem of the form $\mathbin{!}F$ does not take any more resources ("is not any harder") than solving just a single copy. Note that the same does not hold in the general case, i.e., when F is not necessarily $\mathbin{!}$-prefixed. For instance, $Chess \to Chess \wedge Chess$ cannot be (easily) won.



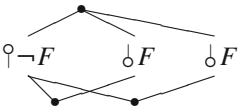
Oformula Exchange



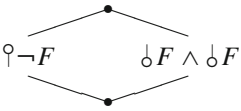
Weakening (twice)



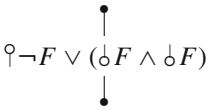
Contraction



Conjunction Introduction



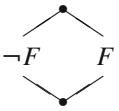
Disjunction Introduction



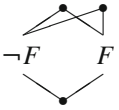
5.4 Fourth example

Below is a proof of $⫰F \rightarrow ⫰⫰F$. Unlike the previously seen examples, proving this formula requires using Duplication:

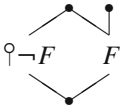
Axiom

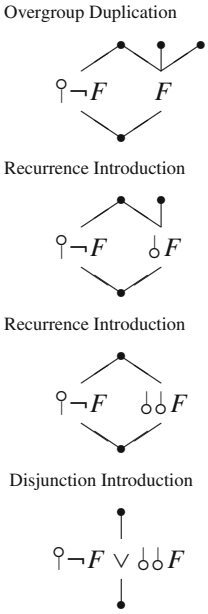


Overgroup Duplication



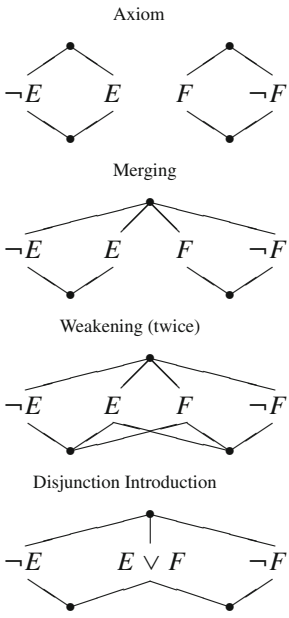
Corecurrence Introduction

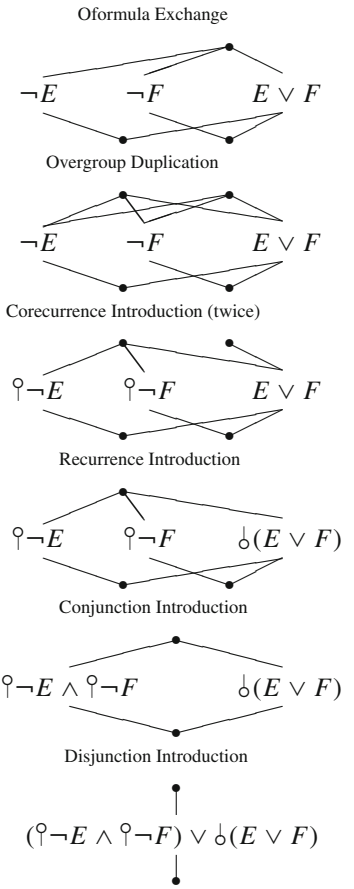




5.5 Fifth example

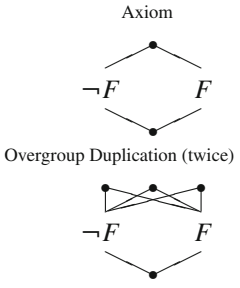
Now we prove $\mathsf{b}E \vee \mathsf{b}F \rightarrow \mathsf{b}(E \vee F)$. The converse $\mathsf{b}(E \vee F) \rightarrow \mathsf{b}E \vee \mathsf{b}F$, on the other hand, can be shown to be unprovable.

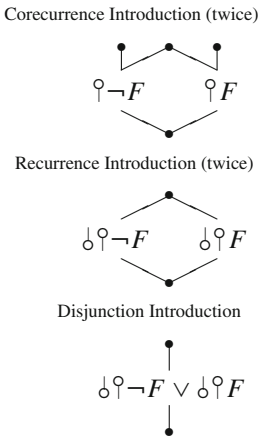




5.6 Sixth example

The formulas proven so far are also provable in affine logic (with $\wedge$, $\vee$ understood as multiplicatives, $⫰$, $⫯$ as exponentials, and $\neg F$ as $F^{\perp}$). The present example shows the **CL15**-provability of the formula $⫯⫰F \rightarrow ⫰⫯F$, which is not provable in affine logic. The converse $⫰⫯F \rightarrow ⫯⫰F$, on the other hand, is unprovable in either system.





Another—longer but recurrence-free—example separating **CL15** from affine logic is

$$(E \wedge F) \vee (G \wedge H) \rightarrow (E \vee G) \wedge (F \vee H)$$

(Blass's [2] principle); constructing a proof of this formula is left as an exercise for the reader.

6 Main theorem

For the terminology used in the following theorem, refer to Sect. 2.6. In addition, by a **constant** (resp. **unary**) **interpretation** we mean one that interprets all atoms as constant (resp. unary) games.

Theorem 6.1 *For any formula F, the following conditions are equivalent:*

(i) **CL15** $\vdash F$;
(ii) F *is uniformly valid;*
(iii) F *is multiformly valid.*

Furthermore:

(a) *The implication* (i) $\Rightarrow$ (ii) *holds in the strong sense that there is an effective procedure which takes any* **CL15***-proof of any formula F and constructs a uniform solution of F.*
(b) *The implication* (ii) $\Rightarrow$ (i) *holds in the strong sense that, if* **CL15** $\nvdash F$*, then, for every HPM $\mathcal{H}$, there is a constant interpretation * such that $\mathcal{H}$ fails to compute F^*.*
(c) *The implication* (iii) $\Rightarrow$ (i) *holds in the strong sense that, if* **CL15** $\nvdash F$*, then there is a unary interpretation † such that $F^\dagger$ is not computable.*

Proof outline: The implication (i) $\Rightarrow$ (ii) (*soundness*), in the form of clause (a), will be proven in Sects. 7 through 10. Uniform validity is stronger than multiform validity,

so the implication $(ii) \Rightarrow (iii)$ is trivial. And the implication $(iii) \Rightarrow (i)$ (*multiform completeness*), in the form of clause (c), as well as clause (b) (*uniform completeness*), will be proven in the forthcoming Part II [25] of the paper.

Of course, **CL15** (i.e., the set of its theorems) is recursively enumerable. At this point, however, we do not have an answer to the following question:

Open Problem 6.2 *Is* **CL15** *decidable?*

As we already know, branching recurrence $\mathfrak{b}$ is the strongest and best motivated, yet not the only, sort of recurrence-style operators studied in CoL. Among the most natural weakenings of $\mathfrak{b}$ are *parallel recurrence* $\mathfrak{p}$ and *countable branching recurrence* $\mathfrak{b}^{\aleph_0}$ (of these two, only $\mathfrak{p}$ was discussed in Sect. 1.1). Here we qualify $\mathfrak{p}$ and $\mathfrak{b}^{\aleph_0}$ as "weakenings" of $\mathfrak{b}$ in the sense that the principles $\mathfrak{b}P \rightarrow \mathfrak{p}P$ and $\mathfrak{b}P \rightarrow \mathfrak{b}^{\aleph_0}P$ are valid (whether it be uniformly and multiformly so) while their converses are not. Semantically, as we probably remember, $\mathfrak{p}A$ is nothing but the infinite conjunction $A \wedge A \wedge A \wedge \ldots$. As for $\mathfrak{b}^{\aleph_0}A$, it is just like $\mathfrak{b}A$, with the only intuitive difference that, while playing $\mathfrak{b}A$ means playing a continuum of copies of A, in $\mathfrak{b}^{\aleph_0}A$ only countably many copies are played—more precisely, it is only countably many copies that eventually matter. This effect can be technically achieved by, say, exclusively limiting our attention to the threads represented by bitstrings that contain only finitely many 1's. While never proven, it is believed [23,31] that $\mathfrak{b}^{\aleph_0}$ is "equivalent" to Blass's [2] *repetition operator* R—at least, in the precise sense that the two operators validate the same logical principles. Strict definitions of $\mathfrak{p}$ and $\mathfrak{b}^{\aleph_0}$ as game operations can be found in [16,17,23], and we will not reproduce them here.

Our system **CL15** becomes incomplete if $\mathfrak{b}$, $\mathfrak{f}$ are understood as (replaced by) either $\mathfrak{p}$, $\mathfrak{y}$ or $\mathfrak{b}^{\aleph_0}$, $\mathfrak{f}^{\aleph_0}$, where $\mathfrak{y}$ and $\mathfrak{f}^{\aleph_0}$ are dual to $\mathfrak{p}$ and $\mathfrak{b}^{\aleph_0}$ in the same sense as $\mathfrak{f}$ is dual to $\mathfrak{b}$. For instance, as shown in [23], the formula

$$P \wedge \mathfrak{b}(P \rightarrow P \wedge P) \wedge \mathfrak{b}(P \vee P \rightarrow P) \rightarrow \mathfrak{b}P,$$

already mentioned in Sect. 1, is not uniformly valid and hence, in view of the soundness of **CL15**, is not provable in the latter. Yet, this formula turns out to be uniformly valid with either operator $\mathfrak{p}$ or $\mathfrak{b}^{\aleph_0}$ instead of $\mathfrak{b}$. The operator $\mathfrak{b}$ turns out to be also logically separated from $\mathfrak{p}$ (but not from $\mathfrak{b}^{\aleph_0}$) by the simpler principle

$$P \wedge \mathfrak{b}(P \rightarrow P \wedge P) \rightarrow \mathfrak{b}P,$$

which is not provable in **CL15** but is uniformly valid when written as $P \wedge \mathfrak{p}(P \rightarrow P \wedge P) \rightarrow \mathfrak{p}P$.

While **CL15** is thus incomplete with respect to $\mathfrak{p}$ or $\mathfrak{b}^{\aleph_0}$, the author has practically no doubts that it however remains sound, meaning that the basic logic induced by $\mathfrak{b}$ (i.e., the set of valid formulas in the signature $(\neg, \wedge, \vee, \mathfrak{b}, \mathfrak{f})$) is a common proper subset of the basic logics induced by $\mathfrak{p}$ and $\mathfrak{b}^{\aleph_0}$:

Conjecture 6.3 *The soundness part of Theorem 6.1, in the strong form of clause (a), continues to hold with* $\mathbin{⅋}$ *and* $\mathbin{⅄}$ *understood as* $\curlywedge$ *and* $\curlyvee$*, respectively.*

Conjecture 6.4 *The soundness part of Theorem 6.1, in the strong form of clause (a), continues to hold with* $\mathbin{⅋}$ *and* $\mathbin{⅄}$ *understood as* $\mathbin{⅋}^{\aleph_0}$ *and* $\mathbin{⅄}^{\aleph_0}$*, respectively.*

At the same time, the author does not have any guess regarding whether one should expect the answers to the following questions to be positive or negative:

Open Problem 6.5 *Is the set of (uniformly or multiformly) valid formulas in the logical signature* $(\neg, \wedge, \vee, \curlywedge, \curlyvee)$ *decidable or, at least, recursively enumerable?*

Open Problem 6.6 *Is the set of (uniformly or multiformly) valid formulas in the logical signature* $(\neg, \wedge, \vee, \mathbin{⅋}^{\aleph_0}, \mathbin{⅄}^{\aleph_0})$ *decidable or, at least, recursively enumerable?*

If the answer in either case is positive, it would be very interesting to find a syntactically reasonable axiomatization. An expectation here is that, if found, such an axiomatization would be more complex than **CL15**.

7 Preliminaries for the soundness proof

The remaining sections of this article are devoted to a proof of the following lemma:

Lemma 7.1 *There is an effective procedure which takes any* **CL15***-proof of any formula* F *and constructs a machine* $\mathcal{M}$ *such that, for any constant interpretation* **,* $\mathcal{M}$ *wins* F^**.*

Clause (a) of Theorem 6.1 is an immediate corollary of the above lemma. To see this, consider an arbitrary **CL15**-proof of an arbitrary formula F. Let $\mathcal{M}$ be the corresponding machine returned by the procedure whose existence is claimed in Lemma 7.1. Now we claim that $\mathcal{M}$ is a uniform solution of F, and hence clause (a) of Theorem 6.1 holds. Indeed, consider an arbitrary (not necessarily constant) interpretation *. How do we know that $\mathcal{M}$ wins F^*? Let, for every valuation e, ^{*e} be the interpretation that interprets each atom P as the game $e[F^*]$. Note that such a ^{*e} is a constant interpretation. In view of Remark 2.6, we may assume $\mathcal{M}$ is an HPM. By definition, $\mathcal{M}$ wins F^* iff, for every valuation e and every run Γ generated by $\mathcal{M}$ on e, Γ is a $\top$-won run of $e[F^*]$. Consider an arbitrary valuation e and an arbitrary run Γ generated by $\mathcal{M}$ on e. A straightforward induction on the complexity of F shows that the game $e[F^*]$ is the same as F^{*e}. The latter, in turn, as a constant game, is the same as $e[F^{*e}]$. Thus, $e[F^*] = e[F^{*e}]$. Lemma 7.1 promises that $\mathcal{M}$ wins F^{*e}. This, in turn, implies that Γ is a $\top$-won run of $e[F^{*e}]$, and hence a $\top$-won run of $e[F^*]$. Since e and Γ were arbitrary, we find that $\mathcal{M}$ wins F^*, as desired.

An advantage of proving clause (a) of Theorem 6.1 through proving Lemma 7.1 is that this allows us to exclusively limit our attention to constant games. Winning strategies/machines for such games can fully ignore the valuation tape, as its content is irrelevant. With this remark in mind, throughout the present part of the paper, with a couple of exceptions, there will be no mention of valuation or the valuation tape in our descriptions of such strategies.

8 The semantics of cirquents

Lemma 7.1 will be proven by induction on the lengths of **CL15**-proofs. To make such an induction possible, we first need to extend our semantics from formulas to cirquents. In rough intuitive terms, such a semantics treats overgroups as generalized ⫰s, with the main difference between the ordinary ⫰ and an overgroup being that the latter can be shared by several arguments (oformulas). Next, undergroups are like disjunctions (or, rather, disjunctions prefixed with generalized ⫰s), with the main difference between ordinary disjunctions and undergroups being that the latter may have shared arguments with other undergroups. As noted earlier, sharing is the main feature distinguishing cirquents from the other, traditional syntactic objects studied in logic, such as formulas or sequents. Finally, the whole cirquent is like a conjunction of its undergroups.

To define our semantics formally, we need the following notational convention. Let Ω be a run, a be (the decimal numeral for) a positive integer, and $\vec{x} = x_1, \ldots, x_n$ be a nonempty sequence of n infinite bitstrings. We shall write

$$\Omega^{\preceq a;\vec{x}}$$

to mean the result of deleting from Ω all moves (together with their labels) except those that look like $a; u_1, \ldots, u_n.\beta$ for some move β and some finite initial segments $u_1, \ldots, u_n$ of $x_1, \ldots, x_n$, respectively, and then further deleting the prefix "$a; u_1, \ldots, u_n.$" from such moves. For instance, if $x = 000\ldots$ and $y = 111\ldots$, then

$$\langle \top 3; 00, 1.\alpha,\ \bot 3; 001, 11.\beta,\ \bot 5; 00, 1.\delta,\ \top 3; 0, 111.\gamma\rangle^{\preceq 3;x,y} = \langle \top\alpha,\ \top\gamma\rangle.$$

See Remark 8.2 below for an explanation of the intuitions associated with the $\Omega^{\preceq a;\vec{x}}$ notation.

Throughout this paper, the letter

$$\epsilon$$

is used to denote the **empty bitstring**. The latter is a prefix (initial segment) of every bitstring.

Definition 8.1 Consider a constant interpretation * (in the old, ordinary sense) and a cirquent

$$C = (\langle F_1, \ldots, F_k\rangle, \langle U_1, \ldots, U_m\rangle, \langle O_1, \ldots, O_n\rangle)$$

with k oformulas, m undergroups and n overgroups. Then C^* is the constant game defined as follows, with Γ ranging over all runs and Ω ranging over the legal runs of C^*:

(i) $\Gamma \in \mathbf{Lr}^{C^*}$ iff the following two conditions are satisfied:

- Every move of Γ looks like $a; \vec{u}.\alpha$, where α is some move, $a \in \{1, \ldots, k\}$, and $\vec{u} = u_1, \ldots, u_n$ is a sequence of n finite bitstrings such that the following condition is satisfied:

$$\begin{array}{l} \textit{whenever an overgroup } O_j\ (1 \leq j \leq n) \textit{ does not contain} \\ \textit{the oformula } F_a,\ u_j = \epsilon. \end{array} \tag{1}$$

- For every $a \in \{1, \ldots, k\}$ and every sequence $\vec{x}$ of n infinite bitstrings, $\Gamma^{\preceq a;\vec{x}} \in \mathbf{Lr}^{F_a^*}$.

(ii) $\mathbf{Wn}^{C^*}\langle\Omega\rangle = \top$ iff, for every $i \in \{1, \ldots, m\}$ and every sequence $\vec{x}$ of n infinite bitstrings, there is an $a \in \{1, \ldots, k\}$ such that the undergroup U_i contains the oformula F_a and $\mathbf{Wn}^{F_a^*}\langle\Omega^{\preceq a;\vec{x}}\rangle = \top$.

Remark 8.2 Intuitively, when C and * are as above, a (legal) play/run Ω of C^* consists of parallel plays of a continuum of threads of each of the games F_a^* $(1 \leq a \leq k)$. Namely, every thread of such an F_a^* is $\Omega^{\preceq a;\vec{x}}$ for some array $\vec{x} = x_1, \ldots, x_n$ of n infinite bitstrings. In the context of a fixed Ω, we may refer to $\Omega^{\preceq a;\vec{x}}$ as **the thread** $\vec{x}$ **of** F_a^*. Next, for an undergroup U_i, let us say that $\top$ is the **winner in** U_i iff, for every array $\vec{x}$ of n infinite bitstrings, there is an oformula F_a in U_i such that the thread $\vec{x}$ of F_a^* is won by $\top$. Now, $\top$ wins the overall game C^* iff it wins in all undergroups of C.

As for the condition (1) of the definition, it can be seen as saying that, for any array $\vec{x} = x_1, \ldots, x_n$ of infinite bitstrings, only some of the elements of $\vec{x}$ are really *relevant* to any given oformula F_a of the cirquent. In particular, an element x_j of $\vec{x}$ is relevant if the overgroup O_j contains F_a. This relevance/irrelevance is in the precise sense that, if an array $\vec{y}$ only differs from $\vec{x}$ in "irrelevant" elements, then, as it is easy to see from condition (1) and the fact that ϵ is a prefix of every bitstring, we have $\Omega^{\preceq a;\vec{x}} = \Omega^{\preceq a;\vec{y}}$.

Definition 8.3 We say that a cirquent C is **uniformly valid** iff there is a machine $\mathcal{M}$, called a **uniform solution** of C, such that, for every constant interpretation *, $\mathcal{M}$ wins C^*.

9 The generalized soundness of CL15

Lemma 9.1 *There is an effective function f from machines to machines such that, for every machine $\mathcal{M}$, formula F and interpretation *, if $\mathcal{M}$ wins $\mathbin{⫰}F^*$, then $f(\mathcal{M})$ wins F^*.*

Proof Theorem 37 of [16] establishes the soundness of affine logic with respect to uniform validity. But affine logic proves $\mathbin{⫰}P \rightarrow P$. So, this formula is uniformly valid, meaning that there is a machine—let us denote it by $\mathcal{N}_0$—that wins $\mathbin{⫰}F^* \rightarrow F^*$ for any formula F and interpretation *. Next, Proposition 21.3 of [5] establishes that computability of static games is closed under modus ponens in the strong sense that any pair $(\mathcal{N}, \mathcal{M})$ of machines can be effectively converted into a machine $h(\mathcal{N}, \mathcal{M})$ such that, for any static games A and B, if $\mathcal{N}$ wins $A \rightarrow B$ and $\mathcal{M}$ wins A, then $h(\mathcal{N}, \mathcal{M})$ wins B. Now it is clear that the function $f(\mathcal{M})$ defined by $f(\mathcal{M}) = h(\mathcal{N}_0, \mathcal{M})$ satisfies the promise of our present lemma. □

Lemma 9.2 *There is an effective function g from machines to machines such that, for every machine $\mathcal{M}$, formula F and constant interpretation *, if $\mathcal{M}$ wins $(F^{\clubsuit})^*$, then $g(\mathcal{M})$ wins F^*.*

Proof In view of Lemma 9.1, it is sufficient to prove our present lemma for $F^{\clubsuit}$ versus $\mathbin{⫰}F$ instead of $F^{\clubsuit}$ versus F. Consider an arbitrary EPM $\mathcal{M}$ and an arbitrary interpretation * (on which the function g is not going to depend). The idea of our proof is very simple and can be summarized by saying that the games $(\mathbin{⫰}F)^*$ and $(F^{\clubsuit})^*$ are essentially the same, with only a minor technical difference in the forms of their legal moves. Specifically, while every legal move of $(F^{\clubsuit})^*$ looks like $1;w.\alpha$ for some finite bitstring w and move α, the corresponding move of $(\mathbin{⫰}F)^*$ simply looks like $w.\alpha$ instead, and vice versa. So, if $\mathcal{M}$ wins $(F^{\clubsuit})^*$, then an "essentially the same" strategy $g(\mathcal{M})$ wins $(\mathbin{⫰}F)^*$.

In more detail, we construct $g(\mathcal{M})$ as an EPM that plays $(\mathbin{⫰}F)^*$ through simulating and mimicking—with certain minor readjustments—a play of $(F^{\clubsuit})^*$ by $\mathcal{M}$ (call the latter the **imaginary play**).[10] Namely, $g(\mathcal{M})$ grants permission whenever it sees that the simulated $\mathcal{M}$ does so[11] and, if the environment responds by a move $w.\alpha$ for some finite bitstring w and move α,[12] it translates it as the move $1;w.\alpha$ made by the imaginary adversary of $\mathcal{M}$. And "vice versa": whenever the simulated $\mathcal{M}$ makes a move $1;w.\alpha$ in the imaginary play of $(F^{\clubsuit})^*$, $g(\mathcal{M})$ translates it as the move $w.\alpha$ in the play of $(\mathbin{⫰}F)^*$—makes the move $w.\alpha$ in the real play, that is. What $g(\mathcal{M})$ achieves by playing this way is that it "**synchronizes**" each thread x of F^* in the real play of $(\mathbin{⫰}F)^*$ with the same thread x of F^* in the imaginary play of $(F^{\clubsuit})^*$.

Consider any run Γ generated by $g(\mathcal{M})$. Let Ω be the corresponding run in the imaginary play of $(F^{\clubsuit})^*$ by $\mathcal{M}$, i.e., the run of $(F^{\clubsuit})^*$ emerged during the simulation in the scenario which made $g(\mathcal{M})$ generate Γ. It is rather obvious that $g(\mathcal{M})$ never makes illegal moves unless its environment or the simulated $\mathcal{M}$ does so first. Hence we may safely assume that Γ is a legal run of $(\mathbin{⫰}F)^*$ and Ω is a legal run of $(F^{\clubsuit})^*$, for otherwise either Γ is a $\bot$-illegal run of $(\mathbin{⫰}F)^*$ and thus $g(\mathcal{M})$ is an automatic winner in $(\mathbin{⫰}F)^*$, or Ω is a $\top$-illegal run of $(F^{\clubsuit})^*$ and thus $\mathcal{M}$ does not win $(F^{\clubsuit})^*$.[13] Now observe that, for any infinite bitstring x, $\Gamma^{\preceq x} = \Omega^{\preceq 1;x}$. It is therefore obvious that, as long as Ω is a $\top$-won run of $(F^{\clubsuit})^*$, Γ is a $\top$-won run of $(\mathbin{⫰}F)^*$. In other words, if $\mathcal{M}$ wins $(F^{\clubsuit})^*$, then $g(\mathcal{M})$ wins $(\mathbin{⫰}F)^*$. Needless to point out that our construction (the function g) is effective, as promised in the lemma. □

We say that a rule of **CL15** other than Axiom is **uniform-constructively sound** iff there is an effective procedure that takes any instance (A, B) (a particular premise-

[10] While the contents of valuation tapes are irrelevant as we deal with constant games, for clarity let us say that $\mathcal{M}$ is simulated in the scenario where the valuation spelled on its valuation tape sends every variable to 0.

[11] Later, in similar descriptions, we shall no longer explicitly mention this obvious detail common to all simulations.

[12] If the environment responds by a move that does not look like $w.\alpha$, such a move is illegal and $g(\mathcal{M})$ can retire with a spectacular victory; and if the environment does not respond at all, $g(\mathcal{M})$ feeds "no response" back to the simulation.

[13] Later, in similar arguments, the assumption of Γ and Ω being legal will usually be made only implicitly, leaving a routine observation of the legitimacy of such an assumption to the reader.

conclusion pair, that is) of the rule, any machine $\mathcal{M}_A$ and returns a machine $\mathcal{M}_B$ such that, for any constant interpretation *, whenever $\mathcal{M}_A$ wins A^*, $\mathcal{M}_B$ wins B^*. Then, of course, as long as $\mathcal{M}_A$ is a uniform solution of A, $\mathcal{M}_B$ is a uniform solution of B. As for Axiom, by its uniform-constructive soundness we simply mean existence of an effective procedure that takes any instance B of (the "conclusion" of) Axiom and returns a uniform solution $\mathcal{M}_B$ of B.

Theorem 9.3 *All rules (including Axiom) of* **CL15** *are uniform-constructively sound.*

Proof Given in Sect. 10. □

Theorem 9.4 *Every cirquent provable in* **CL15** *is uniformly valid.*

Furthermore, there is an effective procedure that takes an arbitrary **CL15***-proof of an arbitrary cirquent C and constructs a uniform solution of C.*

Proof Immediately from Theorem 9.3 by induction on the lengths of **CL15**-proofs. □

Now, Lemma 7.1, proving which was our goal, is an immediate corollary of Theorem 9.4 and Lemma 9.2. Our only remaining duty is to prove Theorem 9.3. This job is done in the following section.

10 The uniform-constructive soundness of the rules of CL15

Below, one by one, we prove the uniform-constructive soundness of all rules of **CL15**. In each case, A stands for the premise of an arbitrary instance of the rule and B for the corresponding conclusion (except the case of Axiom, where we only have B). Next, $\mathcal{M}_A$ always stands for an arbitrary machine, and $\mathcal{M}_B$ for the machine constructed from $\mathcal{M}_A$ and (subsequently) shown to win B^* as long as $\mathcal{M}_A$ wins A^*, for whatever constant interpretation *. It will usually be immediately clear from our description of $\mathcal{M}_B$ that it can be constructed effectively (so that the soundness of the rule is "constructive"), and that its work in no way depends on an interpretation * applied to the cirquents involved (so that the soundness of the rule is "uniform"). Since an interpretation * is never relevant in such proofs, we can take the liberty to omit it and write simply X where, strictly speaking, X^* is meant. That is, we will—both notationally and terminologically—identify formulas or cirquents with the games into which they turn once an interpretation is applied to them.

Also, since we only deal with constant games, the (content of the) valuation tape is never relevant, and we may safely pretend that such a tape simply does not exist. Technically, this effect can be achieved by assuming that the valuation tape of any—real or simulated—machine always spells the same valuation, say, the one that sends every variable to 0.

In each non-axiom case, it will be implicitly assumed that $\mathcal{M}_A$ wins A. It is important to note that our *construction* of the corresponding $\mathcal{M}_B$ will never depend on this assumption; only the subsequent *conclusion* that $\mathcal{M}_B$ wins B will depend on it. Also, $\mathcal{M}_B$ will always be implicitly assumed to be an EPM, and so will be $\mathcal{M}_A$ unless otherwise specified.

10.1 Axiom

Assume that B is an axiom, namely, that it is

The EPM $\mathcal{M}_B$ that wins B works as follows. It keeps granting permission. Every time the adversary makes a move $a; \vec{w}.\alpha$, where $1 \leq a \leq 2n$ and $\vec{w}$ is an array of n finite bitstrings (note that every legal move of B should indeed look like this), $\mathcal{M}_B$ responds by the move $b; \vec{w}.\alpha$, where b is $a+1$ if a is odd, and $a-1$ if a is even.

Notice that what such an $\mathcal{M}_B$ does is applying copycat between the two oformulas/games of each thread of each diamond. Namely, when a, b are as above, Γ is any run generated by $\mathcal{M}_B$ and $\vec{x}$ is any array of n infinite bitstrings, we have $\Gamma^{\preceq a;\vec{x}} = \neg\Gamma^{\preceq b;\vec{x}}$. It is therefore obvious that Γ is a $\top$-won run of B, meaning that $\mathcal{M}_B$ wins B.

10.2 Exchange

Undergroup Exchange does not affect anything relevant: as a game, the conclusion is the same as the premise.

Assume now B follows from A by Oformula Exchange. Namely, oformulas $\#a$ and $\#b$ ($b = a+1$) of A have been swapped when obtaining B from A. We construct $\mathcal{M}_B$ as a machine that works by simulating and mimicking $\mathcal{M}_A$ in the style that we saw in the proof of Theorem 9.2. Note that A and B, as games, are "essentially the same". Hence, all that $\mathcal{M}_B$ needs to do to account for the minor technical difference between A and B is to make a very simple "translation" or "reinterpretation" of moves. Namely, any move α made within a given thread of the F_a (resp. F_b) component of the real play of B $\mathcal{M}_B$ sees exactly as $\mathcal{M}_A$ would see the same move α in the same thread of F_b (resp. F_a), and vice versa. In more precise terms, with $\vec{w}$ ranging over sequences of as many finite bitstrings as the number of overgroups in either cirquent, every move (by either player) $a; \vec{w}.\alpha$ (resp. $b; \vec{w}.\alpha$) of the real play is understood as the move $b; \vec{w}.\alpha$ (resp. $a; \vec{w}.\alpha$) made by the same player in the imaginary play, and vice versa. All other moves are understood exactly as they are, without any reinterpretation.

With a moment's thought, it can be seen that $\mathcal{M}_B$ wins B because $\mathcal{M}_A$ wins A.

A similar idea applies to the case of Overgroup Exchange. The only difference is that here, instead of reinterpreting the occurrence of either oformula as the occurrence of the oformula with which it was swapped, $\mathcal{M}_B$ reinterprets the occurrence of either overgroup as the occurrence of the overgroup with which it was swapped.

10.3 Weakening

Assume B is obtained from A by Weakening. Turning $\mathcal{M}_A$ into $\mathcal{M}_B$ is very easy. If, when moving from B to A, no oformula of B was deleted, then the old $\mathcal{M}_A$ obviously

wins not only A but B as well, because every $\top$-won run of A is automatically also a $\top$-won run of B. Now suppose an oformula F_a of B was deleted. In view of the presence of Exchange in the system, we may assume that F_a is the last oformula of B. In this case $\mathcal{M}_B$ is a machine that plays by simulating and mimicking $\mathcal{M}_A$. In its simulation/play routine, $\mathcal{M}_B$ ignores the moves within F_a, and otherwise (in all other oformulas) plays exactly as $\mathcal{M}_A$ does, except that moves need to be slightly readjusted if the deletion of F_a also resulted in the deletion of some overgroups of B. Namely, $\mathcal{M}_B$ interprets every move $b; \vec{u}.\alpha$ made in B as the move $b; \vec{u}'.\alpha$ made in A and vice versa, where $\vec{u}'$ is the result of removing from $\vec{u}$ the bitstrings (all empty, by the way) corresponding to the deleted overgroups.

10.4 Contraction

In this and the remaining subsections of the present section, as was done in the preceding subsection, in view of the presence of Exchange in the system and the already verified fact of its uniform-constructive soundness, we can and will always assume that the objects—namely, oformulas or overgroups—affected by the rule are at the end of the corresponding lists of objects of the corresponding cirquents.

Assume B is obtained from A by Contraction, with $⚲F$ being the contracted oformula, located at the end of the list of oformulas of B. Let a be the number of oformulas of B, and let $b = a + 1$. Thus, the a'th oformula of B is $⚲F$, and so are the a'th and b'th oformulas of A. Next, let n be the number of overgroups in either cirquent. In what follows, we let $\vec{w}$ range over sequences of n finite bitstrings. Also, in the present case we assume that $\mathcal{M}_A$ is a BMEPM rather than an EPM. In view of Remark 2.6, such an assumption is perfectly legitimate.

As usual, we define $\mathcal{M}_B$ as an EPM that works by simulating $\mathcal{M}_A$ and mimicking it after reinterpreting moves. Nothing is to be reinterpreted in the case of moves that take place within the oformulas other than $⚲F$. As for the $⚲F$ parts, we have:

- $\mathcal{M}_B$ translates every move $a; \vec{w}.0u.\alpha$ (by either player) in the real play of B as the move $a; \vec{w}.u.\alpha$ (by the same player) of the imaginary play of A, and vice versa.
- $\mathcal{M}_B$ translates every move $a; \vec{w}.1u.\alpha$ (by either player) in the real play of B as the move $b; \vec{w}.u.\alpha$ (by the same player) of the imaginary play of A, and vice versa.
- If the (real) environment ever makes a move $a; \vec{w}.\epsilon.\alpha$ in the play of B, $\mathcal{M}_B$ translates it as a block of the two moves $a; \vec{w}.\epsilon.\alpha$ and $b; \vec{w}.\epsilon.\alpha$ by the imaginary adversary of $\mathcal{M}_A$ in the play of A.

Since $\mathcal{M}_A$ is a BMEPM, it may occasionally make a block of several moves at once. In this case $\mathcal{M}_B$ still acts as described above, with the only difference that it will correspondingly make several consecutive moves in the real play, rather than only one move.

The effect achieved by $\mathcal{M}_B$'s strategy can be summarized by saying that it synchronizes every thread y of F of every thread $\vec{w}$ of the first (resp. second) copy of $⚲F$

in A with the thread $0y$ (resp. $1y$) of F of the thread $\vec{w}$ of the (single) copy of $⍰F$ in B.[14]

Consider any run Γ of B generated by $\mathcal{M}_B$. Let Ω be the corresponding run emerged in the imaginary play of A by $\mathcal{M}_A$. Since $\mathcal{M}_A$ wins A, Ω is a $\top$-won run of A. Next, let us fix some array $\vec{x}$ of n infinite bitstrings. Let us agree that, in what follows, when we talk about playing, winning, etc. in A (resp. B) or any of its components, it is to be understood in the context of the array/thread $\vec{x}$ and the play/run Ω (resp. Γ) or the corresponding subruns of it. Our goal is to see that $\mathcal{M}_B$ is the winner in B. This, in turn, means showing that $\mathcal{M}_B$ is the winner in every undergroup of B (see Remark 8.2).

Indeed, consider any (i'th) undergroup U_i^B of B. Since $\mathcal{M}_A$ wins A, the corresponding (i'th) undergroup U_i^A of A is won by $\mathcal{M}_A$. This, in turn, means that there is an $\mathcal{M}_A$-won oformula E in U_i^A.

If E is not one of the two copies of $⍰F$, then the oformula E of B is also won by $\mathcal{M}_B$, because $\mathcal{M}_B$ plays in E exactly as $\mathcal{M}_A$ does. Hence U_i^B is won by $\mathcal{M}_B$.

If E is the left copy of $⍰F$, its being $\top$-won means that there is an infinite bitstring y such that the thread y of F is won by $\mathcal{M}_A$. But, as we have already observed, $\mathcal{M}_B$ plays in the thread $0y$ of F (within the $⍰F$ component of B) exactly as $\mathcal{M}_A$ plays in the thread y of F within the left $⍰F$ component of A. Therefore, the thread $0y$ of F is won by $\mathcal{M}_B$, and hence so is the $⍰F$ component of B, and hence so is (the $⍰F$-containing) undergroup U_i^B.

The case of E being the right copy of $⍰F$ is similar.

10.5 Duplication

In this and the remaining subsections of the present section, whenever $\mathcal{M}_A$ is assumed to be a BMEPM, for simplicity we will pretend that it (unlike its imaginary adversary) never makes more than one move at once. For, otherwise, a block of several moves made by $\mathcal{M}_A$ at once will be translated through several consecutive moves (or several consecutive series of moves) by $\mathcal{M}_B$ as was pointed out in the preceding subsection.

Undergroup Duplication does not modify the game associated with the cirquent, so we only need to consider Overgroup Duplication.

For two (finite or infinite) bitstrings x and y, we say that a bitstring z is a **fusion** of x and y iff z is a shortest bitstring such that, for any natural numbers i, j such that x has at least i bits and y has at least j bits, we have:

- the $(2i-1)$'th bit[15] of z exists and it is the i'th bit of x;
- the $(2j)$'th bit of z exists and it is the j'th bit of y.

For instance, the strings 000 and 11 have only one fusion, which is 01010; the strings 000 and 111 also have one fusion, which is 010101; the strings 000 and 1111 have

[14] Of course, strictly speaking, either cirquent may contain additional copies of $⍰F$. But, as hopefully understood, "the first (resp. second) copy of $⍰F$ in A" in the present context means the a'th (resp. b'th) oformula of A. Similarly for B.

[15] Here and later the count of bits starts from 1, and goes from left to right.

two fusions, which are 01010101 and 01010111. Note that when both x_1 and x_2 are infinite, they have a unique fusion.

The **defusion** of a bitstring z is the pair (x_1, x_2) where x_1 (resp. x_2) is the result of deleting from z all bits except those that are found in odd (resp. even) positions. For instance, the defusion of 01011010 is (0011, 1100).

Assume B is obtained from A by Overgroup Duplication. We further assume that the machine $\mathcal{M}_A$ is a BMEPM, and that the duplicated overgroup is the last overgroup of the premise. Let $n+1$ be the number of overgroups in A. Thus, every legal move of A (resp. B) looks like $a; \vec{w}, u.\alpha$ (resp. $a; \vec{w}, u_1, u_2.\alpha$), where a is a positive integer not exceeding the number of oformulas, $\vec{w}$ is a sequence of n finite bitstrings, u, u_1, u_2 are finite bitstrings, and α is some move.

As always, $\mathcal{M}_B$ works by simulating $\mathcal{M}_A$. Whenever the simulated $\mathcal{M}_A$ makes a move $a; \vec{w}, u.\alpha$, $\mathcal{M}_B$ makes the move $a; \vec{w}, u_1, u_2.\alpha$, where (u_1, u_2) is the defusion of u. Next, whenever the adversary of $\mathcal{M}_B$ makes a move $a; \vec{w}, u_1, u_2.\alpha$ in the real play of B, $\mathcal{M}_B$ translates it as a block of $\mathcal{M}_A$'s imaginary adversary's moves in B. Namely, as the block $a; \vec{w}, v_1.\alpha, \ \ldots, \ a; \vec{w}, v_p.\alpha$ of p moves, where $v_1, \ldots, v_p$ are all the fusions of u_1 and u_2.

The idea behind the above strategy can be summarized by saying that $\mathcal{M}_B$ sees (and plays) every thread $\vec{y}, x_1, x_2$ of every oformula F_a of B exactly as $\mathcal{M}_A$ sees (and plays) the thread $\vec{y}, x$ of the same oformula F_a of A, where x is the fusion of x_1 and x_2. In precise terms this means that whenever Γ is a run of B generated by $\mathcal{M}_B$ and Ω is the corresponding run of the imaginary play of A by $\mathcal{M}_A$, for every oformula $\#a$ of either cirquent, every array $\vec{y}$ of n infinite bitstrings and any infinite bitstrings x_1 and x_2, we have $\Gamma^{\preceq a;\vec{y},x_1,x_2} = \Omega^{\preceq a;\vec{y},x}$, where x is the fusion of x_1 and x_2 (and hence (x_1, x_2) is the defusion of x). For this reason, it is obvious that, as long as (because) $\mathcal{M}_A$ wins A, $\mathcal{M}_B$ wins B.

10.6 Merging

In this and the remaining subsections of the present section, we shall limit ourselves to explaining the work of $\mathcal{M}_B$, leaving it to the reader to verify that such an $\mathcal{M}_B$ wins B as long as $\mathcal{M}_A$ wins A. In each case, as before, $\mathcal{M}_B$ works by simulating and mimicking $\mathcal{M}_A$ after reinterpreting certain moves. We shall limit our descriptions of $\mathcal{M}_B$ to explaining what moves need to be properly reinterpreted and how, implicitly stipulating that any unmentioned sorts of moves are mimicked exactly as they are, without any changes.

Assume B is obtained from A by Merging. Namely, B is the result of merging in A the overgroups O_{n+1} and O_{n+2}, with A having $n+2$ overgroups. Note that every legal move of A (resp. B) looks like $a; \vec{w}, u_1, u_2.\alpha$ (resp. $a; \vec{w}, u.\alpha$), where a is a positive integer not exceeding the number of oformulas in either cirquent, $\vec{w}$ is a sequence of n finite bitstrings, u, u_1, u_2 are finite bitstrings, and α is some move. We further assume that $\mathcal{M}_A$ is a BMEPM.

This is what $\mathcal{M}_B$ does for every integer a not exceeding the number of oformulas in either cirquent:

If the a'th oformula of A is neither in O_{n+1} nor in O_{n+2}, $\mathcal{M}_B$ interprets every move $a; \vec{w}, \epsilon, \epsilon.\alpha$ made by $\mathcal{M}_A$ in the imaginary play of A as the move $a; \vec{w}, \epsilon.\alpha$ that $\mathcal{M}_B$ itself should make in the real play of B. And vice versa: $\mathcal{M}_B$ interprets every move $a; \vec{w}, \epsilon.\alpha$ by its environment in the real play of B as the move $a; \vec{w}, \epsilon, \epsilon.\alpha$ by $\mathcal{M}_A$'s adversary in the imaginary play of A.

If the a'th oformula of A is in O_{n+1} but not in O_{n+2}, $\mathcal{M}_B$ interprets every move $a; \vec{w}, u, \epsilon.\alpha$ made by $\mathcal{M}_A$ in the imaginary play of A as the move $a; \vec{w}, u.\alpha$ that $\mathcal{M}_B$ itself should make in the real play of B. And vice versa: $\mathcal{M}_B$ interprets every move $a; \vec{w}, u.\alpha$ by its environment in the real play of B as the move $a; \vec{w}, u, \epsilon.\alpha$ by $\mathcal{M}_A$'s adversary in the imaginary play of A.

The case of the a'th oformula of A being in O_{n+2} but not in O_{n+1} is similar.

Now assume the a'th oformula of A is in both O_{n+1} and O_{n+2}. $\mathcal{M}_B$ interprets every move $a; \vec{w}, u_1, u_2.\alpha$ by $\mathcal{M}_A$ in the imaginary play as the series $a; \vec{w}, v_1.\alpha, \ \ldots, \ a; \vec{w}, v_p.\alpha$ of its own moves in the real play, where $v_1, \ldots, v_p$ are all the fusions of u_1 and u_2. And $\mathcal{M}_B$ interprets every move $a; \vec{w}, u.\alpha$ by its environment as the move $a; \vec{w}, u_1, u_2.\alpha$ by $\mathcal{M}_A$'s imaginary environment, where (u_1, u_2) is the defusion of u.

10.7 Disjunction introduction

Assume B follows from A by Disjunction Introduction. Namely, the last—a'th—oformula of B is $E \vee F$, and the last two—a'th and b'th $(b = a + 1)$—oformulas of A are E and F.

In its simulation/play routine, $\mathcal{M}_B$ reinterprets every move $a; \vec{w}.\alpha$ (resp. $b; \vec{w}.\alpha$) made by either player in the imaginary play of A as the move $a; \vec{w}.0.\alpha$ (resp. $a; \vec{w}.1.\alpha$) by the same player in the real play of B, and vice versa.

10.8 Conjunction introduction

Assume B follows from A by Conjunction Introduction. Namely, the last—a'th—oformula of B is $E \wedge F$, and the last two—a'th and b'th $(b = a + 1)$—oformulas of A are E and F.

Our description of the work of $\mathcal{M}_B$ in this case is literally the same as in the case of Disjunction Introduction.

10.9 Recurrence introduction

Assume B follows from A by Recurrence Introduction. Namely, the a'th oformula of B is $\diamondsuit F$, and the a'th oformula of A is F. We also assume that n is the number of overgroups in B, and that the new overgroup emerged when moving from B to A is the last, $(n + 1)$'th overgroup of A. Below we let $\vec{w}$ range over sequences of n finite bitstrings, and let u range over finite bitstrings.

If b is an integer other than a, $\mathcal{M}_B$ simply reinterprets every move $b;\ \vec{w}, \epsilon.\alpha$ made by either player in the imaginary play of A as the move $b;\ \vec{w}.\alpha$ by the same player in the real play of B, and vice versa.

As for a, $\mathcal{M}_B$ reinterprets every move $a;\ \vec{w}, u.\alpha$ made by either player in the imaginary play of A as the move $a;\ \vec{w}.u.\alpha$ by the same player in the real play of B, and vice versa. Note that the only difference between the two moves is that, in one case, we have a comma before u, and in the other case we have a period. That is because, in A, u is associated with an overgroup (the overgroup $\#n+1$), while in B it is associated with a $\downarrow$ (the $\downarrow$ applied to F) instead.

10.10 Corecurrence introduction

Assume B follows from A by Corecurrence Introduction. Namely, the a'th oformula of B is $\Diamond F$, and the a'th oformula of A is F. We also assume that n ($n \geq 0$) is the number of the overgroups U_j such that the a'th oformula is contained in U_j within A but not within B (i.e., n is the number of the *new* overgroups in which the a'th oformula was included when moving from B to A), and that all of such n overgroups are at the end of the list of overgroups of either cirquent. Below we let $\vec{w}$ range over sequences of m finite bitstrings, where m is the total number of overgroups of either cirquent minus n. Our construction of $\mathcal{M}_B$ depends on whether $n = 0$ or $n \geq 1$. We consider these two cases separately.

10.10.1 The case of $n = 0$

Intuitively, winning $\Diamond F$ is at least as easy for $\top$ as winning F. This is so because, when playing $\Diamond F$, $\top$ can focus on one single thread—say, the thread $000\ldots$—of (the otherwise many threads of) F, play in that thread as it would simply play in F, and safely ignore all other threads, for winning in a single thread is sufficient. Next, notice that, in the present case (of $n = 0$), A only differs from B in that the latter has $\Diamond F$ where the former has F. Therefore, winning B is at least as easy as winning A.

In more detail, let z stand for the infinite string of 0's. In its simulation/play routine, $\mathcal{M}_B$ reinterprets every move $a;\ \vec{w}.\alpha$ made by $\mathcal{M}_A$ in the imaginary play as its own move $a;\ \vec{w}.u.\alpha$ in the real play, where u is a "sufficiently long" finite initial segment of z—namely, such that u is not a proper prefix of any other finite bitstring v already used in the real play within some move $a;\ \vec{w}'.v.\beta$.[16] Next, whenever the environment makes a move $a;\ \vec{w}.v.\beta$ in the real play, if v is not a prefix of z, $\mathcal{M}_B$ simply ignores it, and if v is a prefix of z, $\mathcal{M}_B$ translates it as the move $a;\ \vec{w}.\beta$ by $\mathcal{M}_A$'s adversary in the imaginary play.

10.10.2 The case of $n \geq 1$

First we generalize to n ($n \geq 1$) the concepts of fusion and defusion introduced in Sect. 10.5 for the special case of $n = 2$.

[16] If u is not "sufficiently long", the move $a;\ \vec{w}.u.\alpha$ may turn out to be illegal.

Consider any n finite or infinite bitstrings $x_1, \ldots, x_n$. We say that a bitstring z is a **fusion** of $x_1, \ldots, x_n$ iff z is a shortest bitstring such that, for any $i \in \{1, \ldots, n\}$ and any positive integer j not exceeding the length of x_i, the following condition is satisfied:

- the $(jn - n + i)$'th bit of z exists and it is the j'th bit of x_i.

For instance, the strings 11, 00 and 111 have four fusions, which are 101101001, 101101011, 101101101 and 101101111. As before, when all n strings are infinite, they have a unique fusion.

Next, the **n-defusion** of a bitstring z is the n-tuple $(x_1, \ldots, x_n)$, where each x_i $(1 \leq i \leq n)$ is the result of deleting from z all bits except those that were found in positions j such that j modulo n equals i. For instance, the 3-defusion of 01011010 is (011, 110, 00).

In its simulation/play routine, $\mathcal{M}_B$ reinterprets every move $a; \vec{w}, u_1, \ldots, u_n.\alpha$ made by $\mathcal{M}_A$ in the imaginary play of A as the series

$$a; \vec{w}, \epsilon, \ldots, \epsilon.v_1.\alpha, \quad \ldots, \quad a; \vec{w}, \epsilon, \ldots, \epsilon.v_p.\alpha$$

(n occurrences of ϵ after $\vec{w}$ in each move; p moves altogether) of its own moves in the real play of B, where $v_1, \ldots, v_p$ are all the fusions of $u_1, \ldots, u_n$. And "vice versa": $\mathcal{M}_B$ reinterprets every move $a; \vec{w}, \epsilon, \ldots, \epsilon.u.\alpha$ made by its environment in the real play of B as the move $a; \vec{w}, u_1, \ldots, u_n.\alpha$ made by $\mathcal{M}_A$'s environment in the imaginary play of A, where $(u_1, \ldots, u_n)$ is the n-defusion of u.

References

1. Avron, A.: A constructive analysis of RM. J. Symb. Logic **52**, 939–951 (1987)
2. Blass, A.: A game semantics for linear logic. Ann. Pure Appl. Logic **56**, 183–220 (1992)
3. Girard, J.: Linear logic. Theor. Comput. Sci. **50**, 1–102 (1987)
4. Guglielmi, A.: A system of interaction and structure. ACM Trans. Comput. Logic **8**, 1–64 (2007)
5. Japaridze, G.: Introduction to computability logic. Ann. Pure Appl. Logic **123**, 1–99 (2003)
6. Japaridze, G.: Propositional computability logic I. ACM Trans. Comput. Logic **7**, 302–330 (2006)
7. Japaridze, G.: Propositional computability logic II. ACM Trans. Comput. Logic **7**, 331–362 (2006)
8. Japaridze, G.: From truth to computability I. Theor. Comput. Sci. **357**, 100–135 (2006)
9. Japaridze, G.: Introduction to cirquent calculus and abstract resource semantics. J. Logic Comput. **16**, 489–532 (2006)
10. Japaridze, G.: The logic of interactive Turing reduction. J. Symb. Logic **72**, 243–276 (2007)
11. Japaridze, G.: From truth to computability II. Theor. Comput. Sci. **379**, 20–52 (2007)
12. Japaridze, G.: Intuitionistic computability logic. Acta Cybern. **18**, 77–113 (2007)
13. Japaridze, G.: The intuitionistic fragment of computability logic at the propositional level. Ann. Pure Appl. Logic **147**, 187–227 (2007)
14. Japaridze, G.: Cirquent calculus deepened. J. Logic Comput. **18**, 983–1028 (2008)
15. Japaridze, G.: Sequential operators in computability logic. Inf. Comput. **206**, 1443–1475 (2008)
16. Japaridze, G.: In the beginning was game semantics. In: Majer, O., Pietarinen, A.-V., Tulenheimo, T. (eds.) Games: Unifying Logic, Language and Philosophy, pp. 249–350. Springer, Berlin (2009)
17. Japaridze, G.: Many concepts and two logics of algorithmic reduction. Studia Logica **91**, 1–24 (2009)
18. Japaridze, G.: Towards applied theories based on computability logic. J. Symb. Logic **75**, 565–601 (2010)
19. Japaridze, G.: Toggling operators in computability logic. Theor. Comput. Sci. **412**, 971–1004 (2011)
20. Japaridze, G.: From formulas to cirquents in computability logic. Logical Methods Comput. Sci. **7**(2), Paper 1, 1–55 (2011)

21. Japaridze, G.: Introduction to clarithmetic I. Inf. Comput. **209**, 1312–1354 (2011)
22. Japaridze, G.: A new face of the branching recurrence of computability logic. Appl. Math. Lett. **25**, 1585–1589 (2012)
23. Japaridze, G.: Separating the basic logics of the basic recurrences. Ann. Pure Appl. Logic **163**, 377–389 (2012)
24. Japaridze, G.: A logical basis for constructive systems. J. Logic Comput. **22**, 605–642 (2012)
25. Japaridze, G.: The taming of recurrences in computability logic through cirquent calculus, Part II. Arch. Math. Logic (to appear)
26. Japaridze, G.: Introduction to Clarithmetic II. Manuscript at http://arxiv.org/abs/1004.3236
27. Japaridze, G.: Introduction to Clarithmetic III. Manuscript at http://arxiv.org/abs/1008.0770
28. Kolmogorov, A.N.: Zur Deutung der intuitionistischen Logik. Mathematische Zeitschrift **35**, 58–65 (1932)
29. Kwon, H., Hur, S.: Adding sequential conjunctions to Prolog. Int. J. Comput. Appl. Technol. **1**, 1–3 (2010)
30. Kwon, H.: Adding a loop construct to Prolog. Int. J. Comput. Appl. Technol. **2**, 121–123 (2011)
31. Mezhirov, I., Vereshchagin, N.: On abstract resource semantics and computability logic. J. Comput. Syst. Sci. **76**, 356–372 (2010)
32. Pottinger, G.: Uniform, cut-free formulations of T, S4 and S5 (abstract). J. Symb. Logic **48**, 900 (1983)
33. Xu, W., Liu, S.: Knowledge representation and reasoning based on computability logic. J. Jilin Univ. **47**, 1230–1236 (2009)
34. Xu, W., Liu, S.: Deduction theorem for symmetric cirquent calculus. Adv. Intell. Soft Comput. **82**, 121–126 (2010)
35. Xu, W., Liu, S.: Soundness and completeness of the cirquent calculus system CL6 for computability logic. Logic J. IGPL **20**, 317–330 (2012)