Fast Interpreter for Logical Reasoning in General Game Playing

Maciej Świechowski
PhD Student at Systems Research Institute, Polish Academy of Sciences
Newelska 6, 01-447 Warsaw, Poland.
m.swiechowski@ibspan.waw.pl
Jacek Mańdziuk
Faculty of Mathematics and Information Science, Warsaw University of Technology
Koszykowa 75, 00-662 Warsaw, Poland.
mandziuk@mini.pw.edu.pl

Abstract

In this paper, we present an efficient construction of the Game Description Language (GDL) interpreter. GDL is a first-order logic language used in the General Game Playing (GGP) framework. Syntactically, the language is a subset of Datalog and Prolog, and like those two, is based on facts and rules. Our aim was to achieve higher execution speed than anyone's of the currently available tools, including other Prolog interpreters applied to GDL. Speed is a crucial factor of the state-space search methods used by most GGP agents, since the faster the GDL reasoner, the more game states can be evaluated in the allotted time. The cornerstone of our interpreter is the resolution tree which reflects the dependencies between rules. Our paradigm was to expedite any heavy workload to the preprocessing step in order to optimize the real-time usage. The proposed enhancements effectively maintain a balance between the time needed to build the internal data representation and the time required for data analysis during actual play. Therefore we refrain from using tree-based dictionary approaches such as TRIE to store the results of logical queries in favor of a memory-friendly linear representation and dynamic filters to reduce space complexity. Experimental results show that our interpreter outperforms the two most popular Prolog interpreters used by GGP programs: Yet Another Prolog (YAP) and ECLiPSe respectively in 22 and 26 games, out of the 28 tested. We give some insights into possible reasons for the edge of our approach over Prolog.

Keywords

First-order logic, General Game Playing, Game Description Language, Prolog, Logical Programming.

I. Introduction

Logic is applied to various aspects of Information Technology, e.g. software and hardware engineering, programming, Artificial Intelligence (AI), and other areas. In many applications, however, logical reasoning is expensive and requires highly efficient implementation. In this paper, we present an application of a Prolog-type language to the so-called General Game Playing (GGP), which is currently regarded as one of the challenges in AI. Our particular focus is on the implementation of the real-time high-performance interpreter of the Game Description Language (GDL) [13], which is used in the GGP framework.

The idea of inventing machines capable of playing games has been accompanying people for a long time before the inception of AI. With the invention of the first computers, Claude Shannon published one of the very first papers regarding computer chess programming [26]. Since 1950s computer game playing has been an important sub-field of AI, for several reasons. The first and foremost one was the temptation to build computers able to defeat humans in the most popular games. Although the early dreams of creating a "super-human" artificial player (by means of mimicking human way of playing) have not been fulfilled so far, computer programs gradually outplayed human world champions in most popular two-player board games (e.g. checkers [23], Othello [4], Backgammon [30], Scrabble [27], or chess [9]) with only few exceptions (Go being the most notable one). On the one hand, programs became stronger players than humans, but, on the other hand, there are still many aspects attributed to (human) intelligent playing, which are missing in computer playing, like intuition, multitasking (multi-game playing), universality or adaptability [14], [15], [16]. For instance, a program which can play chess at master level cannot play a simple Tic-Tac-Toe game at all. Furthermore, a specialized program cannot play any new game until its creator implements algorithms suitable for that game.

With this arguments in mind, a new trend emerged to design artificial players capable of playing virtually any game from a particular class of games at a satisfactory level. This trend is often referred to as multi-game playing or General Game Playing. The idea returns to the roots of AI since its realization requires integration of several "early-AI" concepts, e.g. incremental learning, abstract reasoning, knowledge extraction and knowledge transfer [15]. In this work we focus on a subset of multi-player, deterministic, perfect-information games which are defined in the already mentioned GDL, being a variant of Datalog [1],

[10]. Our particular interest lies in the problem of efficient interpretation of game rules and logical reasoning. With this purpose in mind we have implemented a GDL interpreter as part of our GGP player named MINI-Player [29]. As will be justified later in this paper, the speed of logical reasoning has a crucial impact on the performance of our player. We experimentally verify the efficiency of our interpreter versus EclipseProlog [2] and Yap Prolog [34] interpreters. The results are very encouraging.

The paper is organized as follows: Section II provides a formal description of GGP framework and the GDL. In Section III we present an overview of how the GGP players (and MINI-Player among them) deal with GDL-based game analysis. The next sections are devoted to details of the customized reasoning mechanism tailored for the GGP requirements. We present a description of our logic reasoner compatible with the GDL model. The reasoner runs in two phases: first, it initializes itself upon being provided with a game description (a preprocessing phase, described in Section IV) and creates the Resolution Tree (Section V). Then, the tree is queried interactively during a game (a runtime phase, described in Section VI). Section VII contains optional optimizations, which although could be omitted, significantly increase the interpreter's speed. Experimental setup and results of comparison of our interpreter with the two Prolog interpreters, very popular in GGP community, are presented in Section VIII. The last section is devoted to conclusions.

II. The use of logic in General Game Playing

GGP materializes an idea of multi-game playing, i.e. creation of computer systems able to play many games at a satisfactory level. This particular embodiment of multi-game playing was proposed by Stanford Logic Group at Stanford University in 2005 [8]. Since then, the GGP Competition is annually organized in conjunction with the AAAI Conference. The competition provides an opportunity to test the players (from all over the world) against each other in a common environment. There is a 10 000$ prize for the winner which accompanies the world champion title. In order to make programs play multiple games, a formal way of rules description needs to be defined. Contrary to specialized single-game programs, which have the rules and all additional information required to play the game encoded (so there is no need to externally provide the rules of a game), in the GGP case a game description is an input variable. A GGP program cannot have the game hard-coded as it is expected to play even previously unknown games if only they belong to a certain genre. Actually, the game might even not have existed at the time the program was created. Theoretically, game rules could be described explicitly by a state machine with nodes denoting game states and edges representing transitions between them. However, in vast majority of games such representation is infeasible. For example, in Chess there are 10^{47} states in a complete game tree (among them 10^{30} unique). Even in a relatively simple game like Connect4 [28] there are 10^{13} states. In order to tackle these constraints, a formal logic language was defined to be used in GGP. It is called Game Description Language (GDL). As introduced earlier, GDL is an official communication format and a way of rules' representation in GGP. It is a first-order logic language based on Datalog, which in turn is a subset of Prolog. The GDL has several distinguished keywords having special interpretation. There are also some other limitations compared to Prolog. In this section we present basic properties of GDL and describe its use in defining GGP games.

The class of representable games

- **Finite**. This means finite number of players, finitely many unique states and possible actions in each state and finite number of steps required to reach a terminal state. There are no fixed limits though and these values can be arbitrary high.
- **Multiplayer**. Games can feature any number of players including the case of one player games (puzzles).
- **Synchronous**. This concept means that all players move in each game step and there is one synchronous game state update after this combined move. No other updates are possible between the moves. In turn-based games, usually, there is a "noop" ("no operation") move introduced, which is the only legal move for the players not having a turn.
- **Deterministic.** There is no randomness in game state transition function (such as dice roll, for instance). Recently, a GDL-II version was proposed [31] which allows randomness but it has not been used in the competition yet.

Games can be cooperative, competitive, combination of both or of virtually any other "nature". This aspect depends solely on a game designer.

Representation of game environment

In GDL a database of terms which hold true is maintained. A complete state of a game world is defined by all facts which are true. These facts can result from three possible sources:

- **Constant facts**. Facts which are explicitly defined. Such facts can be regarded as constant laws of game physics, because they never change during a game.
- **Dynamic facts**. Special facts which are defined for the initial state (using *init* keyword) and undergo transitions based on a state update function. This is one of the main differences between GDL and (plain) Prolog which includes no state update logic. During that update a new set of dynamic facts is computed according to the rule defined in a keyword *next* (see below).
- **Results of propositions (dynamic rules).** These are facts which hold true after a successful resolution of arbitrary rules. In terms of logic, they are consequences of such rules or, equivalently, all variable bindings of that rules. GDL users

are allowed to define any number of rules unless their names collide with the keywords. Such rules can be compared to functions in classic programming languages: the inputs are conditions, the results are facts which become true (facts and rules which produce them share a common name) and, finally, these rules are computed like functions. We will use a common term - **relation** - for both rules and facts.

Keywords

Most of the keywords are used in order to define a consistent entry point for the manipulation of a game-play. Below, the meaning of the GDL built-in keywords is explained. For an example usage of these keywords please refer to the excerpt from a game description listed at the end of this section.

role($< R >$**)** - a constant fact name used for assigning roles to game participants.

init($< T >$**)** - a relation which initializes dynamic facts, i.e. defines a set of terms $< T >$ which hold true in the initial state. It is sufficient to be resolved once, before a game starts.

true($< T >$**)** - an auxiliary keyword, used only in conditions, indicating that $< T >$ is required to hold true in the current state.

next($< T >$**)** - a distinguished relation for state transition. Every term $< T >$ which satisfies the conditions is true in the next state.

legal($< R >, < A >$**)** - results of this rule resolved in the current state define actions $< A >$ available to player $< R >$.

does($< R >, < A >$**)** - a condition name meaning that player $< R >$ took action $< A >$ in the preceding state.

terminal - a rule which, when successfully proved in the current state, means that the state is terminal and the game is finished.

goal($< R >, < V >$**)** - used to assign score $< V >$ to player $< R >$. By definition, the results of this rule are only valid in terminal states. Moreover, in a terminal state, there must exist a goal value defined for each player. $< V >$ can be a number from $[0, 100]$ or a non-instantiated variable whose all possible instantiations fall into [0,100] interval.

distinct($< p >, < q >$**)** - means that two variables or a symbol and a variable are syntactically not equal. This is an auxiliary relation that simplifies game description. An opposite relation is not needed because making two variables equal is equivalent to using a common name for them.

Logic operators

not($< C >$**)** - used for negation of condition C.

or($< C_1 >, ..., < C_n >$**)** - OR (logical alternative) operator applied to conditions $< C_1 >, ..., < C_n >$

Logical AND is a default connective between conditions and there is no special symbol for it.

Non-instantiated variables

Non-instantiated variables start with a ? character, e.g. **?player**. Their interpretation does not differ from that in Prolog, so we will not go into details. Variables which share a common name within the scope of a given rule must receive the same groundings. Before the rule is resolved, non-instantiated variables basically mean 'any symbol'. Upon resolution of the rule, they have to be instantiated, i.e. a set of symbols satisfying the rule when replacing the variable needs to be found.

Game playing scenario

Each GGP game is governed by a special program (named Gamemaster) located on the GGP server. The Gamemaster as well as all the players use the provided game description in order to simulate games. The Gamemaster gathers actions chosen by the players and is responsible for the main game simulation. In the GGP framework, a game is played according to the scenario shown in Algorithm 1 (all actions are performed in the same way by both the Gamemaster and all the players; the only difference lies in (3)).

In GGP, a player must be able to perform an extensive search in the state space. Since logical rules must be resolved in each tested state, the speed of the resolution process is critical. The faster the states are computed, the greater portion of the state space is searched in the allotted amount of time and consequently the better, overall, the player is. The magnitude of the improvement partly depends on games played and search methods used.

Simple Example - A definition of Tic-Tac-Toe game

```
;;; roles
(role xplayer)
(role oplayer)

;;; board initial state
(init (cell 1 1 b))
```

ALGORITHM 1: Game playing scenario

(1) compute the initial state [using **init** rules];
repeat
(2) compute legal moves available to players [using **legal** rules];
(3) make moves;
- each player chooses their move individually;
- the Gamemaster makes moves received from all the players (via HTTP);
(a random move in case of no-response or an illegal one);
and sends all the made moves to the players;
[generate **does** facts];
(4) compute the next state;
- resolve **next** rules in the current state and save results;
- clear the current state by means of all dynamic facts;
- current state := saved results of **next** rule's resolution;
(5) check if a state is terminal [using **terminal** rule];
until *state is terminal*;
(6) compute players' scores in a terminal state. [using **goal** rules];

```
(init (cell 1 2 b))
...
(init (cell 3 2 b))
(init (cell 3 3 b))
(init (control xplayer))

;;; user defined relation for turns
(<= (next (control xplayer))
    (true (control oplayer)))

(<= (next (control oplayer))
    (true (control xplayer)))

;;; board update
(<= (next (cell ?m ?n x))
    (does xplayer (mark ?m ?n))
    (true (cell ?m ?n b)))

(<= (next (cell ?m ?n o))
    (does oplayer (mark ?m ?n))
    (true (cell ?m ?n b)))

(<= (next (cell ?m ?n ?w))
    (true (cell ?m ?n ?w))
    (distinct ?w b))

(<= (next (cell ?m ?n b))
    (does ?w (mark ?j ?k))
    (true (cell ?m ?n b))
    (or (distinct ?m ?j) (distinct ?n ?k)))

;;; patterns
(<= (row ?m ?x)
    (true (cell ?m 1 ?x))
    (true (cell ?m 2 ?x))
    (true (cell ?m 3 ?x)))
```

```
(<= (column ?n ?x)
    (true (cell 1 ?n ?x))
    (true (cell 2 ?n ?x))
    (true (cell 3 ?n ?x)))

(<= (diagonal ?x)
     (true (cell 1 1 ?x))
     (true (cell 2 2 ?x))
     (true (cell 3 3 ?x)))

(<= (diagonal ?x)
     (true (cell 1 3 ?x))
     (true (cell 2 2 ?x))
     (true (cell 3 1 ?x)))

(<= (line ?x)
    (row ?m ?x))

(<= (line ?x)
    (column ?m ?x))

(<= (line ?x)
    (diagonal ?x))

(<= open
     (true (cell ?m ?n b)))

;;; available moves
(<= (legal ?w (mark ?x ?y))
    (true (cell ?x ?y b))
    (true (control ?w)))

(<= (legal xplayer noop)
    (true (control oplayer)))

(<= (legal oplayer noop)
    (true (control xplayer)))

;;; goals and terminals
(<= (goal xplayer 100)
    (line x))

(<= (goal xplayer 50)
    (not (line x))(not (line o))(not open))

(<= (goal xplayer 0)
    (line o))

(<= (goal oplayer 100)
    (line o))

(<= (goal oplayer 50)
    (not (line x))
    (not (line o))
    (not open))

(<= (goal oplayer 0)
    (line x))
```

```
(<= terminal
    (line x))

(<= terminal
    (line o))

(<= terminal
    (not open))
```

III. GDL REASONING IN GENERAL GAME PLAYERS

In this section we briefly examine the tools used by top computer players for dealing with GDL. On a general note, there are two main approaches: Prolog-based players and non-Prolog-based ones (i.e. all the others). Majority of the strong players take the former approach. In the most common scenario the rules, written in GDL, are received via HTTP and then a direct transformation to Prolog syntax is performed. Next, the Prolog program is compiled and accessed by the player through some kind of interface (e.g. DLL or standard output). The two most popular Prolog distributions used in GGP framework are Yet Another Prolog (YAP) [34] and ECLiPSe Prolog [2], [6]. YAP is used by CadiaPlayer [7], which won the competition three times (in 2007, 2008 and 2012). Ary [18], two times winner (2009, 2010), uses YAP for competition and a slower SWI Prolog for debugging. Another example of a YAP-based reasoning GGP agent is Magician [32].

ECLiPSe Prolog, which is a choice of Centurio [19] and which is also embedded in sample players available for download from the Dresden GGP site [20], is a popular distribution, too. Another strong program - FluxPlayer [24] (the winner of the 2006 competition), uses Prolog with the so-called fluent calculus. CadiaPlayer and FluxPlayer's approaches are investigated in depth in [3]. The conclusion is that they are very close to each other in terms of speed (which depends on the efficiency of Prolog) and they both outperform all sample players from Dresden site. The examples of the second category - custom reasoning mechanisms - include GGP-Base, Toss and GaDeLaC. GGP-Base is used by a few successful players, but its reasoning performance cannot match the speed of Prolog. As stated in [3] GGP-Base is slower than YAP. One of the reasons of poorer GGP-Base performance is that it is written in Java. Toss [11] transforms game rules into a unique, specific representation, but, according to [11], the automatic transformation is difficult and not always successful. The third example - GaDeLaC [21] is a compiler from GDL to any language (OCAML was chosen in [21]). It is based on automatic generation of native language functions from GDL. A forward chaining approach to GDL was applied in [25]. We plan to contact the authors to setup a comparison of this approach with our interpreter. Another promising approach is [12] in which a reasoner of mGDL, a variant of GDL stripped of function constants, is proposed. The idea is based on generation of C++ functions from the description. The authors claim to have achieved from 28% to 7300% performance gain over YAP. Our aim is to design a top-down interpreter faster than the state-of-the-art YAP interpreter. A complementary goal is to create a strong GGP player, competitive to the top GGP agents, equipped with AI algorithms used for game playing.

MINI-Player

Our GDL interpreter is used in a GGP player named MINI-Player [29], [17], which was one of the 2012 and 2013 GGP Competitions participants. MINI-Player managed to win against some of the former champions and advance to the final stage of the competition (top-8 players). Its relatively high playing strength stems, to some extent, from the efficiency of the embedded interpreter. The system is written partly in C# .NET 4.0 and partly in C++. For the sake of presentation clarity we will restrict its description to the most pertinent aspects only. The readers interested in implementation details (e.g. the code-based description) are requested to contact the authors directly.

The motivation

Our motivation for building a specialized first-order logic GDL interpreter was twofold. Firstly, our goal was to **improve the speed of rules' analysis**. Secondly, we aimed at creating a system that would allow **integration of some additional AI logic in the process of rules computation**. This would not be possible with closed Prolog implementations unless such algorithms were written in Prolog directly. However, Prolog is difficult and inefficient when used to non-logic-based programming aspects like sockets networking, threads, I/O, pointers or floating point operations. Furthermore, the core difference between our system and Prolog lies in the way the systems attempt to prove rules. Prolog engine tries to find a resolution refutation of the negated query. It creates choice-points and tries various variable bindings. If the execution fails, all changes such as variable bindings are undone and the next choice-point is created. This strategy is somewhat similar to the one used by SAT solvers. Our system creates complete resolution trees and tries to prove rules positively (i.e. find all their groundings/variable bindings).

An overview

The usage of our interpreter differs in the two main phases of a GGP game: preprocessing and run-time. In the preprocessing stage the interpreter is responsible for GDL parsing, rules analysis and creating necessary data structures to prepare the run-time environment. Most memory allocations are made in that phase. In short, preprocessing phase can be divided into the four following steps:

- Reading game rules written in GDL from a file or received from the Gamemaster via HTTP.
- Tokenization (described in section IV-A).
- Finding intrinsic variables in definitions of rules (section IV-D).
- Creating resolution trees (OR-AND) for the keyword rules (section V).

During a run-time phase the interpreter is used for the actual game-playing. In particular, the following two main steps are repeatedly executed by the player in that phase:

- Using the resolution trees to reason about game dynamics (section VI).
- Updating the resolution tree with recurrent nodes, if needed (section VI).

IV. Preprocessing

A. Tokenization

At the beginning of the preprocessing phase, rules given as text are divided into tokens to identify terms and their parent-child dependencies. When there is more than one term between two brackets, the first one becomes a parent of all the rest. As a result of tokenization, a list of top (i.e. root) level tokens is obtained.

Root-type tokens are further divided into four main categories:

- **Role**. We count these tokens to know how many players a game has, and store them in an array.
- **Init**. A dynamic state term must follow an init keyword found at topmost level. We put such terms into a dictionary where a key is a name for the state relation (e.g. $cell$) and the value is a collection of facts which hold true in the initial state.
- $<=$. This token denotes a general rule (a conclusion). Its first child is parsed as a rule header (a conclusion consequence) whereas subsequent children are parsed as conditions.
- **Any other text**. If a term discovered at the topmost level is none of the above, it has to be a constant fact. We put such facts in a dictionary based on their type.

The other types of tokens are the following:

- **Relation name**. If the first string parsed after an opening bracket is not a keyword, it is a name of a relation. Such name can appear in the header section (definition) or in a conditions section (query for a rule/fact). Classification whether a relation is a fact or a rule is performed in two passes, because in GDL it is possible to reference a rule defined further on in the text. While conditions for dynamic facts require a true keyword, rule and constant facts conditions are indistinguishable.
- **Variable**. Has to appear as a child of a relation name or the $distinct$ keyword. Starts with "?".
- **Symbol**. Has to appear as a child of a relation name or the $distinct$ keyword. Cannot start with "?".
- **Distinct**. Has to have two children. If all of the children terms are constant, we check if they are distinct only once. In case they are, the distinct condition is removed for being always true. Otherwise, the whole rule which contains the condition is removed for being always false.
- **OR, NOT**. Possible children are: relation name, OR, NOT.
- **Does**. A special relation name used to define a condition about the last performed moves.

Furthermore, there are four reserved rule token types (subcategories of $<=$):

- **Next**. We store such tokens in a dictionary with a key being a name of $next$ fact type.
- **Legal**. We store them in a list of legal move tokens. No dictionary per role or move name is maintained.
- **Goal**. Such tokens are stored in a dictionary and assigned to the respective roles. If a player name is a variable, one copy of a token is assigned to each role.
- **Terminal**. Such tokens are stored in one list devoted to terminal tokens.

During a tokenization process, every symbol is converted to a 2-Byte number and stored in a symbol-number number-symbol dictionary. The aim is to hash all the strings and perform computations on integers exclusively. Operations on numbers are much faster than operations on strings, length independent and more cache friendly. The opposite conversion from a number to the original text is required for the communication with the Gamemaster and for logging/testing purposes. At first, we thought of using 1-Byte per symbol for even better memory efficiency but that would limit the number of possible symbols to 255 only. Since many games use more that hat number, we stack to 2-Byte representation which results in a safe limit of 65535 unique symbols.

B. Negation normalization

Before creation of a resolution tree (a structure used for rules proving) the rules are simplified and normalized in the following way. Negations of alternatives are transformed into conjunctions of negations. For example, NOT(OR(a,b,c)) is changed into NOT(a), NOT(b), NOT(c). In addition, unnecessary embedded negations such as NOT(NOT(...)) are trimmed down so here can be only one or none negations. The normalized form of negation assumes that the NOT keyword has only one condition as an argument. This form is required to simplify the proving formula and reduce the number of possible cases to deal with.

C. Arity normalization

Each relation in GDL must have a constant arity. A plain argument is a variable or an object symbol. However, complex arguments are also possible. For example, the arity of each of the following rules is equal to 3, though the second argument can be either plain (case a) or complex (case b):

```
a)  (composer Frederic Chopin)
b)  (composer (Wolfgang Amadeus) Mozart)
```

In the normalization step, we reduce complex arguments to simple ones by expansion. Removal of all complex arguments allows us to store all the data associated to the grounded rules in the form of a flat array in a memory. Complex arguments can be recurrently nested, which resembles a tree-like dependency difficult to represent linearly in a general case. We need to remove all nested arguments, therefore, the arity normalization is not an optimization but a requirement for the method to work. In order to do this, we first find the maximum arity of each argument among all its occurrences in definitions and conditions. We organize them into domains. A domain consists of a set of indexes of relations together with their maximal arity. For instance, $[\{legal[2], does[2]\}\{arity = 2\}]$ means that the second arguments in **legal** and **does** relations belong to the same domain and the maximum arity of those arguments is two.

Arity detection procedure:

- Iterate over each argument of rule headers and conditions and assign them to domains, e.g. $legal[2]$. If a domain already exists, its arity is updated to max(argument arity, current domain arity).
- If there is a common variable name shared by more than one argument, its associated domains are joint together. As a result a new domain is created which consists of a sum of relation[index] sets and the maximum arity over all domains being joint.

Unnesting procedure:

- Here we go through all arguments which belong to domains of arity greater than one.
- If we encounter an argument of arity smaller than the domain arity, N, we perform an expansion.
- If the expanded argument is a variable of arity 1, then each occurrence of the argument is split into N variables. For example, in the case of N = 3, $?player$ becomes $?player1$, $?player2$, $?player3$. The new names are generated in such a way that they don't collide with any of the existing names.
- If the expanded argument is a symbol or a variable of arity > 1 and $<$ N it is complemented by adding appropriate number of a reserved "empty" symbol, with the number representation of -1, to make the arity equal to N.

Repeating:

- If there are still complex arguments left, the whole algorithm is repeated from the "Arity detection procedure".

The example given at the beginning of this subsection after correction looks as follows:

```
a)  (composer Frederic 'empty' Chopin)
b)  (composer Wolfgang Amadeus Mozart)
```

Although the presented algorithm may seem to be computationally expensive, it is not, in fact. All operations on rules are quick compared to the actual reasoning made later. The whole preprocessing phase (including arity normalization) takes approximately 3 seconds for Chess and 2 seconds for Othello with these two games being, by far, the most costly to analyze. For the rest of the games it takes less than a second on an average consumer-quality PC computer.

Can the procedure be applied to any GDL description?

The necessary and sufficient condition for this transformation to work is that depth of complexity of each argument has to be bounded before run-time. This means that there cannot exist rules which let arguments "grow" recursively such as:

```
(<= (next (integer (succ ?x)))
          (true (integer ?x)))
```

The above rule could lead in 3 state update steps to the following result:

```
(true(integer (succ (succ (succ 1)))))
```

There is an excerpt from the GDL definition stating that: *functional terms cannot grow to unbounded size by use of recursive rules, since terms can be passed along a recursive cycle without adding function symbols, and new terms introduced into recursive cycles must be grounded by a finite set of base relation sentences.* If we treat the $next$ rule, which has the same relation name in the header and a condition, as an implicit recursive rule (and an analogous implicite rule from legal to does) then the presented example is not a valid GDL description. If we **can assume** that we are dealing with GDL descriptions in which arguments cannot grow recursively, then the problem is solved. If we **have to assume** that we are dealing with GDL descriptions in which arguments can grow recursively and the usage requires 100% correctness, then the solution would be to load a description and check whether a statically unbounded argument occur. If it does, revert back to Prolog. For practical purposes it is rarely an issue, because we have not seen any GDL game which does not fulfill this condition.

D. Intrinsic variables detection

For each rule we detect the **intrinsic variables**, i.e. the ones which appear either in the header section (conclusion) or in at least two conditions. Other variables mean nothing else but symbols. The bindings for the intrinsic variables are stored in memory because they have to be returned back in the calling hierarchy (rule header) or matched among themselves (in rule conditions).

Example 1 with intrinsic variables = {?x}.

```
(<= (line ?x)
    (row ?m ?x))
```

Example 2 with intrinsic variables = {?player}.

```
(<= terminal
    (true (control ?player))
    (stuck ?player))
```

V. RESOLUTION TREE

The ultimate products of a preprocessing step are resolution trees. Once created, they are used at run-time to prove propositions, i.e. to find facts which hold true as a result of the respective rules. We devote to them a separate section since they form a back-bone of the proposed solution and connect the preprocessing phase with the interpretation phase. A resolution tree can be regarded as a proving structure for a rule represented in its root node. We create one tree for the **legal** relation, one tree for the **terminal** relation, N trees for the **goal** relations, where N is the number of players and M trees for the **next** relations, where M is the number of dynamic states.

For legal moves, there are two equivalent choices: to create one tree per player (i.e., a constraint for a given player role is built-in in the tree) or to create one tree per each legal relation and filter the results aposteriori (our approach). The latter method is usually much faster when there are common rules for players, i.e. player's name is a variable. For example, single player rules:

```
(<= (goal xplayer 100)(line x))
(<= (goal yplayer 100)(line y))
```

is often defined in a shared rule:

```
(<= (goal ?player 100)(line ?player))
```

An instance of a rule (one $<=$ proposition) in GDL is represented by a set of conditions which imply some consequences. The consequence of a proposition is a set of facts which become true upon successful resolution. In order to obtain a complete set of these true facts one needs to resolve all rules with a common name. Let us return to the Tic-Tac-Toe example. There are three instances of ***line*** rule: a line can be a ***row***, a ***column*** or a ***diagonal***. In order to move from implication to equivalence, the results of all rule instances must be logically summed. We can distinguish a concrete rule instance definition from a query for

the rule. A query always requires the complete set of results from all instances. Based on the above observation, in our GDL interpreter, we distinguish two types of tree nodes.

- AND node - is a concrete single rule realization. It contains a header (a consequence) and conditions. The default logical connective between the conditions is AND. The purpose of AND nodes is to call conditions, merge their results and match the variables. Generally, in AND nodes mutual dependencies among conditions are resolved.
- OR node - is a container for all true facts in a relation. If a relation is a rule type, OR represents a family of rules., i.e. a collection of all rule implications (for instance, the three line implications in Tic-Tac-Toe). In this node, the results from all implications are collected. If a relation is a constant/dynamic fact type, the node stores the results directly, which can be regarded as already derived rule. **Does** fact, i.e. actions taken by the players at the preceding turn is an OR node too, for the sake of consistency.

The root is always an OR node. Nodes of OR and AND types appear alternately. Each AND node is a child of an OR node and vice versa (with the exception of the rooted OR nodes). The idea is presented in Figure 1.

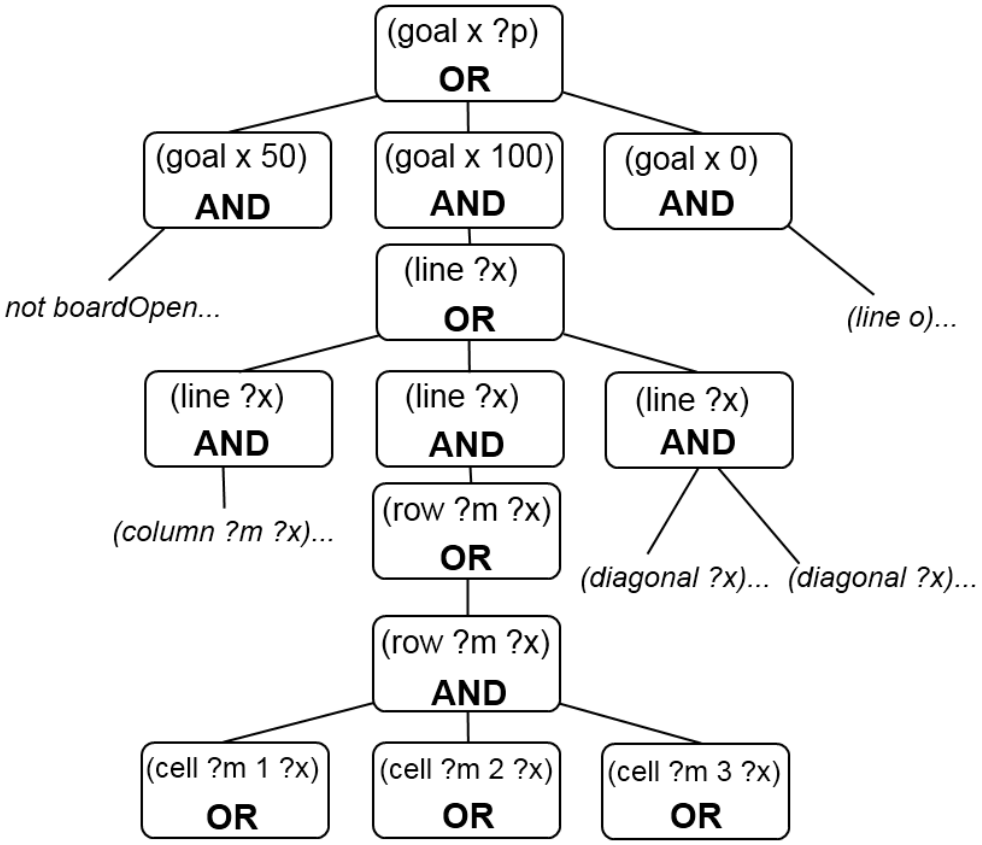

Fig. 1. The OR-AND tree created for the goal rule in Tic-Tac-Toe game. Due to the space limits, only one branch is fully shown. The remaining branches are expanded analogically. AND nodes are concrete rule instances. OR nodes can be regarded as collectors of results.

At the beginning, a global access OR nodes are created for each dynamic or constant fact. They are to be referenced inside a tree. In order to join AND and OR nodes we use an auxiliary abstract structure and some kind of merge operation. Merge operation keeps a reference to the OR node with additional information how the results should be combined with the results obtained so far. More information on merge operations is presented in the following subsection.

We designed a dedicated data structure called a RowSet for storing the results in OR/AND nodes. A word row was borrowed from database theory and will be used interchangeably with record. A row is a vector of arguments of a relation, for example

```
(cell 1 1 b)      ; row = [1 1 b]
(control xplayer) ; row = [xplayer]
```

The RowSet is essentially a continuous block of memory accompanied by a number of functions and properties. The memory is handled by a pointer to 2-Byte number (since this is a basic type for symbols). There are a few control fields that we use in

this context: $index$ - is the current number of records stored, $capacity$ - is the current maximum number of allowed rows with no need for reallocation, $stride$ - is an offset (in bytes) between consecutive records (it is usually equal to twice the arity). An example of a RowSet is presented in Figure 2.

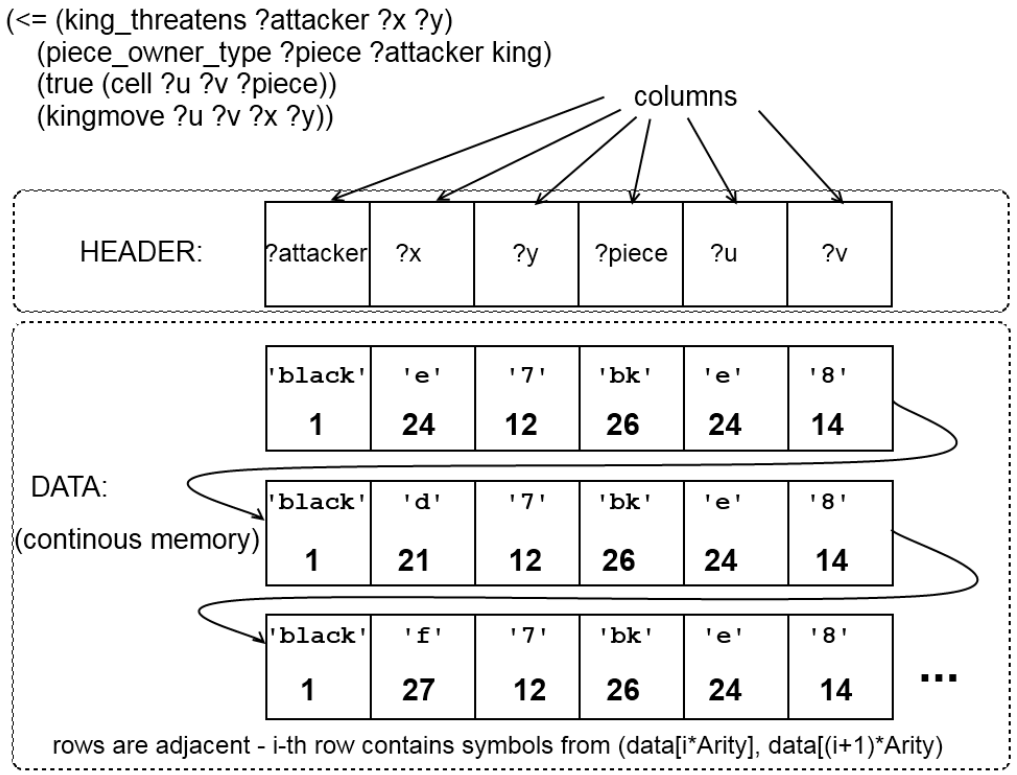

Fig. 2. An example of a RowSet. Each row is a literal with symbols. Rows are stored in memory in a continuous manner (with no physical separation). Cells contain 2-Byte numbers, but for the sake of clarity we also included the dictionary entries they represent. In this example: arity = 5, stride = 10 (arity * 2 Bytes), index = 3 (a number of valid records stored). The total length is equal to 30 Bytes.

The collection is dynamically expanded and reallocated when the expected number of rows would potentially exceed the current capacity. A RowSet contains a pointer to its current capacity. Possible capacities are discretized in order to minimize the number of similar reallocations and to allow memory precaching. We keep a few pre-allocated blocks based on predictions. Deletion of rows is not performed explicitly nor is it executed immediately. There is a list of indices of the deleted rows maintained by the interpreter. When a row eventually has to be deleted, e.g. after a successfully finished proving procedure, its content is switched with the last non-deleted row and the total number of rows (their $index$) is decremented. The switching is avoided when all erased rows are already at the end. A deletion procedure is applied starting from the highest indices so as to make the greatest use of this property. Also, it is trivial to clear the whole RowSet since it requires a single operation only: $index = 0$. This is one of the reasons why a RowSet is an efficient dynamic structure with good memory utilization. Moreover, there are simple logical operations which allow rewriting all rows from one RowSet to another. These operations benefit from a linear representation in memory and copy the entire fragment in one efficient call. There is no specific deletion of columns other than the one resulting from clearing all the rows. Columns, as presented in Figure 2, are associated with variables via storing their groundings. Since the rule conditions are computed in a fixed order, columns that contain the current data are also fixed after each consecutive condition. When a condition is being analyzed, there can be some variables which have not been introduced yet because they appeared later on. Columns associated with such variables are ignored until they are ready to be overwritten, since they may contain some data from previous calls. The system maintains information which columns are filled with symbols from the current rule computation and which ones are not (instead of erasing them). The same is true for symbols stored beyond the $index$ (i.e. last row's offset). Clearing such symbols would take time, so we rather ignore them (they may be overridden by the current data at some point).

ALGORITHM 2: Function: OR.Prove()

```
boolean Result := FALSE;
if a reference to the ground-facts-NodeOR exists then
    /*constant or state facts*/;
    copy results from the referenced NodeOR ;
    Result := (The number of results > 0)
end
foreach child AND node: n do  if n.Prove() == TRUE then
    Result := TRUE;
    take results from n;
    add them to the current results without duplicates;
end
analyze query in the current OR node /*see: Subsection VI-C. */;
filter results /*see: Subsection VI-D.*/;
return Result;
```

ALGORITHM 3: Function: AND.Prove()

```
foreach child pair< MergeOperation, OR > do
    if OR is recurrent and not expanded yet then
        OR.Expand();
    end
    if OR.Prove() == FALSE then
        return FALSE ;
    end
    /*merging the obtained results into AND RowSet; see: Subsection VI-B.*/;
    Merge(this.RowSet, OR.RowSet, OR.Filters);
    FilterResults() /*see: Subsection VI-D*/;
    if resultsCount == 0 then
        return FALSE;
    end
    return TRUE;
end
```

VI. Interpretation

A. Collecting results from proven nodes

Once a resolution tree is created in can be used for rules proving. In this section, we show a top view of how this operation is performed. Generally speaking, we define a recurrent Prove() function available both in OR and AND nodes. Three cases are considered:

- Leaf OR nodes return their results directly.
- Non-leaf OR nodes call Prove() method for each of their descendants (AND nodes) and combine their results into one set with duplicates removal. The function returns failure if all of the child Prove() calls return failure.
- AND nodes call Prove() on their descendant OR nodes and merge their results. The merging operation is elaborated on in subsection VI-B. The function returns failure immediately when any child call returns failure.

AND/OR nodes use a dedicated RowSet structure for storing the results. The pseudocode of Algorithm 2 and Algorithm VI-B for OR and AND nodes, respectively summarizes the procedure of results collection.

B. Merge operation

Each AND node (which represents an implication) contains a RowSet for storing the results. At the beginning, the RowSet is empty and none of the variables is initialized. Conditions of the implication are resolved one after another. Merge operation defines how bindings of variables used in a condition are combined with all the current bindings stored in the RowSet. Such operations are performed during execution of AND.Prove() as presented in the pseudocode of Algorithm . In general we consider three types of merge operation: **single merge**, **complex (OR) merge** and **distinct merge**.

Single merge operation

TABLE I. THE TYPES OF MERGE OPERATIONS

	New Variables	Common Variables	Total Variables Initialized Before *(by previous rule's conditions)*
New Merge	> 0	$= 0$	$= 0$
True Merge	> 0	> 0	> 0
Combine Merge	> 0	$= 0$	> 0
Verify Merge	$= 0$	> 0	> 0
Check Merge	$= 0$	$= 0$	≥ 0
There are two special cases for negated conditions for which merge formula is inverted: *(such conditions cannot initialize new variables due to restrictions imposed on negation in GDL specification)*			
VerifyNOT Merge	$= 0$	> 0	> 0
CheckNOT Merge	$= 0$	$= 0$	≥ 0

There are seven scenarios for merging new results with the existing ones. The use of particular merge type depends on how many variables are being initialized by the condition, how many variables used by the condition have already been initialized and whether there are any variables which have been initialized so far (by other conditions). Possible scenarios for merge operation are summarized in Table I. New Variables are the ones that become initialized for the first time by the condition related to the current merge operation. Common variables are those present in the current condition, which have already been initialized by some other conditions. The last column refers to all variables having been initialized (by other conditions) before the merge operation.

New Merge - uses only variables which were not initialized before. There can be only one such operation per rule instance (AND). Usually, it is the first operation unless any of the previous operations used any intrinsic variables (c.f. subsection IV-D). Technically, the operation consists in copying entire columns from the related proved OR node condition to the AND RowSet.

True Merge - in this operation some variables are new whereas some other have already been initialized, so the results should be matched. For each current record in AND RowSet an intersection with new records from OR RowSet is found. If it is non-empty, an existing record is completed with new variables. If there exists more than one completion for the same intersection, the current record is cloned for every additional completion.

Combine Merge - is applied when there exist some initialized variables, but the current operation does not use anyone of them. Additionally, the operation initializes at least one new variable. In such a case, a Cartesian product of the results is computed. Assuming that N is the number of records before applying the operation and M is the number of new results from OR node, the upper bound of the total number of records after this operation is $N \cdot M$. Typically, due to additional filtering mechanisms, either $N = 1$ or $M = 1$.

Verify Merge - in this case merge operates only on initialized variables. It iterates over the current records and verifies if they have a non-empty intersection with new results. Rows which do not have this property are deleted. The number of records never grows.

Check Merge - is the simplest case. This operation does not use any intrinsic variables at all. The procedure consists in checking if there exists at least one new result with no variable matching (if the condition arity is positive) or just passing the result from the Prove (if the condition arity is equal to zero).

VerifyNOT, CheckNOT Merge - are operations analogous to their positive counterparts with reverted conditions of success.

In the following example six relations are considered:

```
(<= (legal ?player (move ?king ?u ?v ?x ?y))
[1] (true (control ?player))
[2] (piece_owner_type ?king ?player king)
[3] (true (cell ?u ?v ?king))
[4] (kingmove ?u ?v ?x ?y)
[5] (occupied_by_opponent_or_blank ?x ?y ?player)
[6] (not (threatened ?player ?x ?y ?u ?v)))
```

and for each of them the appropriate merge mechanism is presented in Table II.

OR Merge operation

OR nodes which are parts of resolution trees should not be confused with OR operations. OR merge operation models the OR keyword and is in the form of a list of single merge operations explained in the previous paragraphs. Single merge operations which are part of the complex OR operation are performed independently, so their results do not affect each other. Technically, each operation works with a local copy of the RowSet. After the operations are computed, the local RowSets are combined into one by including all rows from the copies. Duplicates are discarded. The whole operation succeeds if at least one of its elementary operations succeeds.

TABLE II. EXAMPLE RULE AND ITS MERGE OPERATIONS

No	New variables	Common variables	Number of variables initialized before	Operation type
1	?player	-	0	New Merge
2	?king	?player	1	True Merge
3	?u ?v	?king	2	True Merge
4	?x ?y	?u ?v	4	True Merge
5	-	?x ?y ?player	6	Verify Merge
6	-	?player ?x ?y ?u ?v	6	VerifyNOT Merge

Distinct Merge operation

This operation is induced by the **distinct** keyword. This is simply a constraint that a variable cannot be instantiated to a specified symbol or two variables cannot be instantiated to the same symbol in one row. In both cases we iterate over the records stored. In the former, we delete records if they contain a banned symbol in the corresponding column. In the latter, we check symbols in two columns corresponding to the arguments of **distinct**. If their instantiations are the same, the row is deleted.

C. Variable constraints: queries

By definition, each instance of a given rule implication has the same (fixed) arity, but the proportion of variables and constants may vary. An argument can be a constant in one rule instance and a variable in another. Additionally, some implications may have constraints implied by duplicated variables in the header. As a consequence, the results stored in AND nodes may have various forms in terms of an order and a number of arguments used. The OR nodes, on the other hand, store results which directly reflect Prolog propositions, i.e. contain all symbols and in the correct order.

A custom made class **Query** is used mainly to address the two following issues. First of all, it maps columns (variables) from AND nodes to columns in OR nodes. We run the query resolution just before the data from AND node is returned (see Algorithm 2). The query is created for each (OR, AND) pair. The second issue, a **Query** is responsible for, is to check, once the results have already been rewritten to the OR-form, if they satisfy constraints (if there are any) imposed by the condition on this particular OR node. Such constraints can be twofold:

- the i-th argument is equal to a symbol S,
- the i-th argument is equal to the j-th argument.

The query is prepared in the preprocessing phase along with the tree construction and used at runtime. We detect whether constraints present in the OR node condition contradict those in AND node header, e.g. an argument is equal to two distinct symbols. If they do, we do not create the AND node thus avoiding the whole branch in the resolution tree that would, otherwise, be started by that node.

D. Filters

It is common in GDL that rules use variables shared among many terms. By definition, those terms must unify on all of the shared variables. Such variables, once instantiatied, can be used to limit the possible instantiations of the variables which are yet to be tested. It is possible in our approach, because as shown at the Algorithm 3 and in section VI-B we collect results from conditions one after another. A mechanism for this limiting behaviour is called filtering. Filters are aimed at reducing the number of intermediate results as soon as possible i.e. whenever a condition uses a variable which will be eventually unified with an already used variable. In most games, reducing the number of temporary facts make the algorithm much faster. However, it is not just an optimization, because in certain cases of recurrence, this mechanism is required for the recurrence to stop and therefore it is required for the algorithm correctness.

In order to implement this idea, each variable should be traced along the path of resolution in the tree. We identify the AND node in which a variable is instantiated for the first time. We call this variable/node a filtering one. By going deeper in the rule-condition hierarchy we find all ocurrences of variables which have to be unified with the filtering one. We call them filtered ones.

In general, a filter contains

- a pointer to the source RowSet, which always belongs to AND node,
- a distance to the source AND node in the resolution tree (this seems to be redundant but is required for the case of recursion),
- a mapping between current RowSet columns and source RowSet columns for the filtered variables.

At runtime, inside OR.Prove() or AND.Prove(), the current results are filtered out to only those, for which there exists at least one row in the source RowSet having the same variables on the mapped columns. Note, that constraints for variables should not

be applied separately. For instance, a filter: [?x ?y] = [1 2], [5 6] is much stronger than a set of two filters [?x] = [1, 5] and [?y] = [2, 6]. The latter one potentially allows receiving quadratically as many results, i.e. [1,2] [1,6], [5,2], [5,6].

Filters are used in AND nodes after a successful merge operation and in OR nodes after the resolution of query, before returning the results. We use extensive mechanism for detecting which filers are required (or useful) and which can be omitted. This selection is rather technical and based on the three principles. Firstly, filters in AND nodes which are not stronger than a merge query are omitted. Secondly, filters in OR nodes which are not stronger than any of the child AND nodes' filters are omitted. The filtering is an optional optimization for games without recurrence. Any recurrent call, in order to terminate, must at some point reach a rule which ends with a failure therefore stopping the process. The terminability can be guaranteed in many ways, one of them being dependent on the variable bindings performed along the path of resolution. Usually, the possible set of bindings shrinks with each recurrent call. Without filters, there is a risk of falling into an infinite loop. Such effect can be observed in Othello. In general, filters play an important role in our method.

E. Recurrence - deferred approach

We do not create recurrent OR nodes in the preprocessing phase, because the process would fall into an infinite loop. Therefore during the preprocessing, we only create a proper merge operation with a null pointer to the OR node. The whole data required to recreate this node (the related condition and the current filters) is passed from higher branches. The OR node is created at run-time when the **Prove** function visits that node (see the first line of Algorithm 3). The creation of such nodes is based on the same scheme as in the case of regular OR nodes in the preprocessing. Once added to a tree, they are used in the same way as other nodes. It should be noticed that a new node needs to be created, if and only if, the recurrence reaches a depth which was never reached before (this value usually stabilizes after the first several simulations).

An example of a simple recurrent definition is presented below:

```
(<= (greater ?a ?b)
    (distinct ?a ?b)
    (succ ?c ?a)
    (greater ?c ?b))
```

VII. Optional optimizations

A. Rows hashing in a RowSet

On demand, a RowSet can also hash its rows in terms of selected columns. Numbers stored in that columns are the hash function arguments. The hashing is useful for rarely changing but frequently accessed data. Selection of columns included in a hash function is made based on the most frequent query. The hashing function is based on a tree idea. Each node stores a symbol and the first and the last indices of a row containing that symbol in a particular column used by the query. The root of the tree is related to the first column taking part in the query. There are as many roots as there are unique symbols in the first column, so there are effectively multiple trees. Let us assume that a node contains a symbol S. The children of the node are related to the next column and represent symbols (in that column) which appear together with S. With such a data structure, in order to check whether a particular combination of symbols is present in a RowSet we start from the first symbol and get the first and last rows indices matching this symbol. If such a symbol does not exist in the tree or the first index is greater than the last index we already know that there are no results matching our query. Otherwise, we continue traversing the tree applying the **max** operator on the first indices and the **min** on the last indices. This way, the search space is reduced. Hashing is performed for constant and dynamic facts. It is an optional optimization, i.e. the interpreter operates with a complete functionality without hashing. Please refer to the fifth column in Table IV, which presents contribution of various enhancements, for the speed-up obtained with the use of hashing.

B. Removal of redundant relations

A GDL description may contain rules which are superfluous. A rule which depends on exactly one other rule and has one implication only (so the dependency is simply a 1-1 mapping) is considered redundant, because it can be replaced by the rule it depends on. Simply, each positive condition which uses the redundant rule is replaced by a condition which refers to that other rule. Consider, for example, the following set of rules:

```
(<= (goal ?player 10)
    open
    (true (cell ?m ?n ?player)))

(<= open
    (true (cell ?m ?n b)))
```

In the $goal$ rule there is a condition which points to the $open$ rule consisting of one condition only (let's assume there are no other $open$ rules). That condition can be simply inserted instead of the $open$ condition, but variable names must be changed so as not to collide with the existing ones. The $?n$ and $?m$ names for variables are already used in the $(true\ (cell\ ?m\ ?n\ ?player))$ condition, so keeping them after the replacement of $open$ by the $(true\ (cell\ ?m\ ?n\ b))$ condition would affect the syntax of a GDL program.

Hence, the transformed set of rules is of the following form:

```
(<= (goal ?player 10)
    (true (cell ?m' ?n' b))
    (true (cell ?m ?n ?player)))
```

The motivation is to reduce the size of a tree and, as a result, the number of performed operations.

C. Removal of unused filters

At run-time, we keep statistics of filters efficiency by counting the number of rows effectively filtered out in each node. If this number is still equal to zero after N visits to a node which contains the filter (in our case $N = 200$), the filter is removed from the node, so as to reduce computation time. We measured how many filters are removed at runtime due to the above described verification mechanism and the magnitude of speed-up accomplished thanks to this dynamic filter removal mechanism. Please refer to Table IV in Section VIII for these statistics.

D. Recurrence tabling

Please note that when there are many instances of recurrent rules and a depth of a typical call is moderately high, the memory consumption becomes a serious issue. In one of the 28 tested games, the available memory limit was reached after just a few seconds. After some tweaking we decided to store each recurrent OR node in a hashmap created for its respective rule. The hash function takes into account the entire context stored for the recurrence expansion. Nodes also contain a boolean variable indicating whether or not they are currently used by Prove method. On the top-most level we divide nodes by the ID of the related condition used to create them. Inside a bucket for the condition we only need to group nodes by their passed filters from higher tree branches. We use a hashmap from the standard language library which allows us not to care about collisions (different objects having the same hash code). The hash function for the passed filters is a (unweighted) sum of hash functions defined for each filter. A hash function for filter is computed as a weighted sum of five components:

$$\textit{FilterHash} = D + M[0].\textit{From} + 16 * M[0].\textit{To} + 32 * M[|M| - 1].\textit{From} + 64 * M[|M - 1|].\textit{To}$$

where D is a distance from the current (filtered) node up to the source (filtering) node; $M[0].From$ and $M[0].To$ denote mapping of the first column from the filtering set onto the filtered set; $M[|M| - 1].From$ and $M[|M| - 1].To$ denote the respective mapping of the last column. If a new node is to be created, first the corresponding hashmap is searched for an idle node of this type. If the quest is successful, that node is reused. If a node is reused, the source (filtering) nodes for all filters in the reused node and its children are corrected based on the distances stored in filters. For example, if a distance equals 2, the source node becomes a grandparent of the current node. This correction is needed, because nodes can be reused anywhere in the tree, if suitable. As soon as the interpretation in a given node is completed, the node is pushed back to the hashmap of idle nodes. The reusing mechanism significantly reduced the memory usage in every game featuring recurrence. In Othello, for instance, the improvement was from over 2GB to 50MB (i.e. by a factor of 40). Out of memory exceptions do not appear any longer.

VIII. Experimental results

In this work the following two objectives were intended to be accomplished:

- speeding-up the computation of rules in comparison to Prolog,
- using the interpreter as part of a computationally efficient GGP contestant.

The degree of fulfillment of the first objective significantly exceeded our expectations. Our interpreter was compared with the two most popular (amongst GGP agents) Prolog interpreters using the same computing facilities (the same PC) and on the same set of GGP games. Many of these games were played during the GGP competitions. The remaining ones were chosen blindly. All games were downloaded from the GGP games repositories [5], [22]. Table III presents the results.

Eight Puzzle and Knight Tour are puzzles. ChineseCheckers, Farmers and Pacman are three-player games.The rest of the games are two-player ones. Some games, like Bidding Tic-Tac-Toe, Bomberman, Chinook, Farmers, Fire Sheep and Logistics are truly simultaneous. An average branching factor varies between games approximately from 5 (Tic-Tac-Toe) to over 1700 (Famers) whereas an average playout length extends from 7 (Tic-Tac-Toe) to over 100 (Checkers, Pilgrimage). Random games (simulations) tend to last longer when a terminal condition is difficult to reach. In general, the results are very promising. The

TABLE III. A COMPARISON OF EFFICIENCY BETWEEN ECLIPSE PROLOG, YAP PROLOG AND OUR INTERPRETER. ALL TESTS WERE RUN ON THE SAME MACHINE CONFIGURATION USING A SINGLE THREAD. THE NUMBERS IN THE TWO RIGHTMOST COLUMNS DENOTE THE FACTOR (IN PER CENT) OF SPEED ADVANTAGE OF OUR INTERPRETER IN PARTICULAR GAMES OVER ECLIPSE AND YAP PROLOGS, RESPECTIVELY. A VALUE OF 100% DENOTES THE SAME PERFORMANCE LEVEL.

The average number of complete random simulations performed in 20 seconds.					
Game	**ECLiPSE Prolog (A)**	**YAP Prolog (B)**	**Our Solution C**	**C/A *100%**	**C/B *100%**
Bidding Tic-Tac-Toe	10333	9196	18814	**182%**	**205%**
Bomberman	13528	6602	41791	**309%**	**633%**
Breakthrough	753	904	1459	**194%**	**161%**
Cephalopod Micro	66	54	118	**180%**	**218%**
Checkers	183	198	204	**111%**	**103%**
Chinese Checkers	1079	1703	2764	**256%**	**162%**
Chinook	268	602	1200	**448%**	**199%**
Connect-4	5689	6913	11562	**203%**	**167%**
Connect-4 Suicide	5535	6813	11435	**207%**	**168%**
Dual Connect-4	3099	5741	7587	**245%**	**132%**
Eight Puzzle	5570	3850	24577	**441%**	**638%**
Farmers	4914	11430	6725	**137%**	59%
Farming Quandries	868	1901	6141	**707%**	**323%**
Fire Sheep	176	1614	2253	**1284%**	**140%**
Knight Through	1636	1620	2542	**155%**	**157%**
Knight Tour	153292	63852	189616	**123%**	**297%**
Logistics	2451	3721	11822	**482%**	**318%**
Nine Board Tic-Tac-Toe	2011	3366	6439	**320%**	**191%**
Othello	9	71	32	**341%**	45%
Pacman	1143	812	3469	**304%**	**427%**
Pentago	950	1571	868	91%	55%
Pentago Suicide	920	1577	845	92%	54%
Pilgrimage	316	983	624	**197%**	63%
Platform Jumpers	11	188	906	**8608%**	**482%**
Quarto	2979	4465	6523	**219%**	**146%**
Sheep and Wolf	1753	991	5080	**290%**	**512%**
Tic-Tac-Toe	51295	28057	109989	**214%**	**392%**
Zhadu	238	852	365	**154%**	43%
Number of places 1st/2nd/3rd	**0 - 9 - 19**	**6 - 15 - 7**	**22 - 4 - 2**		

interpreter is faster than YAP and significantly faster than Eclipse. The only game, in which our program is worse than both YAP and Eclipse is Pentago and its suicide counterpart - Pentago Suicide. YAP is also faster in Farmers, Othello, Pilgrimage and Zhadu, so there are still opportunities for improvement of our interpreter.

In addition, we also measured the contribution of particular optimizations to the overall magnitude of speed-up. Table IV contains data on how much faster the interpreter performs with a particular optimization being switched on versus being switched off. We included filters (the first column) due to the fact that, although they are needed for correctness, in most of the games they function as an optimization mechanism. Out of the tested games, only in Othello they are required for the algorithm to work. It turns out that in two games, Pentago and Pentago Suicide, the removal of the potentially unnecessary filters slows down the process (by 13% and 14%, respectively, as shown in the table). Since we remove filters which have not successfully filtered any results in the first 200 visits, the only explanation of this fact is that some of the removed filters in Pentago and Pentago Suicide started to filter the results later. The overall performance is higher if no filters are removed in these two games. A more sophisticated heuristic for detection of the filters which should be removed is one of the future goals. Removal of the redundant filters is an optional optimization, not required for the method to work. In general, all proposed enhancements proved useful and for some games they provide the increase in performance of orders of magnitude. Only removal of redundant relations seems to have just a tiny positive impact but at the same time it is an easy optimization to apply.

Our second goal has also been fulfilled although there is still room for potential improvement in this matter. MINI-Player built on top of the interpreter features an algorithm for computation of partial satisfaction of a goal condition. This formula is an extended version on the regular proving formula. A degree of goal satisfiability guides one of the simulation strategies. More

TABLE IV. RELATIVE EFFICIENCY OF VARIOUS OPTIMIZATIONS.

Game	% of the overall speed-up due to filtering	% of the removed filters (dynamically)	% of the speed-up due to the removal mechanism	% of the speed-up due to hashing of constant/state facts	% of the speed-up due to redundant conditions removal
Bidding Tic-Tac-Toe	400%	57%	2%	1%	0%
Bomberman	31%	20%	3%	24%	1%
Breakthrough	549%	43%	24%	1693%	3%
Cephalopod Micro	28%	35%	21%	-3%	0%
Checkers	58%	14%	42%	80%	0%
Chinese Checkers	29%	37%	40%	17%	3%
Chinook	10%	8%	41%	0%	2%
Connect-4	5%	3%	3%	19%	2%
Connect-4 Suicide	5%	3%	0%	19%	2%
Dual Connect-4	3%	4%	10%	10%	1%
Eight Puzzle	0%	0%	0%	0%	0%
Farmers	100%	82%	41%	824%	0%
Farming Quandries	10%	44%	20%	5%	0%
Fire Sheep	14%	4%	9%	5%	0%
Knight Through	8415%	41%	34%	814%	0%
Knight Tour	0%	44%	32%	13%	0%
Logistics	0%	14%	17%	62%	0%
Nine Board Tic-Tac-Toe	13%	23%	3%	9%	0%
Othello	REQUIRED	43%	5%	197%	1%
Pacman	510%	31%	2%	6912%	0%
Pentago	14%	21%	-14%	0%	0%
Pentago Suicide	13%	21%	-13%	0%	0%
Pilgrimage	180%	30%	0%	15%	0%
Platform Jumpers	0%	32%	7%	53%	5%
Quarto	1659%	8%	7%	2%	0%
Sheep and Wolf	398%	57%	6%	5%	0%
Tic-Tac-Toe	0%	10%	0%	0%	0%
Zhadu	577%	8 %	7%	182%	0%

information on tournament features and the overall description of our player are presented in dedicated papers [29], [17].

A. The main reasons of the proposed approach effectiveness

It is difficult to tell which aspects of our approach give an edge over the Prolog implementation for particular games, however, some general trends can be observed. Our interpreter has an advantage in games with quick playouts such as Tic-Tac-Toe, Knight Tour, Eight Puzzle or Bomberman. This may be caused by cache-friendly usage of memory, so the CPU can virtually load large parts of the game logic into the cache. We also benefit from the fact that there are no nested arguments in the system since they are converted to non-nested ones (nesting to a depth greater than one is not present in GDL). This property significantly simplifies both storing and handling the arguments because each of them can be represented by a single number. This, in turn, is one of the reasons why results of queries can be stored as big chunks of memory.

In order to provide a deeper insight into the reasons of soundness of our approach we enumerate below some of the Prolog's inefficiencies (in the GGP context) and describe the ways our interpreter tackles them:

1) Prolog is optimized to give any result as fast as possible. It accepts a so-called goal to prove (which is a question to the inference engine) as an input and, if succeeds, returns the first answer. The answer is a boolean success value: TRUE or FALSE and, in the case of TRUE, a proposition fulfilling the goal (e.g. the first legal move for $(legal\ ?player\ ?move)$). Such results are streamed one by one, because in many cases the complete set of proposition is not required. In GDL, however, there is no need for such behavior, because for every relation a complete set of instantiations is needed for the purpose of simulating a game. Indeed, we always need the complete set of legal moves, goal values, initial and next facts.
2) Prolog operates on variables. It uses the Warren Abstract Machine (WAM) [33] concept. An abstract machine for the execution of Prolog consists of a memory architecture and an instruction set. However, mainly due to the fact that compound (nested) arguments can be mixed with simple ones, each argument is treated separately and loaded into a memory register for storing its instantiation. We eliminate compound arguments (see section IV-C) so each argument can be represented by a single number. Therefore, it is easy to calculate the size of each fact in the memory which is equal to the number of symbols multiplied by the size of a single number representation:
$size(fact) = symbolsCount * sizeof(short) = 2 * symbolsCount[Bytes]$

As a consequence we can easily compute the memory footprint of any set of facts and store it in a continuous block of memory (one-dimensional array).

3) Prolog uses choice-points and backtracking. There is a reference to nondeterminism here, because Prolog operation may have more than one result and because the results cannot be predicted in a linear analysis. A tree structure reflects the dependencies among rules, in a general case. Prolog engine resolves nondeterminism by making an assumption once it finds a non-instantiated variable. This is called a choice-point, i.e. trying a variable binding and pushing it into a stack. Starting from that choice-point, all the following computations are made as though the variable was constant until there are no more choice-points to be made or the resolution ends with a failure, e.g. there exists an unsatisfied condition. In such a case, all actions since a tentative choice-point are undone and the next choice-point is made if possible. This process is called backtracking. It is worth noticing, that a backtracking scheme may result in checking the same condition multiple-times. We found such behavior vulnerable to some GDL constructions. The approach we propose is a single-pass method operating on complete sets of partial results.
4) State update logic in Prolog is slow. The reason is that it must be done manually. First, queries corresponding to *next* relations must be completed and the results returned to the calling program. Then, an internal database of previous facts must be cleared and finally updated with the new set of facts. In our inference engine, the *next* relations already compute the results into a proper place in the memory, which is used by conditions corresponding to dynamic facts. Moreover, Prolog does not distinguish dynamic from constant facts in a resolution process. Conversely, our method can take advantage of it, e.g. by bringing in hash function for constant facts which is computationally expensive to construct but fast in use. We can also pre-compute all static queries on constant facts.

The gain in Farming Quandries and Platform Jumpers is mainly attributed to efficient sorting and hashing, which is of particular importance in these games. Our approach, on the other hand, is not very well suited for games with a complex recurrence, i.e. those with many recurrent rules and large numbers of nodes created inside recurrent calls (e.g., growing exponentially with respect to a depth of a tree). Othello is one of the examples. It is worth noting that the number of simulations per second in particular games does not deviate much from the average. A nearly identical results obtained for similar pairs of games, i.e. Connect-4/Connect-4 Suicide or Pentago/Pentago Suicide confirm stability of results (the main difference between games in the above-mentioned pairs lies in the definition of their goal conditions).

IX. Conclusions

Logic is used in various areas of computer science and there is a demand for efficient tools for dealing with logic-based problem descriptions. In this paper, we have presented a complete Game Description Language interpreter, designed for the purpose of a particular embodiment of multi-game playing called General Game Playing. The interpreter is based on the principle of a resolution tree composed of OR and AND nodes. Each node uses a flexible and efficient data structure for storing results. Symbols present in a game description are converted to 2-Byte numbers and managed with the help of a high performance pointer algebra. The system is designed to minimize memory operations. We also limit the number of temporary results due to filtering mechanism and speed-up critical computations by variables hashing.

The primary reason for making this contribution is to provide a detailed description of how to create a fast and efficient Prolog-like interpreter optimized for a concrete application. We believe that in spite of the focused usage of our interpreter this work is of general interest and may be regarded in a broader perspective, since it is strongly based on general aspects of the first-order-logic. We tried to include only those problem-dependent implementation details which are absolutely required for better understanding of universally-designed higher level concepts.

Since many low-level technical issues were omitted, the interested readers are welcome to contact the authors directly with regard to specific implementation details. The crucial parts of the interpreter are data storage and representation and the way of gathering the computed rules. In GDL, it is more robust to operate on the whole terms (rows) rather than test each of the possible variables groundings. Moreover, the class of "questions" asked to the interpreter is known in advance and includes all types of *init*, *next*, *legal*, *terminal* and *goal* rules found in game descriptions. Hence, the resolution tree and the variable matching strategy can be created during the preprocessing phase. Hashing is introduced for rarely changing sets of facts. In GDL these are constant facts and dynamic facts, which change in fixed intervals upon game state updates. The most troublesome aspect was dealing with the recursion mechanism which, when approached straightforwardly, requires the resolution tree either to be infinite or contain cycles (losing the tree property in fact).

Our interpreter outperformed the two most popular Prolog interpreters by a large margin. It appeared to be the most effective choice in 22 out of 28 test games. Eleven of these games were used during the last GGP Competition. The remaining ones were selected blindly from the GGP games repository [5]. YAP, coming second, is the most effective choice in six games whilst we found no games in our experiments for which ECLiPSe appeared to be the best choice. It is worth noticing that in the least favoured games, Zhadu and Othello we reached 43% and 45% of YAP's performance, respectively. That loss was significantly smaller than advantage observed in the case of better suited games. Our interpreter performed 86 times more simulations than ECLiPSE did in Platform Jumpers and 12 times more in Fire Sheep. YAP, in turn, turned out to be roughly 5-6 times slower than our interpreter in Bomberman, Eight Puzzle, Platform Jumpers and Sheep and Wolf. It can be concluded that YAP is faster

than ECLiPSE in general, though there are still 9 games where ECLiPSE performs better than YAP. YAP results also seem to be more repeatable.

The logic reasoning system described in this paper, can be embedded into a GGP agent, as we did with our MINI-Player, or transferred to other areas where first-order logic is used. In the future, we intend to explore the possibility of using logic computations to create methods of action selection in tree search problems. This issue is typical for, but not limited to, game playing. In GGP, an agent is provided with no domain knowledge except for the game rules. In many other problem domains such knowledge could possibly be generated in a form of logical rules and consequently used by the interpreter to enhance the performance of the whole AI system. We also plan to release the final version of the package containing the source and binaries under the GNU GPL open-source license, after its completion.

FUNDING

This work was supported by the Foundation for Polish Science under International Projects in Intelligent Computing (MPD) and The European Union within the Innovative Economy Operational Programme and European Regional Development Fund [M.S.] and the National Science Centre in Poland, project number DEC-2012/07/B/ST6/01527.

APPENDIX - DISCUSSION ABOUT PRESERVATION OF THE GDL SEMANTICS

In this appendix, we will elaborate on the correctness of our method. For illustration we will refer to the example of Tic-Tac-Toe description contained in section II. Our approach despite containing many implementation-oriented technical details uses a relatively simple algorithm on the top level. With every query to prove, often referred to as a goal in Prolog vocabulary, we get a complete set of results at once. Therefore, there is no fix-point algorithm testing satisfaction of consecutive variable bindings. There is no backtracking to the last fix-point either. To get results we look for the matching facts and rules. Conditions in the rules are tested one after another in the same fashion as the aforementioned external query. Upon resolution of a condition i.e. getting the results from it, the complete set of constraints (as in the GDL/Datalog definition) is resolved. As soon as a variable is used in a condition, all unifications are made, therefore the correctness can be preserved in just one pass over conditions. In addition, we use filters which enable doing early unifications of variables introduced on different levels in the call hierarchy (before the results will be actually returned to the higher level).

A. A semi-formal descriptive proof of preservation of the GDL semantics

In order to show that our method preserves the GDL semantic the following statement needs to be proven: given a GDL description, a fact is $true$ if and only if it is computed as $true$ by our method.

Case 1 - facts which are explicitly defined without the GDL rules (without using the $<=$ constructions).
In the case of Tic-Tac-Toe the examples include: $(cell\ ?x\ ?y\ ?piece)$ or $(role\ ?player)$. This is a trivial case, because such facts are explicitly stored in dedicated OR nodes (those few static OR nodes created one for each type of facts). A query asking for all the facts receives this data directly. If a query contains constraints, they are applied according to Algorithm 2.

In such a case, one OR node is modeling our GDL query and uses the other one which contains the unconstrained results (line 1 of Algorithm 2). Checking if a result preserves a query constraints is self-explanatory, so we will not go into details here for the sake of clarity of the presentation.

Case 2 - facts which become true due to the GDL rules (defined by $<=$ constructions).
We will show that in Algorithm VI-B. each condition in a rule is asked for the facts which satisfy this rule. The proof will be split into two parts:
2a. we will explain that a set of facts which is returned by a given rule's condition is correct.
2b. Once 2a) is proven, we will show that a rule returns correct set of facts.

Proof of 2a. Each condition is a GDL query. If this query asks for constant/state facts then we can refer to the Case 1, already covered. Otherwise, it means that a query asks for facts whose existence is determined by the rules' results. That creates a recurrently looped definition - a condition returns a correct set of results if rules defined for the condition's relation return the correct set of results. We can inductively apply this reasoning to conditions until we reach a condition which is related exclusively to constant or dynamic facts. That is the only possible stop condition for this recurrently defined reasoning. In this case, it is only necessary to prove two assumptions to make the proof complete. The first one is that the returned set of results is correct for constant and dynamic facts, which we already did in the Case 1. The second assumption is 2b which is required for completeness.

Proof of 2b. We can now assume that we obtain correct set of results from rule's conditions. We need to show that, after having taken all the conditions into account, the result returned from this rule is correct. As defined in section IV-D, we store bindings for intrinsic variables, i.e. the ones which are present in a rule header (in order to be able to return them) or which appear in at least two conditions (to be able to unify them). However, we do not simply store symbols bound to variables. What we store are vectors of length equal to the number of intrinsic variables. Each index in a vector maps the respective variable.

At the beginning there are no variables initialized, so symbols can be random. Subsequent conditions initialize zero or more variables. At the end of the process, all variables are initialized. It is then sufficient to return sub-vectors that include only indices related to the header variables. Any constant symbols present in the query are included afterwards. Let us formulate two simple properties. **Property: Linked symbols.** From each condition we take symbols bound to every variable used in that condition. What is more - we do not take symbols independently, but we rather perform intersection of the set of rule's intrinsic variables and the ones used in the condition to extract n-tuples of symbols from the condition set of results (where n is the number of shared variables). Symbols which belong to the same tuple naturally satisfy the condition because they were extracted from the condition results. Since in most cases ($n >1$) the tuples allow for only certain combinations of symbols to be valid, as opposed to (many more) possible combinations of symbols when variables are treated independently. We treat such symbols as linked together.

Property: Anytime correctness. Sets of symbols taken from conditions are placed into the global rule results container using MergeOperations (MOs). MOs are defined in such a way that they cannot break any linked symbols already present in the results. It is worth noticing that due to this property, sets of symbols bound to already used variables cannot grow, they can only be reduced by subsequent conditions. A number of combinations (n-tuples of symbols for the respective variables), however, may increase. If proving a condition fails, regardless of how many shared variables are used, proving the whole rule fails, too.

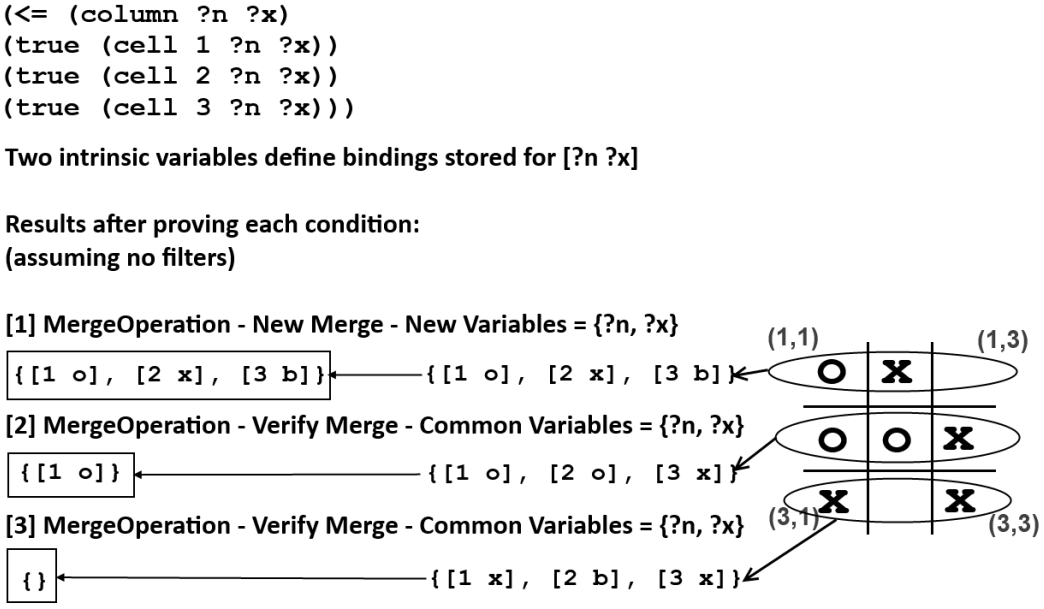

Fig. 3. Computing results of a rule.

If proving a condition succeeds, the aforementioned n-tuples are merged with the existing results via MO. A simple example (based on the the Tic-Tac-Toe's description) of a $column$ rule's proving mechanism is visualized in Figure 3.

It is to be noticed that:

- From each condition, we take vectors of all variable bindings which have to be returned by the rule or unified with other variables (Linked symbols). This proves the completeness.
- We place those vectors in the RowSet in such a way that all constraints imposed by this condition and all other conditions resolved so far are preserved (Anytime correctness). This proves the correctness.

Consequently, we state that we the returned set of results, i.e. a vector of symbols substituting variables, is correct and exhaustive.

REFERENCES

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] Krzysztof R. Apt and Mark Wallace. *Constraint Logic Programming Using Eclipse*. Cambridge University Press, New York, NY, USA, 2007.

[3] Yngvi Björnsson and Stephan Schiffel. Comparison of GDL Reasoners. In *Proceedings of the IJCAI-11 Workshop on General Game Playing (GIGA'13)*, 2013.

[4] Michael Buro. The Evolution of Strong Othello Programs. In *Proceedings of the IWEC-2002 Workshop on Entertainment Computing*, pages 81–88, Makuhari, Japan, 2002.

[5] Dresden University of Technology. Dresden Online Games Repository [ONLINE], Available at: http://130.208.241.192/ggpserver/index.jsp/.

[6] ECLiPSe. Eclipse [ONLINE] Available at: http://eclipseclp.org/.

[7] Hilmar Finnsson and Yngvi Björnsson. Simulation-Based Approach to General Game Playing. In *AAAI*. AAAI Press, 2008.

[8] Michael R. Genesereth, Nathaniel Love, and Barney Pell. General Game Playing: Overview of the AAAI Competition. *AI Magazine*, 26(2):62–72, 2005.

[9] Feng-Hsiung Hsu. *Behind Deep Blue: Building the Computer that Defeated the World Chess Champion*. Princeton University Press, Princeton, NJ, USA, 2002.

[10] Shan Shan Huang, Todd Jeffrey Green, and Boon Thau Loo. Datalog and Emerging Applications: An Interactive Tutorial. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 1213–1216, New York, NY, USA, 2011. ACM.

[11] Łukasz Kaiser and Łukasz Stafiniak. Translating The Game Description Langauge to Toss. In *Proceedings of the 2nd International General Game Playing Workshop, GIGA'11*, pages 91–98, 2011.

[12] Adrian Lancucki. GGP with Advanced Reasoning and Board Knowledge Discovery. *CoRR*, abs/1401.5813, 2014.

[13] Nathaniel Love, Timothy Hinrichs, David Haley, Eric Schkufza, and Michael Genesereth. General Game Playing: Game Description Language Specification [ONLINE] Available at: http://games.stanford.edu/readings/gdl_spec.pdf, March 2008.

[14] Jacek Mańdziuk. Computational Intelligence in Mind Games. In W. Duch and J. Mańdziuk, editors, *Challenges for Computational Intelligence*, volume 63 of *Studies in Computational Intelligence*, pages 407–442. Springer-Verlag, Berlin, Heidelberg, 2007.

[15] Jacek Mańdziuk. *Knowledge-Free and Learning-Based Methods in Intelligent Game Playing*, volume 276 of *Studies in Computational Intelligence*. Springer-Verlag, Berlin, Heidelberg, 2010.

[16] Jacek Mańdziuk. Towards Cognitively-Plausible Game Playing Systems. *IEEE Computational Intelligence Magazine*, 6(2):38–51, 2011.

[17] Jacek Mańdziuk and Maciej Świechowski. Generic Heuristic Approach to General Game Playing. In Maria Bielikov, Gerhard Friedrich, Georg Gottlob, Stefan Katzenbeisser, and Gyorgy Turan, editors, *SOFSEM*, volume 7147 of *Lecture Notes in Computer Science*, pages 649–660. Springer, 2012.

[18] J. Méhat and T. Cazenave. Ary, A General Game Playing Program. In *Board Games Studies Colloquium*, Paris, 2010.

[19] Maximilian Möller, Marius Thomas Schneider, Martin Wegner, and Torsten Schaub. Centurio, a General Game Player: Parallel, Java- and ASP-based. *KI*, 25(1):17–24, 2011.

[20] Dresden University of Technology. Dresden WWW site [ONLINE] Available at: http://www.general-game-playing.de/.

[21] Abdallah Saffidine and Tristan Cazenave. A Forward Chaining Based Game Description Language Compiler. In *Proceedings of the IJCAI-11 Workshop on General Game Playing (GIGA'11)*, 2011.

[22] Sam Schreiber. Tiltyard Server Games Repository [ONLINE], Available at: http://www.ggp.org/view/tiltyard/games/.

[23] Jonathan Schaeffer, Neil Burch, Yngvi Björnsson, Akihiro Kishimoto, Martin M ller, Robert Lake, Paul Lu, and Steve Sutphen. Checkers Is Solved. *Science*, 317(5844):1518–1522, 2007.

[24] Stephan Schiffel and Michael Thielscher. Fluxplayer: A Successful General Game Player. In *Proceedings of the 22nd AAAI Conference on Artificial Intelligence (AAAI-07)*, pages 1191–1196. AAAI Press, 2007.

[25] Michael Schofield and Abdallah Saffidine. High Speed Forward Chaining for General Game Playing. In *Proceedings of the IJCAI-11 Workshop on General Game Playing (GIGA'13)*, 2013.

[26] C. E. Shannon. Programming a Computer for Playing Chess. *Philosophical Magazine (Series 7)*, 41(314):256–275, 1950.

[27] Brian Sheppard. World-Championship-Caliber Scrabble. *Artif. Intell.*, 134(1-2):241–275, January 2002.

[28] Brian L. Stuart. Connect 4 As a Problem in Artificial Intelligence and Robotics. *SIGCSE Bull.*, 26(2):41–46, June 1994.

[29] Maciej Świechowski and Jacek Mańdziuk. Self-Adaptation of Playing Strategies in General Game Playing. *IEEE Transactions on Computational Intelligence and AI in Games*, 6(3):1–15, 2014.

[30] Gerald Tesauro. TD-Gammon, a Self-Teaching Backgammon Program, Achieves Master-Level Play. *Neural Comput.*, 6(2):215–219, March 1994.

[31] Michael Thielscher. GDL-II. *KI - Künstliche Intelligenz*, 25:63–66, 2011.

[32] K. Walędzik and J. Mańdziuk. An Automatically-Generated Evaluation Function in General Game Playing. *IEEE Transactions on Computational Intelligence and AI in Games*, 2014. Accepted for publication 08/10/2013. Available in Early Access. DOI: 10.1109/TCIAIG.2013.2286825.

[33] David H. D. Warren. An Abstract Prolog Instruction Set. Technical Report 309, AI Center, SRI International, 333 Ravenswood Ave., Menlo Park, CA 94025, Oct 1983.

[34] YAP. Yet Another Prolog [ONLINE] Available at: http://www.dcc.fc.up.pt/∼vsc/Yap.