Prolog versus specialized logic inference engine in General Game Playing

Maciej Świechowski
PhD Student at
Systems Research Institute
Polish Academy of Sciences
Warsaw, Poland
Email: m.swiechowski@ibspan.waw.pl

Jacek Mańdziuk
Faculty of Mathematics and Information Science
Warsaw University of Technology
Warsaw, Poland
Email: j.mandziuk@mini.pw.edu.pl

***Abstract*—In this paper, we study the problem of applying an inference engine, i.e. a mechanism able to derive answers from a knowledge base, to General Game Playing (GGP). Our focus is on the General Game Playing Competition framework proposed by the Stanford Logic Group. This particular embodiment of a multi-game playing uses the Game Description Language (GDL) for representing the knowledge base by means of game rules. The GDL is a variant of Datalog and a subset of Prolog, so there is a strong bond with formal logic languages. We present basic principles of our system which is dedicated to GDL. The motivation is to exceed the performance of Prolog engines applied to GDL, which is currently a common approach. Empirical results show that our solution is indeed faster in 6 out of 7 tested games.**

I. INTRODUCTION

General Game Playing (GGP) is an emerging trend in Artificial Intelligence aimed at creating programs capable of playing a variety of games. For many games, the effort to come up with a strong playing algorithm has been made so far exclusively by programmers, designers or players. The goal of GGP is to build "intelligent" agents which, being provided with a game description, discover a strong way of playing by themselves. Such agents usually integrate many AI concepts like learning, rules analysis, abstract reasoning and knowledge representation. A long-term goal is to transfer those concepts to other contexts as well as to make a step towards a general intelligence.

Effective development of multi-game playing agents is considered one of the most challenging goals in the domain of traditional (classical) board and card game playing [1], [2].

In this paper, we refer to the International General Game Playing Competition [3], [4] which was proposed by the Stanford Logic Group in 2005. The competition is held annually and provides a unique challenge, motivation and testing environment for researchers. The players are communicated game rules by the so-called Gamemaster at runtime. The Gamemaster is responsible for the HTTP communication (since players do not communicate directly) and checking whether actions chosen by participants are legal. Since the players must understand the rules of an arbitrary game, there is a formal language designed for the GGP purpose called Game Description Language (GDL) [5], [6]. GDL is a variant of Datalog [7] with the same logical syntax. The only difference to Datalog is that in the GDL there are several reserved clauses which have a predefined interpretation related to game playing. There are also some notation differences: Datalog uses a Prolog-like notation while the GDL typically uses the Knowledge Interchange Format notation (KIF) [8]. A detailed description on how the GDL works is not possible in the limited space of this paper so we suggest to consult the specification [5] and the on-line tutorial [6]. Everything in GDL is conceptualized by propositions and facts which hold true. Facts in GDL can become true from three sources. As there is no official naming convention, we propose and use the following one in scope of this paper:

- **Constant facts** Facts which are explicitly defined. They never change within the scope of one game.
- **Dynamic facts** Facts which can be initially defined (using *init* keyword) and undergo transitions based on a state update function. The state update occurs during a game, when players perform their actions. This is one of the main differences between GDL and plain Prolog which includes no game-playing logic. During that update a new set of dynamic facts is derived using a keyword *next* (see below).
- **Results of propositions (dynamic rules)** These are facts which hold true after a successful derivation of arbitrary rules (i.e. these are all instantiations of that rules). GDL users are allowed to define any number of rules unless their names collide with the keywords. We will use a common term - **relation** - for all rules or facts with the same name.

The GDL, like Datalog, is a purely predicate and declarative logic language. Both GDL and Datalog are syntactically subsets of Prolog. For instance, Prolog allows complex functional arguments, unlimited recursion and negation (there are restrictions on recursion and negation in Datalog/GDL), infinite sets, a *cut* operator as well as many system functions such as I/O. Prolog, which has roots in formal logic and first-order logic, is the most widely used among logic programming languages.

The GDL covers the class of N-player games $(N > 0)$ which are finite, deterministic and synchronous. The last restriction means that each player makes a move at the same time which is followed by an atomic advancement to a next state. In turn-based games, this general schema is fulfilled by

adding a no-operation move with no effects. The set of native GDL keywords consist of the following ones: ***role***, ***init***, ***true***, ***next***, ***legal***, ***does***, ***terminal***, ***goal*** and ***distinct***. There are also logical operations: ***not*** and ***or***. Due to space limitations we are unable to get into detail in GDL-based game definitions. Various examples of GDL game encodings, presenting all possible GDL structures, can be found in on-line repositories [9] and [10].

The most common usage of GDL game description is performing a game simulation from the initial state to a terminal state:

1) Compute the initial state by instantiating all facts in **init** relation.
2) Compute moves available to players by instantiating all facts in **legal** relation.
3) Apply chosen moves by generating **does** facts.
4) Compute the next state, i.e. the set of *next* facts, using **next** relation.
5) Clear the current state and update it with all the computed next facts.
6) Check if a state is terminal using **terminal** relation. If not, go back to 2.
7) Compute players' scores in a terminal state using **goal** relation.

The remainder of the paper is organized as follows: Section 2 provides a motivation behind designing a GDL inference engine and its comparison to Prolog. In section 3 we present description of our custom inference engine. Section 4 is devoted to empirical results and conclusions.

II. Motivation

In this section, we examine possible options of incorporating an inference engine in a GGP program. This section may serve as a guide for AI researchers who would like to get involved in GGP research.

1) Prolog.: Majority of the most successful players use Prolog as the inference engine. The favorite distribution is YAP [11], which is very fast, well-documented and comes with a C interface. Using YAP from C is easier in Unix/Linux systems, but with some effort it is also possible on Windows. Because the GDL uses a different semantics, the rules must be converted to a Prolog-compliant program. In addition, some artificial rules must be created to resolve the GDL keywords. Among the Prolog approaches to GDL, YAP is regarded as the state-of-the-art option.

2) GGP-Base.: GGP-Base [12] is an open-source java implementation of a simple framework for General Game Playing. It is maintained by the author of a two-time competition champion, program TurboTurtle, and probably used in this player although it is only a hypothesis, since no papers describing this player have been published yet. In our tests, the GGP-Base inference engine appeared to be up to 10 times slower than YAP. Moreover it can be embedded only into programs written in java, which is not a high-performance language.

3) Dresden sample players.: There are three basic players available at the Dresden WWW site [13]. Two of them use ECLiPSe Prolog distribution [14] which is significantly slower in the GGP scenario - see IV. The third one is not focused on performance and therefore it even slower than ECLiPSe. There is a recent publication [15] regarding comparison between YAP, GGP-Base and Dresden sample players. The conclusion is similar to ours, i.e. the two latter are no match for Prolog when it comes to real-time performance.

4) Custom interpreter.: Using a custom inference engine is encouraged due to three main reasons.

Firstly, its is important to note that most of the state-of-the-art players, e.g. CadiaPlayer [16], are simulation based. Also the GGP players currently developed by the authors [17], [18] as well as their previous player [19] are simulation-based agents. A tailor-made interpreter, when properly implemented, can reach a better speed performance than Prolog. After all, Prolog is a much more general and universal tool. The aim of reaching a better speed performance (reflected in higher full-game simulations count) is motivated by the fact, that the more simulations, the better state-space search is and in result the players are stronger.

Secondly, in contrast to Prolog, in a custom engine it is possible to incorporate AI algorithms into the resolution mechanism.

Thirdly, with a custom solution, we can choose a programming language we like the most.

In [20], the authors also propose a custom interpreter based on generating functions in a native language. However, such an approach has many pitfalls. The GDL descriptions are typically designed and written with Prolog in mind. Such a direct translation to other language can result in a very inefficient code. Our approach, which is more GDL oriented, has therefore a different underpinning idea. An updated and more refined version of the approach is a forward chaining interpreter [21].

Below we present a comparison of the principles of YAP resolution mechanism and a custom inference engine focusing on the aspects which make our interpreter better-suited for the GDL. Certainly, our goal is by no means to convey a message that our solutions is better than YAP in general. We only claim that it is more effective in the case of GDL.

1) Prolog is optimized to give any result as fast as possible. It accepts a so-called goal to prove and, if succeeds, returns the first answer found. In GDL there is no need for such a behavior, because for every relation a complete set of instantiations is needed.
2) Prolog operates on variables basis. It uses the Warren Abstract Machine (WAM) [22] concept, an abstract machine for the execution of Prolog consisting 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.
   In our interpreter, full instantiations of relation results are stored in a continuous block of memory allowing for efficient operations.
3) Prolog uses choice-points and backtracking to address the problem of nondeterminism, when an operation may have more than one result. In Prolog a tree

structure reflects the dependencies between rules, in a general case. A Prolog engine resolves nondeterminism by making an assumption once it finds an uninstantiated 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 is 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 the backtracking scheme may result in checking the same condition multiple-times. We found such behavior vulnerable to some GDL constructions.
The method we propose is a single-pass method operating on a complete sets of partial results.

4) State update logic in Prolog is slow. 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 custom inference engine, the *next* relations already compute the results and put them into a proper place in memory, which is used by conditions corresponding to the dynamic facts. Moreover, Prolog does not distinguish dynamic from constant facts in a resolution process. We can take advantage of it, e.g. by bringing in a hashing function for constant facts which is computationally expensive to construct but fast in use. We can also pre-compute static queries on constant facts.

III. Building Blocks of Our System

A. Preprocessing - preparation phase

The preprocessing step takes place at the very beginning of a game, after reading a GDL description (e.g. from file or through HTTP) and before the actual play. It is aimed at preparation of the interpreter for the real-time usage. The preprocessing can be seen as a one-time rules compilation. The idea is to do as much preparation as possible in order to reduce the workload during the interactive rules querying.

In the first place, a plain GDL text is tokenized. Tokens are single words (i.e. continuous text without white characters and brackets) organized into a tree-like structure. A token can have up to one parent and any number of descendant tokens. The final result of tokenization is a list of root tokens having a NULL parent. Whenever the parser reads an opening bracket it goes one level down in the current token tree. Similarly, a closing bracket means backtracking to the previous parent. Whitespace characters separate tokens of the same level except for the relation-name token. Relation, i.e. rule name or fact name can only appear as the first string after an opening bracket and becomes the parent of all other symbols up to the corresponding closing bracket.

The next step is to identify tokens based on their function. The set of tokens is transformed into a higher-level representation which operates on such entities as rule, condition, term, attribute, variable and symbol. A term is a tuple of variables and/or symbols. A condition is composed of a term (or multiple terms in the *OR* condition case) and attributes such as *true/not*. A rule is composed of a header tuple and conditions. A fact is represented by a single term. Keywords as well as relations of the same name are independently grouped together to simplify access to them.

For each condition found in the description, we prepare a unique set or rules (if such are defined) which are related to that particular condition. This involves remodeling rules if needed to fit all the constraints present in a condition. The remodeling process includes unifying variables names or replacing variables with constants. If new conditions emerge that way, the process is repeated until no new conditions are created. As a result we have copies of rules tabled for each condition. The following example presents the idea of remodeling a rule for a condition:

```
rule as defined in the GDL:
(<=(km ?u ?v ?u ?y)
   (adjacent ?v ?y)
   (coordinate ?u))

condition: (km 1 1 ?x ?x)
rule after remodeling:

(<=(km 1 1 ?u ?u)
   (adjacent 1 ?u)
   (coordinate ?u))
```

When the logical hierarchy is correctly modeled, we perform a simple normalization to have the rules defined in a compact and unified way. This process includes removal of nested arguments. In the Prolog literature it is called flattening [23]. In all possible places where a nested argument can appear, regardless if it is inside a condition or a rule header, we perform correcting actions. It is important to recursively track all the arguments which can be assigned the same instantiations. Let N denotes the maximal arity of nested arguments. If a variable is found in such place and it is not a nested expression we clone it N times. If a variable is a nested expression, we add a special empty symbol at the end of the expression cloned M times where M is N minus the expression arity.

We also remove any brackets. Constants are extended to the maximal arity by empty symbols in a similar fashion. We also eliminate redundant rules and conditions as well as redundant *not/or* expressions such as *(not(not(...)))* and redundant *(not(distinct(...)*. In addition, the order of conditions is optimized here.

1) OR-AND Tree: The final step of preprocessing is to construct a so-called OR-AND tree using a logical representation introduced in previous steps. The tree is the main part of our interpreter used to run queries on GDL rules. It could also be named a proof-tree, because each branch is a path of execution starting from the initial query and stopping when a ground set of facts is reached or query is proven false. Each branch, and therefore each level, in a tree is made of consecutive OR and AND nodes and always ends with an *OR* one. Both type of nodes expose a recurrent method called ***Prove*** returning

$TRUE$ or $FALSE$ for success and failure, respectively. If a return value is $TRUE$, the node additionally contains the results, i.e. grounded terms, for the query it proves.

1) **AND node:** represents one instance of a rule which is an implication such as:
```
(<= (kingmove ?u ?v ?x ?v)
    (adjacent ?u ?x)
    (coordinate ?v))
```
In the above example, the AND node contains two conditions and their consequence. The purpose is to call the conditions' OR nodes and merge the obtained results. The results computed in this AND node will only store three variables: ?u ?v ?x because the fourth argument is equal to the second one. In addition, only variables which appear in the implication header (consequence) or appear at least two times are stored. Variables which appear only once in a body mean nothing more than any symbol.
The node is named "AND" due to the fact that conditions are joined using conjunction. The ***Prove*** method subsequently calls ***Prove*** methods of the descendant OR nodes and immediately returns $FALSE$ when any of the methods fails or $TRUE$ when all methods pass.
2) **OR node:** represents a condition imposed on a relation. Since conditions are not related to particular implications, the OR node computes and stores all groundings which became true from any source (a rule instance, init/next facts or explicitly defined constant facts). An OR node collects the results from all sources defined for the relation and formats them in a unified way so that each argument is used. In the following example, the result has a form of [e 5 e 6] (i.e. is composed of 4 elements) unlike in the AND node case result: [e 5 6] (3 elements).
```
km: kingmove

(<=(km ?u ?v ?u ?y)
   (adjacent ?v ?y)
   (coordinate ?u))

(<=(km ?u ?v ?x ?v)
   (adjacent ?u ?x)
   (coordinate ?v))

(<=(km ?u ?v ?x ?y)
   (adjacent ?u ?x)
   (adjacent ?v ?y))
```
The top-level OR nodes are created for each basic "question" to the inference engine, e.g. which moves are legal in the current state or whether a state is terminal. Lower-level OR nodes represent more specialized "questions", which the higher ones depends on in the logical resolution process. The ***Prove*** method subsequently calls methods of the descendant AND nodes and returns $TRUE$ if at least one of the methods returned $TRUE$. Leaf-level OR nodes are at the ends of resolution branches. They are associated with state facts or constant facts. In general, rules without any conditions (therefore always true) could also be leaves in our tree, but these are converted to syntactically equivalent constant facts. Constant facts are put into the corresponding OR nodes only once because they never change, whereas dynamic facts are inserted at each game update step using the *next* keyword. In this case, the ***Prove*** method only checks whether there exists at least one fact which holds true. The OR-AND trees idea is similar to what is referred to as partial evaluation or partial execution [23].

B. *Memory Representation*

In Datalog and GDL, given a query such as (goal ?player ?value), we expect to get results having the form of fully grounded tuples, e.g. (goal white 100), (goal black 0). However, the internal representation of intermediate results is not limited by any specification as long as the final output is correct. In our scenario, we found one of the most robust approaches to be the most efficient.

First of all, each symbol which occurs in a GDL description is converted to a short integer (16-bit) number and the interpreter operates solely on numbers. The conversion back to text is done only when the final results are to be returned. We have also tested other options, e.g. reserving 32 bits for a symbol results in a significantly slower program execution, while an 8-bit representation limits the number of available symbols to only 255, which is not enough for many games.

Second of all, we propose a data structure, which we call *RowSet*, for storing results. Due to an intuitive similarity, we will use the term "row" interchangeably with the term "result" and "record". A RowSet contains raw data as well as several control fields such as number of records, maximum capacity or arity (the number of symbols per row), and functions which help to operate on the data. The raw data is just a memory pointer to short ($short*$ in C++ notation). Although the memory representation is linear due to efficiency, it is good to think of a RowSet as of a two-dimensional database grid. Columns denote variables which have to be bound (in AND nodes), or just ordered arguments (in OR nodes). Nodes of both types contain exactly one RowSet each, but there is a subtle difference which is a consequence of how the ***Prove*** method works in both types of nodes. OR nodes store only complete rows. In AND nodes, subsequent proven conditions initialize columns based on the variables they use for the first time, and rows are filled with data in the respective columns.

Third of all, we do not implement a typical "clear column" operation and instead starting from the second use of a RowSet, data from previous usage is overwritten. Due to the fact that we keep track of which columns are initialized after which conditions, there is no risk of accessing invalid data. One of the crucial features performance-wise is that there are no memory deallocations of the data stored in RowSets. Furthermore, memory allocations occur only incrementally - if a certain capacity was never reached before.

C. *Common Operations*

One of the basic operations on a RowSet is finding its intersection with another RowSet. An intersection is defined as a set of columns which are associated to the same variables in both RowSets. In the following example, we would like to

TABLE I. THE TYPES OF MERGE OPERATIONS

	New Variables	Common Variables	Variables Initialized Before *(by previous operations in the rule)*
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 special cases for negated conditions for which merge formula is inverted:			
VerifyNOT Merge	$= 0$	> 0	> 0
CheckNOT Merge	$= 0$	$= 0$	≥ 0

find rows which match on columns associated to ?m and ?x variables:

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

The common usage is to iterate over each row ***r1*** in an AND node RowSet ***R1*** and find the matching rows from a condition OR RowSet ***R2***. Searching in ***R2*** is realized in *O(n)* time using a high-performance pointer algebra. As soon as we find that symbols in some columns do not match we skip to the next row. If no such matching rows can be find, the ***r1*** row is marked to be deleted.

Additionally, RowSets containing constant and state facts are hashed per each combination of columns that is used by any *FindRow* function. Deletion of rows is delayed until it is needed. If the method which marked some rows to be deleted returns $FALSE$, we already know that the ***Prove*** method will return $FALSE$ and no rows will be used and therefore no rows have to be deleted. Actual delete operation is performed by switching the contents of a row to be deleted with the last one's and decrementing the total number of rows by one. Clearing the whole RowSet is as simple as setting the number of rows (row counter) to zero.

As introduced in the previous section, every condition in an AND set alters the current results. Conditions are resolved one after another. The condition may initialize new columns if it contains variables not used by previous conditions, it may fill the RowSet with new data and delete some existing rows. We identified seven cases which require separate handling formulae. We named them Merge Operations, because they specify how rows from a condition should be merged with current rows in AND set. We present the taxonomy of Merge Operation in Table I.

New Merge - uses only variables which were not initialized before. There can be only one such operation per rule instance (AND). Technically, the operation consists in copying entire columns from the related proved condition OR node to the AND RowSet. If possible, it is realized by one memory copy system function.

True Merge - with this operation some variables are new whereas some variables have already been initialized, so the results should be matched. For each current record in AND RowSet an intersection is found with new records from OR RowSet. 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 an OR node, the upper bound of total records after the operation is N * 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 an intersection with any new results. Rows which do not possess this property are deleted. The number of records cannot grow.

Check Merge - is the simplest case. This operation does not use any variables at all. The procedure consists in checking if there exist 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 a reverted condition of success. Merge Operations can be combined using the ***or*** keyword. Each operation works independently with its own local copy of the AND RowSet. There is also a simple *Distinct Operation* modeling the ***distinct*** keyword. With limited space we are forced to skip many details, especially lots of tricks and optimizations which are of rather technical nature but, at the same time, clearly contribute to a high execution speed.

D. Return Query

A condition which contains only uninstantiated variables with distinct names, e.g. (goal ?player ?value), is of the plain, generic form. However, not every condition and not every implication header have such a form. There can be several constraints such as duplicate variables or instantiated variables. Moreover, constraints from a condition and the corresponding implication headers do stack, as in the following example:

```
(<= (exampleRule ?a 2 ?a)
    (condition ?a 1))

(<= (exampleRule ?a ?b 5)
    (condition ?a ?b))

(<= (otherRule ?x)
    (exampleRule ?y ?x 4))
```

By stacking the condition (exampleRule ?y ?x 4) and both implication headers we can immediately remove the second one, due to a contradiction. From the second implication, the only possible result is [4 2 4].

In effect, the Return Queries are responsible for:

- mapping columns from AND nodes to the parental OR node,
- rewriting the results by putting rows at the end of OR RowSet. This may require to fill the missing constant arguments, because AND-node RowSets store only instantiations of variables,
- discarding duplicates and records which do not pass the constraints by simply not incrementing the row number.

E. Enhancements

The resolution strategy in our engine is top-down, with no backtracking and operates on sets of rules' instantiations rather than a single variable at a time. Such a combination of features is vulnerable to some language constructions in GDL/Datalog which produce an excessive number of intermediate results. We had tested many possible approaches, but the most effective solution was the use of filters.

A filter is an additional constraint joining two RowSets belonging to AND nodes. The one which is visited earlier, i.e. higher in the ***Prove method*** call-tree, is a filtering RowSet. The second one is a filtered RowSet. The idea is to keep track of a variable during the tree construction process to find which variables correspond to each other. Variables A and B correspond to each other if one of the following conditions is fulfilled:

- they share a common name within an implication,
- they appear at the same ordinal index in a condition and an implication header of the same relation as the condition,
- there exists C which corresponds to A and B (recurrent definition).

Filters are used in situations where Merge Operation initializes at least one new variable in the current AND RowSet and there exist corresponding variables already initialized higher in the tree. We found that it is important to always operate on a complete set of corresponding and initialized variables between two AND RowSets and not only on the newly initialized variables. In the example below, assume that columns [1,2] in SET1 are directly mapped to columns [1,2] in SET2.

```
RowSets records:
SET1 = {[1,4][4,1]}
SET2 = {[1,1][4,4]}
```

If we first filter SET1 on column 1 and then on column 2 we always find the matching rows and the set remains the same. If we filter on both (combined) columns we get an empty set, since there are no matching rows.

Filters are created in the preprocessing step based on the static rules analysis, however they are monitored at run-time whether any rows are actually successfully filtered. If a filter did not change the results in 200 consecutive uses, it is removed.

Additionally we use quite complex rules to identify the cases when a filter should not be created at all, e.g. when it is weaker than an already created one and therefore will be useless. Clearly, if a previous filter uses the same set (or a superset) of variables it will dominate over the newly-created one. More complex mechanisms detect the cases when a particular combination of filters dominates over a new potential filter, but their description is beyond the scope of this paper.

In general, the advantage of filter-based approach is that bindings of variables are remembered during the logical resolution process and are used to limit new potential bindings as soon as possible. Bindings of variables are not stored in a dedicated way - instead, we directly refer to the RowSets which contain them. Each such RowSet and thus each filter already preserves constraints imposed by the rule it belongs to. Therefore, the filtering mechanism is stronger than just storing variable bindings separately.

Another important optimization is based on elimination of repeatedly computed results. If there exist multiple identical conditions having the same filters then their OR nodes, by definition, must contain the same results between consecutive state updates. We identify such OR nodes and gather them into a set they have reference to. Whenever the proving algorithm visits one of these nodes, it first checks if there exists another OR node in the common set for which the results are already computed. At the beginning of a simulation and after each state update, all results of nodes in common sets are reset to 'not computed'. Usually, there are not many of such nodes, but even for a few, the speed-up factor accomplished thanks to saved computations can be pertinent.

F. Recurrence

Recurrence is a bit tricky because according to our OR-AND tree creation procedure it leads to an infinite loop. Because of this problem, we do not immediately expand OR nodes for recursively defined conditions. Instead, we store the context needed to create such nodes in the parent merge operations. This context includes the GDL condition and set of filters passed from higher levels of the tree. The filters are defined only by column mappings and relative distance of sources (filtering sets) up in the tree. Theoretically, we also need a parent AND node to properly place a new node in the tree but this is available at runtime. The actual creation of the OR nodes for recursive condition is delayed until it is needed. When the proving algorithm ends up in an merge operation for which the context is stored and by this we know it is a recursive call we either create a node or fetch it from a hash-table of already created nodes. The hash-table stores all currently unused OR nodes for which the same context was used. By the same context we mean exactly the same condition and filters having equivalent structure (the same used columns and relative distances to the source). When an OR node, either created or taken from the hash-table, is assigned we also assign correct source RowSets for the filters based on the current placement in the tree. Introduction of the hash-table greatly reduces memory footprint which otherwise can become an issue when recurrence takes place.

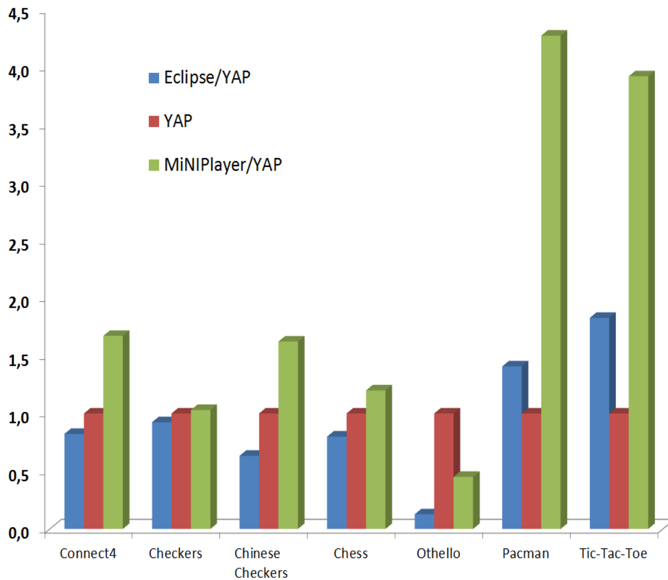

Fig. 1. A comparison of numbers of simulations completed in 30 seconds by the two Prolog-based inference engines (ECLiPSe,YAP) and our program. We selected YAP as an arbitrary reference point to keep the chart's height smaller. The X-axis represents tested games while the Y-axis denotes the performance compared to YAP (hence, the heights of red/middle bars are always equal to one).

IV. Results and Conclusions

In this paper, we have proposed an efficient solution for interpreting the GDL and compared it with an application of Prolog to perform the same task. Our solution is tailored for the GGP scenario, focuses on run-time and makes a good use of the preprocessing step. The OR-AND trees represent the idea of a precomputed logical resolution strategy. They are used together with various optimizations and with a memory-friendly implementation. The general conclusion is that our interpreter performs visibly more game-simulations per second than a Prolog-based one. We believe that after tuning particular elements of our interpreter and experimenting with the system as a whole we were able to find appropriate balance between simplicity and efficiency of proposed solution.

The interpreter has been implemented and embedded into our recent General Game Playing program called MiNI-Player [17]. We tested its real-time execution speed by counting how many random simulations, from the initial state to a terminal state, could be run in 30 seconds. We averaged the results from a few tens of tests although it is worth to note that the results are stable and repetitive. All tests were performed using the Intel Pentium Core i5 processor, 4GB RAM and a single thread.

Figure 1 shows the results compared to ECLiPSe [14] and YAP-based [11] inference engines applied to GDL. We have chosen a set of 7 popular games. In four scenarios, our engine is vastly superior, in two games it is slightly better than the second best whereas it is slower than YAP in one game (Othello). The results are very promising, although there is still a lot of room for potential improvements.

The effectiveness of proposed solution depends on various modifications and improvements implemented in our inference engine compared to typical Prolog implementation. There is no single factor responsible for such a good performance, however the mechanism of filters is the most decisive one with memory management mechanisms coming next. The filters improve performance by 5% in Connect-4, 29% in Chinese Checkers, 58% in Checkers, 100% in Chess and by an impressive value of 510% in Pacman. Their effect is negligible in Tic-Tac-Toe whereas in Othello their usage is actually required for the recurrence to terminate.

The most troublesome aspect of the interpreter is recurrence handling mechanism. The result in Othello speaks for himself and indeed serving the recurrent calls takes most of the time in Othello. It seems that the aspect of more efficient handling of recurrence is a potential avenue for improvement. Recurrence in GDL description usually encodes some kind of an iterative loop, where each recurrent call represents the iteration step. In programming languages, compilers can often predict the result of a loop, unwind it or at least simplify. We plan to follow this direction in our future work.

It is worth underlying that in addition to high execution speed, a custom interpreter allows having access to every internal part of the proving procedure. We took advantage of this possibility and embedded an approximate partial goal evaluation procedure in the MiNI-Player. The idea is to approximate a degree of certain rules' satisfiability in a given game state. This property is especially useful to determine how far is our player located from the goal states in a game.

Logic languages are widely used in academia and become more and more popular in industry, especially in multi-agent environments. We believe that this paper may serve as an inspiration for writing dedicated interpreters for logic languages, in particular for the GDL. In general, the faster the interpreter the higher the search speed and responsiveness of the system, regardless of its particular application domain.

Acknowledgment

M. Świechowski 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. The research was financed by the National Science Centre in Poland, based on the decision DEC-2012/07/B/ST6/01527.

References

[1] J. Mańdziuk, *Knowledge-Free and Learning-Based Methods in Intelligenet Game Playing*, ser. Studies in Computational Intelligence. Berlin, Heidelberg: Springer-Verlag, 2010, vol. 276.

[2] ——, "Computational Intelligence in Mind Games," in *Challenges for Computational Intelligence*, ser. Studies in Computational Intelligence, W. Duch and J. Mańdziuk, Eds. Berlin, Heidelberg: Springer-Verlag, 2007, vol. 63, pp. 407–442.

[3] M. R. Genesereth, N. Love, and B. Pell, "General game playing: Overview of the aaai competition," *AI Magazine*, vol. 26, no. 2, pp. 62–72, 2005. [Online]. Available: http://games.stanford.edu/competition/misc/aaai.pdf

[4] "Stanford general game playing internet site," Published online:, http://games.stanford.edu/. [Online]. Available: Available at: http://games.stanford.edu/

[5] N. Love, T. Hinrichs, D. Haley, E. Schkufza, and M. Genesereth, "General game playing: Game description language specication," Mar. 2008. [Online]. Available: http://games.stanford.edu/readings/gdl_spec.pdf

[6] S. L. Group, "Stanford game description language internet site." [Online]. Available: http://games.stanford.edu/games/gdl.html

[7] S. Abiteboul, R. Hull, and V. Vianu, *Foundations of Databases*. Addison-Wesley, 1995.

[8] A. Boer, T. F. Gordon, K. van den Berg, M. Di Bello, A. Förhécz, and R. Vas, "Specification of the legal knowledge interchange format," Estrella, Deliverable 1.1, 2007.

[9] GGP.org repostiory, "Ggg.org online games repository," 2010. [Online]. Available: http://www.ggp.org/view/tiltyard/games/

[10] dresden.de repository, "Dresden online games repository," 2010. [Online]. Available: http://130.208.241.192/ggpserver/public/show_games.jsp/

[11] YAP, "Yet another prolog," Mar. 1989. [Online]. Available: http://www.dcc.fc.up.pt/ vsc/Yap/

[12] S. Schreiber, "Ggp-base package." [Online]. Available: http://code.google.com/p/ggp-base/

[13] T. Dresden, "Dresden ggp internet site." [Online]. Available: http://www.general-game-playing.de/

[14] K. R. Apt and M. Wallace, *Constraint Logic Programming using Eclipse*. New York, NY, USA: Cambridge University Press, 2007.

[15] Y. Björnsson and S. Schiffel, "Comparison of gdl reasoners," in *Proceedings of the IJCAI-11 Workshop on General Game Playing (GIGA'13)*, 2013.

[16] H. Finnsson and Y. Björnsson, "Simulation-based approach to general game playing," in *AAAI*. AAAI Press, 2008. [Online]. Available: http://ru.is/faculty/hif/papers/cadiaplayer_aaai08.pdf

[17] M. Świechowski and J. Mańdziuk, "Self-adaptation of playing strategies in general game playing," *IEEE Transactions on Computational Intelligence and AI in Games*, 2014, accepted for publication 20/06/2013. Available in Early Access. DOI: 10.1109/TCIAIG.2013.2275163.

[18] 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.

[19] J. Mańdziuk and M. Świechowski, "Generic heuristic approach to general game playing," in *SOFSEM*, ser. Lecture Notes in Computer Science, vol. 7147. Springer, 2012, pp. 649–660.

[20] A. Saffidine and T. Cazenave, "A forward chaining based game description language compiler," in *Proceedings of the IJCAI-11 Workshop on General Game Playing (GIGA'11)*, 2011.

[21] M. Schofield and A. Saffidine, "High speed forward chaining for general game playing," in *Proceedings of the IJCAI-11 Workshop on General Game Playing (GIGA'13)*, 2013.

[22] D. H. D. Warren, "An abstract prolog instruction set," AI Center, SRI International, 333 Ravenswood Ave., Menlo Park, CA 94025, Tech. Rep. 309, Oct 1983.

[23] J. Lloyd, *Logic Programming: The 1995 International Symposium*. MIT Press, 1995.