Fundamenta Informaticae 28 (1996) 37–54
IOS Press

Designing Dependencies*

Howard A. Blair
School of Computer and Information Science
Syracuse University
Syracuse, New York, USA

Abstract. Given a binary recursively enumerable relation R, one or more logic programs over a language $\mathbf{L}$ can be constructed and interconnected to produce a dependency relation D on selected predicates within the Herbrand base $\mathbf{B_L}$ of $\mathbf{L}$ isomorphic to R. D can be, optionally, a positive, negative or mixed dependency relation. The construction is applied to representing any effective game of the type introduced by Gurevich and Harrington, which they used to prove Rabin's decision method for S2S, as the dependency relation of a logic program. We allow games over an infinite alphabet of possible moves. We use this representation to reveal a common underlying reason, having to do with the shape of a program's dependency relation, for the complexity of several logic program properties.

Keywords: logic program, dependency relation, game, complexity

1. Introduction

Results on the expressive power of logic programs and the complexity or undecidability of various logic program properties obviously depend to a considerable extent on representing various relations with certain desired properties as models of a program, often where the models themselves have additional properties such as being stable or supported, for example. Since there is about as much variety in the proof techniques that have been used to obtain these results as there is in the results themselves, it would be clarifying to have a reasonably uniform means of obtaining them.

Given a binary recursively enumerable relation R, one or more logic programs over a language $\mathbf{L}$ can be constructed and interconnected to produce a dependency relation D on selected predicates within the Herbrand base $\mathbf{B_L}$ of $\mathbf{L}$ isomorphic to R. D can be, optionally, a positive, negative or mixed dependency relation.

We introduce a game-theoretic approach through which many results having to do with complexity, degrees of unsolvability and expressive power of logic programs can be obtained in a uniform way. [10, 15]. After showing how the game trees of arbitrary effective Gurevich-Harrington (GH) games can be represented as dependency relations in programs, we will apply the games to give two results about the degree of unsolvability of certain logic program properties. One is previously known, but a new and simpler proof is given, and the other appeared only in a preliminary version of this paper [7].

The main contribution of this paper is to show that the game-theoretic approach taken here is a useful, unifying device for complexity investigations. The argument for this point

*Research partially supported by the U.S. Army Research Office through the Mathematical Sciences Institute of Cornell University.

is that two theorems, which at first sight appear to be quite different, the one having to do with models of definite clause programs (where negations do not occur in program clause bodies), the other having to do with the property of local stratification (which appears to be intrinsically about dependencies on negations within the program), are actually two instances of the same underlying theorem about the degree of undecidability of the class of winning strategies for the games.

One class of GH games have the feature that winning plays for one of the players (player 0) correspond to well-founded sequences of dependencies among atoms in the ground-instantiated version of the corresponding programs. It will be seen that it is easy to control the degree of unsolvability of the class of winning plays available to one of the players by adjusting the parameters of the game. By using the correspondence and varying the logical connection between the players, and hence varying the type of dependency relation embodied by the overall program, the complexity of various properties of the program can be read off.

We first define the games and give an obvious preliminary representation of the games as logic programs. Then, since what is logically expressed by a program's clauses is to be closely related to the program's dependency relation, the third section discusses converting a definite clause program into a binary definite clause program, which is a convenient device for coupling logical dependency to calling dependency.

We will then be in a position to see how to represent the game trees of GH games as dependency relations in programs. Within such programs, certain subprograms represent the players. By varying computable parameters within the player programs and by varying the manner in which the player programs are connected, we will be able to read off diverse results having to do with complexity and degrees of unsolvability associated with logic programs. It will become clear that the various manners in which programs may be connected are simple and do not have to hide encodings of complex properties. In particular, we will show that two quite distinct complete Π^1_1 properties of logic programs, namely unique fixed points of Horn clause programs, and local stratification, owe their high degree of unsolvability to the same underlying property.

We assume that the reader is familiar with the basics of the foundations of logic programming. An excellent, widely available introduction the subject is by K. R. Apt, [2]. In most cases readers who are insufficiently familiar with logic programming will have their puzzlements cleared up with a few minutes' perusal of Apt's article.

2. Gurevich-Harrington Games

To present Gurevich-Harrington games [10] we follow an amalgamation of the approaches of Yakhnis and Yakhnis [15] and Gurevich and Harrington [10].

Definition 2.1. *(GH-games).* There are two players, designated for convenience as 0 and 1. Thus, if p is a player then $1-p$ is her opponent. Player 0 is assumed to play first. Moves alternate between the players. To begin specifying a GH-game an alphabet Σ is fixed. The approach used in this paper permits Σ to be infinite, which extends the notion of game originally presented in [10]. A *play* is an infinite sequence of elements of the alphabet Σ. A finite (possibly empty) prefix of a play is a *position.* A *move* of a player consists of choosing a letter σ from Σ and appending it (i.e. suffixing it) to the end of a position to form another position. Let $\mathcal{P}$ be the set of all plays over the alphabet Σ. $\mathcal{P}$ may be considered as a tree whose nodes comprise the set of all positions over Σ. A *Gurevich-Harrington game (GH-game)* is specified by (1) a *game tree* $\mathcal{G}$ which is a subset of $\mathcal{P}$, and (2) a subset W of $\mathcal{P}$ which is called the *winning set* for player 0. The complement of W in $\mathcal{G}$ is the *winning set* for player 1. (The descriptions of $\mathcal{P}$, $\mathcal{G}$ and W regard a tree as a set of paths rather than a set of nodes.) Game trees may contain leaf nodes. The notion of a play is extended to include positions that occur as leaf nodes in a game tree.

Yakhnis and Yakhnis reserved the term *GH-game* for those games whose winning sets are Boolean combinations of *basic* sets of plays where a basic set of plays $[C]$ has infinitely many positions (finite initial segments) in the set C of positions, called the *kernel* of $[C]$. We do not need to require or exploit this restriction for our current purposes.

Note that a game tree determines the possible moves available to the players in each position. For this reason, we may think of a GH-game **G**'s game tree as the *rules* of **G**. Conversely, a complete specification of the possible moves available to the players from every position that they can actually reach determines a game tree.

We want to focus on GH-games that can be played by deterministic and nondeterministic computing procedures. Such games have recursively enumerable game trees. In other words, there must be a uniform means of computably generating the possible moves available in every position that could actually be reached by the players starting from the empty position. A generate-and-test approach to find the possible moves is sufficient. This requirement is distinct from the more stringent requirement that the n^{th} possible move available in each reachable position be uniformly computable.

Definition 2.2. *(Effective GH-games).* Let $\mathcal{G}$ be the game tree of GH-game **G**. Let R be the set of all pairs $(\sigma_1 \cdots \sigma_n, \sigma)$ such that $\sigma_1, \ldots, \sigma_n, \sigma$ are elements of the alphabet of **G** and $\sigma_1 \cdots \sigma_n \sigma$ is a position in a play in $\mathcal{G}$. We refer to R as the set of *rules* of G. We say that **G** is *effective* iff the set of rules of **G** is recursively enumerable.

In the preceding definition we assume that the alphabet of **G**, Σ, is effectively given. Specifically, we could identify Σ with the set $\{0, 1, \ldots, k\}$ where k is the cardinality of Σ if Σ is finite, otherwise we could identify Σ with the set of natural numbers. It turns out that if R is recursive [recursively enumerable], then the set of positions that can be reached by the players is also recursive [recursively enumerable]. We leave this to the reader.

When not too much violence is done to the reader's sense of grammar, we will refer to the *set* of rules of **G** simply as the *rules of* **G**.

If player 0 moves first, positions in which player 0 moves, the collection of which is denoted by $\text{Pos}_G(0)$, are of even length, and positions in which player 1 moves, the collection of which is denoted by $\text{Pos}_G(1)$, are of odd length.

Before discussing the representation of games by logic programs with their dependency relations, we formally introduce strategies. Conceptually, a strategy is a means by which players can select moves. The next definition makes precise what is meant by a *deterministic strategy.*

Definition 2.3. Let $p \in \{0, 1\}$ be a player in game G. A *deterministic p-strategy* is a function $f : \text{Pos}_G(p) \longrightarrow \Sigma$ such that if $\alpha \in \text{Pos}_G(p)$ then $\alpha \cdot f(\alpha) \in \mathcal{T}_G$. The set of positions in $\mathcal{T}_G$ *consistent with* a deterministic p-strategy f is inductively defined by:

i) the empty sequence Λ is consistent with f.
ii) if α is consistent with f and $\alpha \in \text{Pos}_G(1-p)$ then every child of α is consistent with f.
iii) if α is consistent with f and $\alpha \in \text{Pos}_G(p)$ then $\alpha \cdot f(\alpha)$ is consistent with f.

A play is *consistent* with f if every position in the play is consistent with f. A deterministic p-strategy *wins* $G = \langle \mathcal{T}_G, p, W \rangle$ if every play consistent with f is in W. Player p *wins* G if there is a winning deterministic p-strategy.

After we show how to represent game trees we will focus attention in this paper on a class of games for which the winning strategy for player 0 involves entering into what we call a *well* of a binary relation R which is a certain kind of well-founded subrelation of R. The recursion-theoretic complexity of wells of recursive and recursively enumerable binary relations can be controlled so as to vary up through the Π^1_1 sets. We will in turn use this property of wells to control the complexity of the winning strategies in the games on which we focus. This will enable us to unify two quite different Π^1_1-completeness results about logic program properties.

3. Games as Logic Programs

Next, we show how to represent an effective GH-game as a logic program where the players are represented as procedures and positions are passed between players through calls of one player by another.

Suppose **G** is an effective GH-game. The set of rules of **G** is a binary relation between positions and members of the alphabet of **G**.

We want to be able to compute moves from various positions. The notion of *compute* that we will need for logic programs is the obvious one in terms of least models, and was formalized in *cf.* [5, 2]. Among the most elegant early treatments of computability in logic programming is due to Andreka and Nemeti, [1]. However, in that paper, details concerning computability over effectively presented Herbrand universes are not treated, the authors having restricted their treatment to Herbrand universes, isomorphic to the natural numbers, generated by a single constant and unary function symbol. The following slight elaboration that incorporates auxiliary function and predicate symbols is a great convenience.

Definition 3.1. Let the signature of $\mathbf{L}'$ be a subset of the signature of $\mathbf{L}$. Let R be an n-ary relation over the Herbrand universe $U(\mathbf{L}')$ of $\mathbf{L}'$ and let S be the relation computed by (P, p). That is, for all terms $t_1, \ldots, t_n$ in the Herbrand universe $U(\mathbf{L})$

$$S(t_1, \ldots, t_n) \text{ iff } P \models p(t_1, \ldots, t_n)$$

Then (P, p) *computes* R *with respect to* $\mathbf{L}'$ iff $S \cap U(\mathbf{L}')^n = R$.

The following lemma can be established by a variety of means; in particular, *cf.* [5, 6]. There are difficulties that arise when dealing with languages with infinite signatures, [6]. These difficulties are avoided with the technical restriction concerning finite signatures in the following lemma. The statement of the lemma presupposes that the signature of the language $\mathbf{L}$ has been effectively given.

Lemma 3.1. Let the signature of first-order language $\mathbf{L}'$ be a finite subset of the signature of $\mathbf{L}$. Let R be an n-ary recursively enumerable relation over the Herbrand universe $U(\mathbf{L}')$ of $\mathbf{L}'$. Then we can effectively construct a definite (Horn) clause program P with n-ary relation symbol p such that (P, p) computes R with respect to $\mathbf{L}'$.

Computing with respect to L' is useful when we do not want to have to keep track of the extra tuples computed by a program due to the introduction of `cons`, `nil`, and the various f_p function symbols introduced in the construction of *binary extensional equivalent programs*, discussed in section 4.

We immediately apply the lemma to computing the rules of **G**. All we need to do is represent the set of rules of **G** as a binary relation on an Herbrand universe of a language with a finite signature. It is important not to unnecessarily multiply the size of terms representing positions, particularly when the alphabet of **G** is finite.

When the alphabet is infinite, three function symbols suffice. (Of course, we could get away with just two, but that would uselessly complicate the narrative.) We use the binary function symbol `b`, the unary function symbol `s` and the constant `0`. We assume the symbols of **G**'s alphabet Σ are represented as natural numbers, and a natural number k is represented, in turn, by the term

$$\underbrace{\mathtt{s(s(}\cdots\mathtt{s}\mathtt{(0)}\cdots\mathtt{))}}_{k}$$

which later on we will use in the abbreviated form $\mathtt{s}^k\mathtt{(0)}$.

Now, if $\sigma \in \Sigma$, we will *identify* σ with its numerical representation and just write σ instead. Let $\mathcal{F}_{\mathbf{G}}$ be the set of function symbols $\{\mathtt{b}, \mathtt{s}, \mathtt{0}\}$. If $\Sigma = \{\mu_1, \ldots, \mu_k\}$ is finite

we can dispense with numerical representations altogether and just use the elements of Σ as constants, in which case we take $\mathcal{F}_{\mathbf{G}}$ to be the set of function symbols $\{\mathtt{b}, \mu_1, \ldots, \mu_k\}$.

Fix a language for first-order logic, $\mathbf{G_L}$, whose function symbols are those in $\mathcal{F}_{\mathbf{G}}$ and whose predicate symbols contain all those we will use below in specifying the representation of the effective GH-game, **G**.

With these notational conventions we can represent the position $\sigma_1 \cdots \sigma_n$ by the term $\mathtt{b}(\sigma_n,\mathtt{b}(\sigma_{n-1}, \ldots, \mathtt{b}(\sigma_1,\mathtt{0})\cdots))$. We use the list notation of PROLOG to abbreviate the clumsy terms involving the symbol b. The above term representing a position is written $[\sigma_n, \ldots, \sigma_1]$. A nonground term such as `b(X,Y)` is written `[X|Y]`. We write the sequence of moves in a position backwards when representing the position as a list of moves because we want to append moves to the "right" end of positions by extending finite sequences. Lists, being syntactic terms, are much more easily prefixed with new elements than appended with them. These representational conventions allow us to identify the rules of **G**, R, with a binary relation on the Herbrand universe $U(\mathbf{G_L})$ which we also denote by R.

We use the symbols ρ, ρ' with and without subscripts to refer to positions *and* their representations according to the above conventions. Similarly we use σ and μ with and without subscripts to refer to moves.

Since we have supposed that **G** is effective, R is recursive enumerable. Apply lemma 3.1. We obtain a program $P0_R$ in which a predicate symbol `r0` occurs such that $(P0_R, \mathtt{r0})$ computes R with respect to $\mathbf{G_L}$ and no function symbol occurs in $P0_R$ other than those in $\mathcal{F}_{\mathbf{G}}$.

We follow the notational conventions of PROLOG, using lower case identifiers to denote predicate and function symbols, and identifiers that begin with an upper case letter to denote variables.

To represent game trees as dependency relations in programs where positions are passed between players through calls of one player by another, we will need to carefully control how a ground atom instantiating a literal occurring in the clauses representing one player may depend on a ground atom instantiating a literal occurring in the clauses representing the other player. The technique that we will use requires that most of the predicate symbols occurring in one player's representing clauses do not occur in the other player's representing clauses. To meet this requirement, we will make a "copy" $P1_R$ of the clauses in $P0_R$, where the clauses of $P1_R$ are obtained from the clauses of $P0_R$ by renaming predicate symbols as necessary so that no predicate symbol that occurs in $P1_R$ also occurs in $P0_R$, since both players need to be able to use the clauses for computing R to choose moves. In particular, we assume that `r0` is renamed to `r1`.

Definition 3.2. To represent **G** we collect into a program $P_{\mathbf{G}}$ the clauses of $P0_R$ and $P1_R$ together with the following two clauses

```
player0(Position,Move) ← r0(Position,Move)
                         ∧ player1([Move|Position],Move1)
player1(Position,Move) ← r1(Position,Move)
                         ∧ player0([Move|Position],Move1)
```

$P_{\mathbf{G}}$ is the *definite clause representation* of **G**.

The sense in which $P_{\mathbf{G}}$ represents **G** needs an explanation. The next proposition provides such an explanation. After we introduce *binary extensional equivalents* in the next section, we will be able to see the game tree of **G** represented in the dependency relation of a set of clauses closely related to the the binary extensional equivalent of $P_{\mathbf{G}}$.

In the statement of the proposition that follows we adopt the notation that if ϵ is a $\{0,1\}$ valued-expression then `player`ϵ is `player0` if $\epsilon = 0$ and is `player1` if $\epsilon = 1$. We will continue to use this and similar obvious notation in the remainder of the paper.

Proposition 3.1. Suppose **G** is an effective GH-game and ρ is a position in which player i is to choose a move. Let ρ' be a position in which player j is to choose a move. (i may or may not equal j.) Then

there is a path in the game tree of **G** from ρ to ρ' beginning with move μ
iff
$P_{\mathbf{G}} \cup \{\texttt{player}j(\rho', t)\} \models \texttt{player}i(\rho, \mu)$, for any ground term t.

Hereafter, we assume we are working with a fixed language **L** for first order logic without identity. We assume that **L** contains each of the function and predicate symbols that we will use. Although it is not strictly necessary, it is a bit more natural to think of the binary extensional equivalent programs, to be discussed in the next section, as being finite whenever the program from which one is derived is finite. For this purpose, the signature of **L** needs to be finite, and in fact we will use only finitely many function and predicate symbols.

4. Binary Logic Programs

Binary logic programs are definite clause programs with at most one atom occurring in the body of each clause. Binary programs are important because for such programs entailment and dependency coincide. This relationship will be made precise in proposition 4.1., below.

A difficulty with controlling dependencies in programs is due to the fact that if conjunctions in clause bodies are replaced by disjunctions, then the dependency relation of the resulting program is the same as that of the original, but the models of the resulting program are, in general, vastly different. Another way to look at the difficulty is by considering a clause such as

```
p(x) ← q(x,y), r(y)
```

contained in some program P which has a least model in which, for example, `q(a,b)` is false. `p(a)` still depends on `r(b)`, but this was perhaps not intended. We would like to control dependencies through the semantics of the program. This is achievable by converting a program P to a binary program which has the same least model as P with respect to the predicates defined in P.

Definition 4.1. Let P be a normal logic program *cf.* [12], and let grd(P) be the set of ground clauses which are instances of clauses in P. The relations *refers positively to* and *refers negatively to* are defined by

A refers positively [negatively] to B
iff
there is a clause $A \leftarrow L_1, \ldots, L_n \in$ grd(P) such that B [$\neg B$] is L_i for some $i \in \{1, \ldots, n\}$.

Define the *depends positively on* relation to be the reflexive transitive closure of the *refers positively to* relation, and let the *depends negatively on* relation be

(*depends positively on*) $\circ$ (*refers negatively to*) $\circ$ (*depends positively on*)

where $R_1 \circ R_2$ denotes the composition of R_1 and R_2. When only definite clauses occur in P, we say, simply, *refers to* in place of *refers positively to* and *depends on* in place of *depends positively on.*

Definition 4.2. Let ground atom A depend positively on ground atom B with respect to program P. Then the pair (A, B) is said to be a *logical dependency* iff $P \cup \{B\} \models A$. A program is *dependency sound* if every pair of ground atoms in the positive dependency relation of P is a logical dependency.

Definition 4.3. A *binary* logic program is a program where each program clause either has the form $A \leftarrow B$ or is a unit clause A, where A and B are atoms.

The following proposition shows how binary programs "equate" entailment and dependency.

Proposition 4.1. Every binary program is dependency sound.

Definition 4.4. Let L be a first order language and let P_1, P_2 be definite clause logic programs over L. Let L' be a sublanguage of L and suppose the restrictions of the least models of P_1 and P_2 to the Herbrand base of L' are the same. Then P_1 and P_2 are said to be *extensionally equivalent* with respect to L'.

Definition 4.5. Let P be a definite clause program. Extend L to a language L' by adjoining a new function symbol f_p for each predicate symbol p in L. f_p has the same arity as p. Corresponding to each atom $p(t_1, \ldots, t_n)$ of L, the *translation*, $f_p(t_1, \ldots, t_n)$ is a term of L'. In general, for each atom A of L, let t_A denote the translation of A. Corresponding to P the *binary extensional equivalent* Q of P is defined as follows. Extend L' by adjoining a new binary predicate symbol `stack`, a new binary function symbol `cons` and a new constant symbol `nil`. Corresponding to each program clause

$$A \leftarrow B_1, \ldots, B_n$$

of P, form the clause

$$\texttt{stack}(\texttt{cons}(t_A, \texttt{Y}), \texttt{Z}) \leftarrow \texttt{stack}(\texttt{cons}(t_{B_1}, \texttt{cons}(t_{B_2}, \ldots, \textit{cons}(t_{B_n}, \texttt{Y}) \ldots)), \texttt{Z}).$$

Q also contains a *bridging clause* for each predicate symbol p:

$$p(\texttt{X1}, \ldots, \texttt{Xn}) \leftarrow \texttt{stack}(\texttt{cons}(f_p(\texttt{X1}, \ldots, \texttt{Xn}), \texttt{nil}), f_p(\texttt{X1}, \ldots, \texttt{Xn}))$$

Finally, Q contains the *terminating* clause

$$\texttt{stack}(\texttt{nil}, \texttt{Z}).$$

Occasionally, it will be convenient to be able to ensure that the *depends on* relation within binary definite clause programs is *Noetherian.*

Definition 4.6. A binary relation R on a set A is *well-founded* iff there is no sequence $\{a_n\}_{n=0}^{\infty}$ of elements of A such that

$$R(a_1, a_0),\ R(a_2, a_1), \ldots, R(a_i, a_{i-1}),\ R(a_{i+1}, a_i), \ldots$$

R is *Noetherian* (terminology borrowed from the literature of term-rewriting systems, *cf.* [11]) iff the converse of R is well-founded.
A *path* in R from a_0 to a_n is a finite sequence

$$a_0, a_1, \ldots, a_{n-1}, a_n \quad (n > 0)$$

of elements of A such that

$$R(a_{i-1}, a_i), \text{ for all } i = 0, \ldots, (n-1)$$

Thus, R is Noetherian iff from any element a of A, every path in R that starts from a is finite.

In the case of binary extensional equivalent programs, it will suffice to add a step-counter argument to the `stack` predicate for the *depends on* relation to be Noetherian.

Definition 4.7. The *step-counter augmentation* of a binary extensional equivalent program Q is obtained by adding a step-counter argument to each of the clauses in Q to obtain clauses of the form

$$\texttt{stack}(\texttt{s}(\texttt{S}), \texttt{cons}(t_A, \texttt{Y}), \texttt{Z}) \leftarrow$$
$$\quad \texttt{stack}(\texttt{S}, \texttt{cons}(t_{B_1}, \texttt{cons}(t_{B_2}, \ldots, cons(t_{B_n}, \texttt{Y})\ldots)), \texttt{Z})\,.$$
$$p(\texttt{X1}, \ldots, \texttt{Xn}) \leftarrow$$
$$\quad \texttt{stack}(\texttt{S}, \texttt{cons}(f_p(\texttt{X1}, \ldots, \texttt{Xn}), \texttt{nil}), f_p(\texttt{X1}, \ldots, \texttt{Xn}))$$
$$\texttt{stack}(\texttt{0}, \texttt{nil}, \texttt{Z})\,.$$

Proposition 4.2. The binary extensional equivalent of P is extensionally equivalent to P with respect to the language of P.

Proof:
Use the bridging clauses. □

Proposition 4.3. Let Q be the binary extensional equivalent of P and let A and B be ground atoms in the language of P. Then

$$A \text{ depends on } \texttt{stack}(\texttt{nil}, t_B) \text{ iff } Q \models A \text{ and } B \text{ is } A\,,$$

and similarly for step-counter augmentations.

The following proposition will be convenient when we come to considering programs with unique fixed points.

Proposition 4.4. Suppose P is a binary program without unit clauses, and therefore with an empty least model. Then P has no nonempty supported models iff the *depends on* relation of P is Noetherian.

Proof:
If the *depends on* relation of P is not Noetherian then there is an infinite sequence of ground atoms

$$A_0, \ldots, A_n, \ldots$$

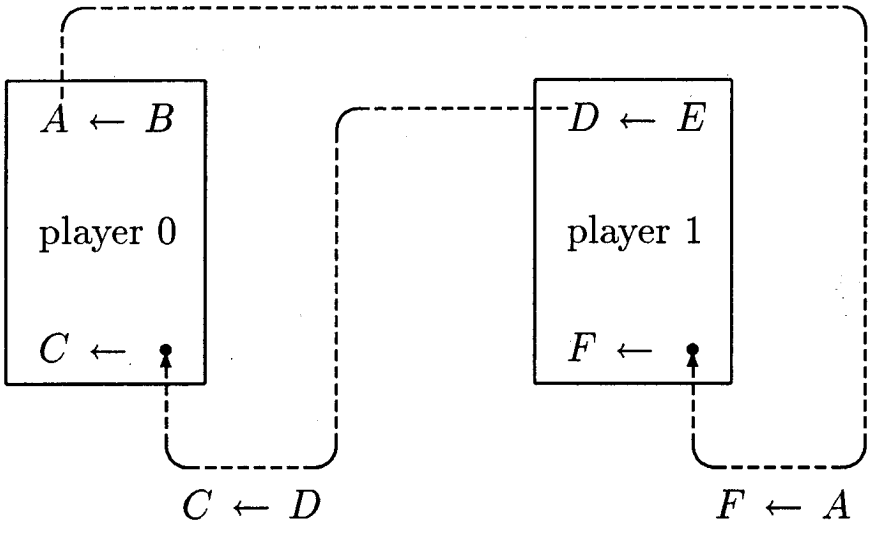
such that A_i depends on A_{i+1} for all $i \in \mathbf{N}$. Since P is binary, $\mathbf{T}_P(\{A_{i+1}\})$ contains A_i, for each i. Let I be the set of atoms in the above sequence. Then $I \subseteq \mathbf{T}_P(I)$. Hence, since $\mathbf{T}_P$ is monotonic, there is a fixed point of $\mathbf{T}_P$ above I, which is, a fortiori, nonempty. Conversely, if $\mathbf{T}_P$ has a nonempty fixed point then we have immediately that the *depends on* relation is not Noetherian since P has no unit clauses. □

5. Game Trees as Dependency Relations

In this section we complete the representation of the game trees of effective GH-games as dependency relations of logic programs. The figure below depicts the structure of the dependency relation in the program that we will form from the binary extensional equivalent of $P_{\mathbf{G}}$, the definite clause representation of effective GH-game $\mathbf{G}$. The idea is that dependency flows through each of the player representations without being able to cross between them except at selected entry and exit points.

Definition 5.1. Let $P^*_{\mathbf{G}}$ be the binary extensional equivalent of $P_{\mathbf{G}}$. We modify $P^*_{\mathbf{G}}$ by *replacing* the terminating clause

$$\texttt{stack}(\texttt{nil}, \texttt{Z}).$$



by the *connecting clauses*

$$\texttt{stack}(0, \texttt{nil}, f_{\texttt{player}i}(\texttt{Position}, \texttt{Move})) \leftarrow \texttt{player}(i-1)(\texttt{[Move|Position]}, \texttt{Move1}).$$

for $i = 0, 1$. The resulting program is called the *binary clause representation* of $\mathbf{G}$, and is denoted by $\mathrm{BR}_{\mathbf{G}}$.

For $i = 0, 1$ let Q_i be the binary extensional equivalent of the program consisting of the clauses in Pi_R together with

$$\begin{aligned}\texttt{player}i(\texttt{Position}, \texttt{Move}) \leftarrow\ & \texttt{r}i(\texttt{Position}, \texttt{Move}) \\ & \wedge\ \texttt{player}(i-1)(\texttt{[Move|Position]}, \texttt{Move1})\end{aligned}$$

The programs Q_0 and Q_1 will be convenient for proving the next proposition which establishes the correspondence between the game tree of $\mathbf{G}$ and the dependency relation of $\mathrm{BR}_{\mathbf{G}}$.

Proposition 5.1. Suppose $\mathbf{G}$ is an effective GH-game with rules R and ρ is a position in which player$i$ is to choose a move. Let ρ' be a position in which player$j$ is to choose a move. Then

there is a path in the game tree of $\mathbf{G}$ from ρ to ρ' beginning with move μ
iff
$\mathrm{BR}_{\mathbf{G}} \bigcup \{\texttt{player}j(\rho', t)\} \models \texttt{player}i(\rho, \mu)$
iff
$\texttt{player}i(\rho, \mu)$ depends on $\texttt{player}j(\rho', t)$ with respect to $\mathrm{BR}_{\mathbf{G}}$.

Proof:
Recall that $P^*_{\mathbf{G}}$ is the binary extensional equivalent of $P_{\mathbf{G}}$ and contains a terminating clause. By proposition 4.2. we have

$P^*_{\mathbf{G}} \models \texttt{player}i(\rho, \mu)$
iff
$\texttt{player}i(\rho, \mu)$ depends on $\texttt{stack}(0, \texttt{nil}, f_{\texttt{player}i}(\rho, \mu))$ with respect to $P^*_{\mathbf{G}}$.

The dependency soundness of $\mathrm{BR}_{\mathbf{G}}$, which is given by proposition 4.1., together with the previous equivalence implies, for any ground term t,

$\mathrm{BR}_{\mathbf{G}} \bigcup \{\texttt{player}j(\rho', t)\} \models \texttt{player}i(\rho, \mu))$
iff
$\texttt{player}i(\rho, \mu)$ depends on $\texttt{player}j(\rho', t)$ with respect to $\mathrm{BR}_{\mathbf{G}}$.

To complete the proof of the proposition it suffices to note that for any ground terms $t_1, t_2, t_3, t_4, t_5, t_6$ if

$$A \text{ is } \mathtt{stack}(\mathtt{s}^k(\mathtt{0}), \mathtt{cons}(f_{\mathtt{player}_i}(t_1, t_2), t_3)$$

and

$$B \text{ is } \mathtt{stack}(\mathtt{s}^{k'}(\mathtt{0}), \mathtt{cons}(f_{\mathtt{player}_{i'}}(t_4, t_5), t_6)$$

and

$$A \text{ refers to } B \text{ with respect to } \mathrm{BR}_{\mathbf{G}}$$

then

$$i = i'$$
$$k = k' + 1$$

and

$$A \text{ refers to } B \text{ with respect to } Q_i \text{ but not } Q_{i-1}. \qquad \square$$

Observe that having set up a positive dependency relation to represent the game tree of **G** we can easily set up a negative dependency relation to represent the same game tree by negating the literals in the bodies of the connecting clauses,

$$\mathtt{stack}(\mathtt{0}, \mathtt{nil}, f_{\mathtt{player}_i}(\mathtt{Position}, \mathtt{Move})) \leftarrow \neg\, \mathtt{player}(i-1)(\mathtt{[Move|Position]}, \mathtt{Move1})$$

to produce *negative connections.*

In the next section we will examine a class of GH-games which have well-founded game trees. The corresponding binary clause representation is locally stratified [8].

6. $\Gamma(R, x_0)$

We present an exact definition of a class of games in terms of two parameters, $x_0 \in \mathbf{N}$, where $\mathbf{N}$ is a fixed set of *nodes* and $R \subseteq \mathbf{N} \times \mathbf{N}$. We then illustrate the play of the games with an example in which R is represented as a finite directed graph, and discuss the nature of the games' winning strategies. Subsequently, we will be interested in games for which the underlying fixed set of nodes is (countably) infinite. Hence we will then identify N with the set of natural numbers $\mathbf{N}$.

Definition 6.1. $\Gamma(R, x_0)$ is played as follows:
Initially, set `z` := x_0. Player 0 *moves* first. The players alternate moves until one of them wins or loses. A *play* of the game is either a finite sequence of moves beginning with player 0's first move and ending with a move of player 1 resulting in a win for one of the players, or an infinite sequence of moves beginning with player 0's first move. The command `choose z` chooses a natural number and assigns it to the variable z.

player 0 executes: (`x := z; choose z`)
player 1 executes:

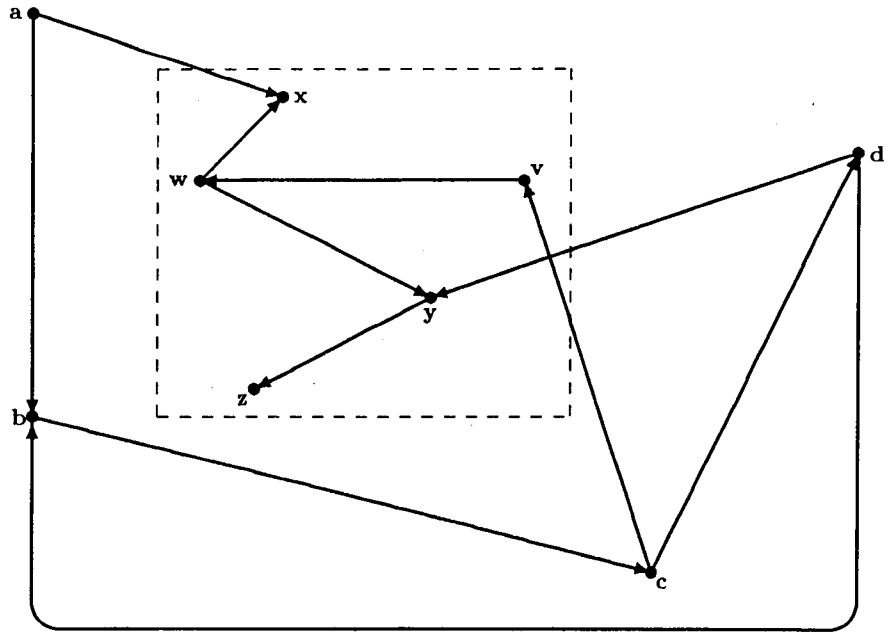
if R(x,z) **then** (`choose z`; **if not** R(x,z) **then** player 0 wins)
else (`choose z`; (**if** R(x,z) **then** player 1 wins **else** player 0 wins))

Example 6.1. Let R be the binary relation on the set of nodes

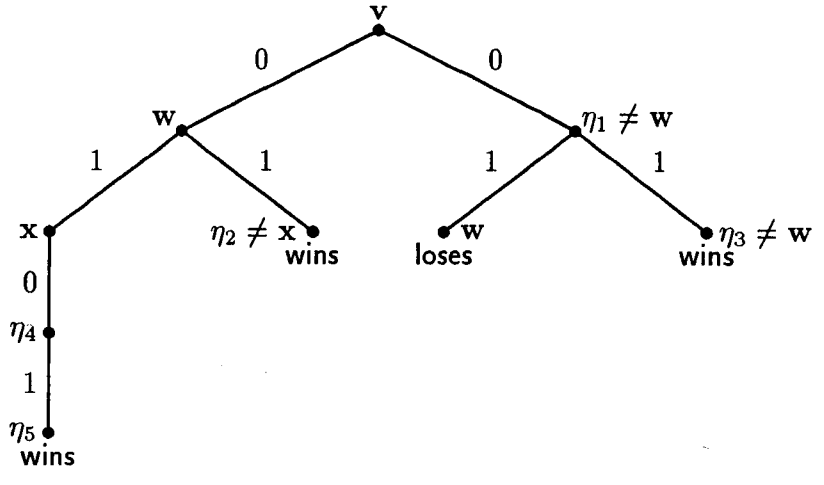
$$\mathcal{N} = \{\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}, \mathbf{v}, \mathbf{w}, \mathbf{x}, \mathbf{y}, \mathbf{z}\}$$

that is depicted below as a directed graph.

The game-tree below depicts all of the possible sequences of plays of the game $\Gamma(R, \mathbf{v})$. The edges are labeled by the player making the play, and the nodes into which the edges le-



ad are labeled by the value of z chosen during the play. Additionally, the leaves are labeled by whether player 0 wins or loses.

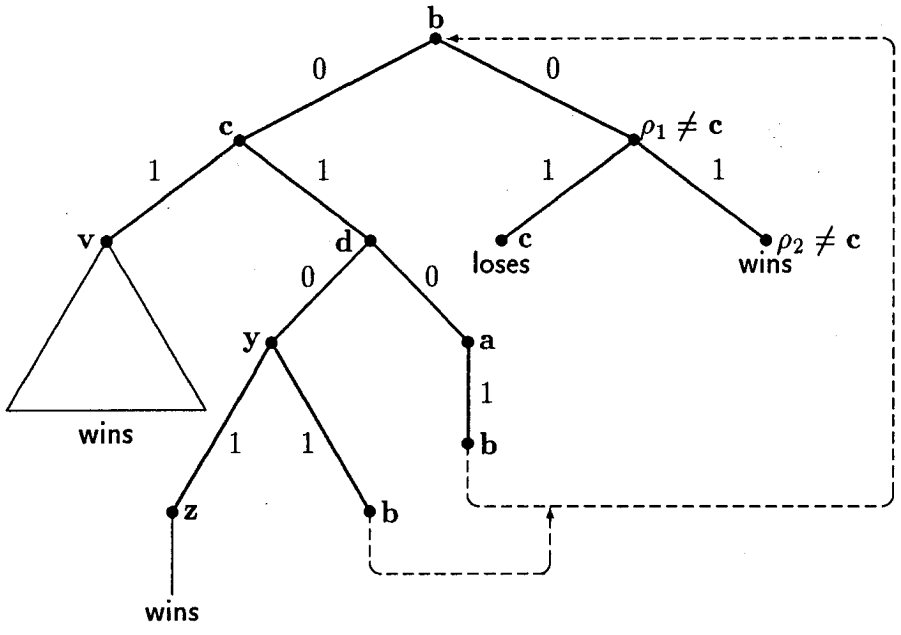


The above game tree node labels η_4 and η_5 can be any of the nodes in $\mathcal{N}$.

The next game tree below depicts all of the possible sequences of plays of the game $\Gamma(R, \mathbf{b})$.

Player 0 wins the game whenever player 1 makes a choice for the value of z that breaks the relation R or, in terms of directed graphs, chooses a node that is not immediately adjacent to the current node along the edges of the graph R. There are two ways player 1 can win. The first way occurs when player 0 breaks R but was not forced to. Player 1 has an immediate opportunity to re-choose the value of z to re-establish R and win (or lose if she fails to re-establish R during such a move.) The second way for player 1 to win occurs when he is able to keep the game going indefinitely without player 0 explicitly winning at any finite stage of the play. By definition, and consistent with the definition of *winning* in Gurevich-Harrington (GH) games, we define such an infinite play as a *winning* play for player 1. However, whether the reader thinks of an infinite play not otherwise won by player 0 as a win for player 1, or merely as a play without a winning outcome for either player is immaterial for our present purposes (although not for other purposes to which GH-games have been put in the literature

[15].) It follows that as soon as player 1 chooses a node η such that all paths in R originating at η are finite, (or if the game is initialized to such a node) then player 0 can force a win for herself. In order to keep most games from being always winnable by player 0, player 1 is given an opportunity to "correct" player 0's choice as long as the relation R can be maintained.



For example, in the game tree for $\Gamma(R, \mathbf{b})$, above, when, while at **d**, player 0 chooses **y**, player 1 has the opportunity to escape back to a node from which infinite paths originate by implicitly backing up to **d** and choosing **b**. It should also be noted that it is always in the interest of both players to maintain the relation R (where player 1 may implicitly back up one move to, in effect, replay player 0's move), if possible. Thus, in the game $\Gamma(R, \mathbf{b})$, if player 0 is to avoid loss outright, then she must choose **c** on her first move. Player 1 then will choose **d** in order to avoid choosing a node from which no infinite paths proceed. Player 0 then chooses **y** in order to try to get to such a node. (The alternative, choosing **a**, allows player 1 to effectively return to the initialized position of the game on his next move.) Player 1 then "corrects" player 0's choice by choosing **b** and the game is now as it was when initialized. In this way, player 1 has a winning strategy (or at at least a strategy to avoid loss) in the game $\Gamma(R, \mathbf{b})$.

We introduce some terminology for binary relations that will allow us to be more concise in describing winning strategies for the games $\Gamma(R, x_0)$. With the terminology made precise in the next definition we can say that the strategy for player 1 to avoid losing is for player 1 to avoid crossing the boundary of a *well* in R.

Definition 6.2. We borrow terminology from graph theory via the following notation and terminology: The *trace* $R_{A'}$ of R on a subset A' of A is defined by

$$R_{A'}(x, y) \text{ iff } x \in A',\ y \in A' \text{ and } R(x, y).$$

A subrelation R' of R is *full* iff R' is the trace of R on a subset A' of A.
We say that R' is a *well* in R iff R' is maximal in the set of full Noetherian subrelations of R. The idea is that there are no paths leading out of wells, and one cannot move along a path in a well indefinitely. We take maximal relations of this kind because players of the games are interested in boundaries of wells.

The *field* of R, denoted by $\mathrm{fld}(R)$, is defined by

$$\mathrm{fld}(R) = \{x \mid \exists y\, R(x, y)\} \cup \{x \mid \exists y\, R(y, x)\}$$

Let R' be a subrelation of R. Then the *boundary* of R' is the set of elements a of A such that

$$\{a \in A \mid \exists y \in \mathrm{fld}(R')[a \notin \mathrm{fld}(R') \wedge R(a,y)]\}.$$

A path in R from a to b *crosses* the boundary of a well W in R iff a is not in the well but b is in the well. (The last element of A in the path from a to b that is not in $\mathrm{fld}(W)$ is on the boundary of the well.) Note that a path does not terminate within the field of a well iff the path can be properly extended to a path with the same property. This completes definition 4.6.

Example 6.2. In the directed graph corresponding to the relation R of example 6.1. the relation $R_{\mathbf{V}}$ is the trace of the relation R on the nodes **v**, **w**, **x**, **y** and **z**. $R_{\mathbf{V}}$ is a well whose boundary consists of the nodes **a**, **c** and **d**.

Proposition 6.1. Player 0 wins $\Gamma(R,x)$ iff x is within the field of a well in R.

What makes $\Gamma(R,x_0)$ interesting for investigations of degrees of unsolvability is that computable R can be easily chosen to make the set of all x_0 such that player 0 wins $\Gamma(R,x_0)$ complete Π^1_1.

First, some notation: $\langle x,y\rangle$ is the code number (using a bijective pairing function) of the pair (x,y). If $c = \langle x,y\rangle$, then $(c)_0 = x$ and $(c)_1 = y$. The function φ_z is the z^{th} partial recursive function with respect to a fixed acceptable indexing. Equivalently, z is the index of W_z, the z^{th} recursively enumerable subset of $\mathbf{N}$. The notation is as in [14].

Lemma 6.1. Let $R(x,z) \leftrightarrow \varphi_{(x)_0}((z)_0)$ converges within $(z)_1$ steps. Then the set of all n_0 such that player 0 has a winning strategy in $\Gamma(R,n_0)$ is complete Π^1_1.

Proof:
(Sketch. *cf.* [14].) Let C be the productive center of the identity function. C is a complete Π^1_1 set. Player 0 has a winning strategy in $\Gamma(R,n)$ iff $(n)_0 \in C$. The strategy has player 0 always choose z such that $R(x,z)$ holds, unless $W_{(x)_0} = \emptyset$, in which case player 1 will lose on his next turn. Such a choice can always be made if play starts from $(n)_0 \in C$ because $(x)_0 \in C$ implies either $W_{(x)_0} = \emptyset$ or $\exists z\,[(z)_0 \in W_{(x)_0} \subseteq C]$. If player 0 chooses z by this strategy, then for player 1 to avoid losing, he must either confirm player 0's choice of z or choose $z' \neq z$ with the property that $(z')_0 \in W_{(x)_0} \subseteq C$. C is structured so that it has a well-ordered partition $C = \bigcup_{\gamma=0}^{\omega_1^{\mathrm{ck}}} C_\gamma$ such that $a \in C$ implies $a \in C_{\alpha+1}$ for some $\alpha < \omega_1^{\mathrm{ck}}$, which, in turn, implies $W_a \in C_\alpha$. (ω_1^{ck} is the least nonconstructive ordinal.) This property entails that eventually player 0 must be able to choose z such that $W_{(z)_0} = \emptyset$. □

In order to exclude certain unwanted entailments within the logic program representations of the games $\Gamma(R,n_0)$, we will use the following variation of the preceding lemma.

Corollary 6.1. Let $\Phi(Y) = \{x \mid x \in W_y \text{ for some } y \in Y\}$. Then $\bigcup_{i=0}^{\infty} \Phi^i(\{n\})$ is recursively enumerable. Define f by $W_{f(n)} = \bigcup_{i=0}^{\infty} \Phi^i(\{n\})$ and let

$$R_{n_0}(x,z) \leftrightarrow \varphi_{(x)_0}((z)_0)\,\varphi_{f(n_0)}((x)_0)$$

both converge within $(z)_1$ steps. Then the set of all n_0 such that player 0 has a winning strategy in $\Gamma(R_{n_0},n_0)$ is complete Π^1_1.

It should be observed that if R is recursively enumerable,then a winning strategy for player 1 in $\Gamma(R,n_0)$, if it exists, is in general recursive in a complete Π^1_1. Player 0's winning strategy, if it exists, is at worst recursive in the halting problem. At the cost of adjusting the games by complicating the relation R a little, we can focus on a class of games where player 0's strategies are recursive, while player 1's strategies are still in general recursive in a complete Π^1_1 set.

7. Representing Players

Note that, informally, the players in $\Gamma(R, x_0)$ nondeterministically map $\mathbf{N}$ to $\mathbf{N}$.

Definition 7.1. Let P_0 be the program consisting of only the unit clause $\mathtt{p0(X, Z0, WinLoss)}$. Informally, $\mathtt{Z0}$ is the new value chosen by player 0. $\mathtt{WinLoss}$ records whether player 0 wins or loses in a finite number of moves.
Let P_1 be the program

$$\mathtt{p1(X, Z, Z0, 0, 0)} \leftarrow \mathtt{p_R(X, Z, s(0))} \wedge \mathtt{p_R(X, Z0, 0)}.$$
$$\mathtt{p1(X, Z, Z0, WinLoss, s(0))} \leftarrow \mathtt{p_R(X, Z, s(0))} \wedge \mathtt{p_R(X, Z0, s(0))}.$$
$$\mathtt{p1(X, Z, Z0, WinLoss, 0)} \leftarrow \mathtt{p_R(X, Z, 0)} \wedge \mathtt{p_R(X, Z0, WinLoss)}.$$

$\mathtt{p_R}$ computes the characteristic function of relation R. The fifth argument of p1 is intended to record that play should continue when the second clause succeeds.

Suppose that R is a recursive relation. Let P_R be a definite clause program that computes the characteristic function of R using the predicate symbol $\mathtt{p_R}$.

Assume that there are no predicate symbols that occur in both programs P_0 and P_1. Assume also that the only predicate symbol that occurs in both programs P_1 and P_R is p_R. Further assume that p_0, the predicate symbol in the head of the clause in P_0 does not occur in program P_R. The nonintersection of the sets of predicate symbols occurring in these programs can easily be arranged without loss of generality by renaming predicate symbols as necessary. (That a program to compute the characteristic function of R using p_R can be constructed from an explicit definition of R can be established by a variety of techniques; in particular, see [13].) The game has to get started. For this purpose we introduce the following definition.

Definition 7.2. An *initializing clause* a clause of the form

$$\mathtt{start(s^y(0), WinLoss)} \leftarrow \mathtt{p0(s^y(0), Z0, WinLoss)}.$$

for some $y \in \mathbf{N}$.

We will set up the program corresponding to $\Gamma(R, x_0)$ in two stages. In the first stage we define the player programs assuming that the relation R is recursive and that the corresponding program P_R is at hand. In the second stage we show how to connect the player programs together so that play may pass between them. The means of connection will be regarded as an adjustable parameter involving the presence or absence of negation signs. We also want to have that the *depends on* relations with respect to each of the player programs, respectively, are Noetherian. We do this by adding a step-counter to the programs representing the players.

Definition 7.3. Let Q_0 be the binary extensional equivalent of P_0 and let $Q_{R,1}$ be the binary extensional equivalent of $P_1 \cup P_R$. Let $\mathbf{player}_0$ and $\mathbf{player}_{R,1}$ be the step-counter augmentations of Q_0 and $Q_{R,1}$, respectively. The predicate symbol $\mathtt{stack}$ in each of the two programs is assumed to be renamed so that the programs have no predicate symbols in common. We also further assume, without loss of generality, that the only function symbols occurring in program P_R are the unary symbol $\mathtt{s}$ and the constant symbol $\mathtt{0}$.

Hereafter, we will refer to the programs P_0 and P_1 as the *prototype player* programs, and the programs $\mathbf{player}_0$ and $\mathbf{player}_{R,1}$ as the **player** programs.

The next proposition informally says that $\mathbf{player}_0$ and $\mathbf{player}_{R,1}$ are correct implementations of player 0 and player 1, respectively, in the game $\Gamma(R, x_0)$.

Proposition 7.1. Let L' be the sublanguage of L whose function symbols are the unary function symbol $\mathtt{s}$ and constant $\mathtt{0}$.

1) Using predicate symbol p0, **player**$_0$ computes with respect to L' the relation P consisting of all tuples $(\mathtt{s}^x(\mathtt{0}), \mathtt{s}^z(\mathtt{0}), \mathtt{s}^w(\mathtt{0}))$.
2) Using p1, **player**$_{R,1}$ computes with respect to L' the relation Q where $Q(\mathtt{s}^x(\mathtt{0}), \mathtt{s}^z(\mathtt{0}), \mathtt{s}^{z_0}(\mathtt{0}), \mathtt{s}^w(\mathtt{0}), \mathtt{s}^u(\mathtt{0}))$ holds iff ānу of the following conditions hold: (i) $R(x,z)$ and $\neg R(x,z_0)$ and $w = u = 0$, (ii) $R(x,z)$ and $R(x,z_0)$, and $u = 0$, (iii) $\neg R(x,z)$ and $R(x,z_0)$ and $w = 1$ and $u = 0$, or (iv) $\neg R(x,z)$ and $\neg R(x,z_0)$ and $w = u = 0$.

Proof:
By proposition 4.2., it suffices to show that the prototype player programs P_0 and $P_1 \bigcup P_R$ compute the relations P and Q, given in the proposition, using p0 and p1, respectively. This is nearly immediate. □

We now show how to connect the **player** programs. This will be done by replacing the empty bodies of the *terminating clauses* in the **player** programs by calls to instances of p0 and p1 literals.

Definition 7.4. The clauses (1) - (4), below, are called *connecting* clauses.

(1) $\mathtt{stack(0,nil}, f_{\mathtt{p0}}\mathtt{(X,Z,WinLoss))} \leftarrow \mathtt{p1(X,Z,Z1,WinLoss,s(0))}.$
(2) $\mathtt{stack(0,nil}, f_{\mathtt{p0}}\mathtt{(X,Z,WinLoss))} \leftarrow \neg\mathtt{p1(X,Z,Z1,WinLoss,s(0))}.$
(3) $\mathtt{stack(0,nil}, f_{\mathtt{p1}}\mathtt{(X,Z,Z0,WinLoss,s(0)))} \leftarrow \mathtt{p0(Z0,Z1,WinLoss)}.$
(4) $\mathtt{stack(0,nil}, f_{\mathtt{p1}}\mathtt{(X,Z,Z0,WinLoss,s(0)))} \leftarrow \neg\mathtt{p0(Z0,Z1,WinLoss)}.$

Connecting clauses (1) and (3) are said to be *positive*; connecting clauses (2) and (4) are *negative*. A *connection* is any one of the four programs consisting of two connecting clauses obtained by selecting *one* of the two clauses (1) and (2) and by selecting *one* of the two clauses (3) and (4). A *game program* consists of the initializing clause, and the clauses for the **player** programs but where the terminating clauses of the **player** programs are replaced by a connection.

8. Unifying Two Theorems

In this section we show that two theorems that give the degree of unsolvability of two distinctly different classes of normal logic programs are actually two manifestations of the same underlying complexity of the dependency relations determined by the programs in these classes. This complexity is determined by lemma 6.1., above.

By a *sufficiently large* language we mean a language with at least one constant and one nonconstant function symbol and at least one binary predicate symbol or one binary function symbol. By independent means the following two theorems can be established.

Theorem 8.1. *If L is a sufficiently large language, the set of normal logic programs over L that are locally stratified is complete* Π^1_1.

Theorem 8.2. *If L is a sufficiently large language, the set of definite clause programs over L with a unique supported Herbrand model is complete* Π^1_1.

The first of these theorems is proved in [8]. The second is contained in an unpublished technical report, [4]. In this section we observe that both theorems are obtainable by essentially the same proof using lemma 6.1. The point is that the lemma is very generic, and the two theorems follow nearly immediately by the same short routine line of reasoning about game programs. We now prove both of these theorems together.

Proof:
Form two programs, Q^+ and Q^- as follows. First, choose $y \in \mathbf{N}$ and form the **player** programs using relation R_y where R_y is as in corollary 6.1. Next, connect the **player** programs by replacing their terminating clauses by a connection consisting of the positive connecting

clauses in forming Q^+ and the negative connecting clauses in forming Q^-. Include the initializing clause

$$\mathtt{start}(\mathtt{s}^y(\mathtt{0}), \mathtt{WinLoss}) \leftarrow \mathtt{p0}(\mathtt{s}^y(\mathtt{0}), \mathtt{Z0}, \mathtt{WinLoss}).$$

in Q^+ and Q^-. This completes the construction of Q^+ and Q^-. We now have the following claims.

claim 1: Q^- is locally stratified iff player 0 has a winning strategy in $\Gamma(R_y, y)$.

claim 2: Q^+ has a unique supported Herbrand model (which is empty) iff player 0 has a winning strategy in $\Gamma(R_y, y)$.

We complete the proof of the theorems by proving claims 1 and 2. We prove claim 1 first. A proof of claim 2 will then be at hand almost immediately. A program is locally stratified iff the *depends negatively on* relation is Noetherian.

The *depends negatively on* relation (with respect to Q^-) is not Noetherian

iff

there is an infinite sequence of ground atoms

$$A_0, \ldots, A_n, \ldots$$

such that A_i depends negatively on A_{i+1} for all $i \in \mathbf{N}$

iff

(see the remark immediately following the proof.)

there is a sequence

$$\begin{gathered}
\mathtt{p0}(\mathtt{s}^{k_0}(\mathtt{0}), \mathtt{s}^{k_1}(\mathtt{0}), \mathtt{s}(\mathtt{0})) \\
\mathtt{p1}(\mathtt{s}^{k_0}(\mathtt{0}), \mathtt{s}^{k_1}(\mathtt{0}), \mathtt{s}^{k'_1}(\mathtt{0}), \mathtt{s}(\mathtt{0}), \mathtt{s}(\mathtt{0})) \\
\mathtt{p0}(\mathtt{s}^{k'_1}(\mathtt{0}), \mathtt{s}^{k_2}(\mathtt{0}), \mathtt{s}(\mathtt{0})) \\
\mathtt{p1}(\mathtt{s}^{k'_1}(\mathtt{0}), \mathtt{s}^{k_2}(\mathtt{0}), \mathtt{s}^{k'_2}(\mathtt{0}), \mathtt{s}(\mathtt{0}), \mathtt{s}(\mathtt{0})) \\
\vdots \\
\mathtt{p0}(\mathtt{s}^{k'_n}(\mathtt{0}), \mathtt{s}^{k_{n+1}}(\mathtt{0}), \mathtt{s}(\mathtt{0})) \\
\mathtt{p1}(\mathtt{s}^{k'_n}(\mathtt{0}), \mathtt{s}^{k_{n+1}}(\mathtt{0}), \mathtt{s}^{k'_{n+1}}(\mathtt{0}), \mathtt{s}(\mathtt{0}), \mathtt{s}(\mathtt{0})) \\
\vdots
\end{gathered}$$

of atoms such that each atom in the sequence *depends negatively on* the succeeding atom

iff

there is an infinite sequence

$$k_0, k_1, k'_1, k_2, k'_2, k_3, \ldots, k_n, k'_n, k_{n+1}, \ldots$$

such that $R_y(k_0, k_1)$, $R_y(k_0, k'_1)$ and for each $i \in \mathbf{N}$: $R_y(k'_i, k_{i+1})$ and $R_y(k'_i, k'_{i+1})$.

iff

Player 0 does not have a winning strategy for the game $\Gamma(R_y, y)$.

This completes the proof of claim 1. To prove claim 2, replace *depends negatively on* by *depends positively on* in the above argument. The new argument goes through because the *depends positively on* relation, with respect to each of the player programs separately, is Noetherian. □

The reader may wonder whether, in the preceding proof, dependencies between, for example, atoms of the form $\mathtt{p0}(t_1, t_2, t_3)$ and $\mathtt{p1}(u_1, u_2, u_3, u_4, u_5)$ are relevant when the terms $t_1, t_2, t_3, u_1, u_2, u_3, u_4, u_5$ may contain occurrences of function and constant symbols other than $\mathtt{s}$ or $\mathtt{0}$. Such dependencies are relevant. However, by replacing every term v, where v has the form $\mathtt{s}^k(v')$ and the principal function symbol of v' is neither $\mathtt{s}$ nor $\mathtt{0}$, with v^*, where v^* is $\mathtt{s}^k(\mathtt{0})$, we obtain a dependency between $\mathtt{p0}(t_1^*, t_2^*, t_3^*)$ and $\mathtt{p1}(u_1^*, u_2^*, u_3^*, u_4^*, u_5^*)$.

9. Conclusions and Future Work

The aim of this paper has been to show the utility of viewing interacting logic program procedures as players in a game. The utility in this approach is that a variety of results about the degrees of unsolvability of logic program properties and expressive power can be established by representing various phenomena as properties of game trees. Sequences of moves in the game are represented as dependencies, and subsequently as entailments, with respect to various logic programs.

We speculate that the game tree technique presented here will also be useful in investigations of subrecursive complexity properties of function-symbol-free programs, provided the stack-machine aspect of a binary extensional equivalent of a function-symbol-free program is kept clearly separate from the relations defined by the program itself. The reduction of programs to binary programs given in this paper also suggests that properties of programs closely related to dependencies between atoms should be further investigated by studying dependencies between finite *sets* of atoms. The techniques presented here also suggest that reversing the direction of application of the games may aid future investigation of GH-games. We stress that even where considerations about games are not essential, such considerations are illuminating.

GH-games as we have used them in this paper do not require us to treat only GH-games whose winning sets are definable in terms of kernels of positions (*cf.* the comments following definition 2.1.) However, we believe that restricting to such GH-games as well as employing subsequent notions studied by Yakhnis and Yakhnis [15] will prove to be a powerful technique for investigating logic programs, computationally and semantically.

Acknowledgements

The author has benefitted from discussions with Anil Nerode, Jeffrey Remmel, Victor Marek, Alexander Yakhnis and Vladimir Yakhnis.

References

[1] Andreka H. and Nemeti I.: "The Generalized Completeness of Horn Predicate Logic as a Programming Language", *Acta Cybernetica*, **4**, 1978, 3–10.

[2] Apt, K. R.: "Logic Programming", J. van Leeuwen (ed.), *Handbook of Theoretical Computer Science*, Amsterdam: Elsevier, 1990, 494–574.

[3] Blair, H. A.: "The Recursion-Theoretic Complexity of the Semantics of Predicate Logic as a Programming Language", *Information and Control*, **54**(1), 1982, 25–47.

[4] Blair, H. A.: *Decidability in the Herbrand Base.* (Manuscript) Workshop on Deductive Databases and Logic Programming, Washington D.C., 1986, Syracuse University Logic Programming Research Group Technical Report LPRG-TR88-13.

[5] Blair, H. A.: "Canonical Conservative Extensions of Logic Program Completions". *IEEE Symposium on Logic Programming*, San Francisco, 1987, 154–161.

[6] Blair, H. A.: "Metalogic Programming and Direct Universal Computability", Abramson, H and Rogers, M. H. (eds.), *Meta-Programming in Logic Programming*, Cambridge: MIT Press, 1989, 53–63.

[7] Blair, H. A.: "Game Characterizations of Logic Program Properties", *Logic Programming and Nonmonotonic Reasoning*, Lecture Notes in Artificial Intelligence no. 928, Berlin: Springer, 1995, 99–112.

[8] Blair, H. A., Marek, V. W. and Schlipf, J. S.: "The Expressiveness of Locally Stratified Programs", *Annals of Mathematics and Artificial Intelligence*, **15**(1995), 209–229.

[9] Cholak, P. and Blair, H. A.: "The Complexity of Local Stratification", *Fundamenta Informaticae*, **21**(4), 1994, 333–344.

[10] Gurevich, Y. and Harrington, L.: "Trees, Automata and Games", *Proceedings of the 14th Annual ACM Symposium on Theory of Computing*, 1982, 60–65.

[11] Huet, G.: "Confluent Reductions: Abstract Properties and Applications to Term Rewriting Systems", *Journal of the Association for Computing Machinery* **27**(4), 1980, 797–821.

[12] Lloyd, J. W.: *Foundations of Logic Programming, (2nd. ed.)*, Berlin: Springer-Verlag, 1987.

[13] Nerode, A. and Shore, R.: *Logic for Applications*, Berlin: Springer-Verlag, 1993.

[14] Rogers, H. *Theory of Recursive Functions and Effective Computability*, New York: McGraw-Hill, 1967.

[15] Yakhnis, A. and Yakhnis, V. "Extension of Gurevich-Harrington's Restricted Memory Determinacy Theorem: A Criterion for the Winning Player and an Explicit Class of Winning Strategies", *Annals of Pure and Applied Logic*, **48**(3), 1990, 277–297.