Game Description Language Compiler Construction*

Jakub Kowalski and Marek Szykuła

Institute of Computer Science, University of Wrocław, Poland
{kot,msz}@ii.uni.wroc.pl

Abstract. We describe a multilevel algorithm compiling a general game description in GDL into an optimized reasoner in a low level language. The aim of the reasoner is to efficiently compute game states and perform simulations of the game. This is essential for many General Game Playing systems, especially if they use simulation-based approaches. Our compiler produces a faster reasoner than similar approaches used so far. The compiler is implemented as a part of the player Dumalion. Although we concentrate on compiling GDL, the developed methods can be applied to similar Prolog-like languages in order to speed up computations.

Keywords: General Game Playing, Game Description Language, Compiler Construction

1 Introduction

The aim of General Game Playing (GGP) is to develop a system that can play variety of games with previously unknown rules. Unlike standard artificial game playing, where designing an agent requires special knowledge about the game, in GGP the key is to create an universal algorithm performing well in different situations and environments. As such, General Game Playing was identified as a new Grand Challenge of Artificial Intelligence and from 2005 the annual AAAI GGP Competition is taking place to foster and monitor progress in this research area [5]. Because of its universal domain, GGP combines multiple disciplines [19] from searching, planning, learning [1, 4, 12, 15, 17] to evolutionary algorithms, distributed algorithms and compiler construction [11, 14, 16, 20].

In many General Game Playing systems it is crucial to have an efficient reasoning algorithm performing simulations of the game. More computations means a larger game tree traversed, more gained knowledge, deeper search or more simulations in Monte Carlo algorithms. In response to these needs, we developed our compiler. Because of used optimizations, it produces very effective reasoners which can compute game states faster than other so far known approaches.

The paper is organized as follows. Section 2 provides necessary background and describes current state of the art. Step by step details of our construction are presented in Section 3. Section 4 contains overview of experimental results. We conclude in Section 5.

* This research was supported in part by Polish MNiSW grant IP2012 052272

2 Game Description Language

To develop a general game playing system, there is a need for a standard to encode game rules in a formal way. For the sake of World Wide GGP Competition, Game Description Language (GDL) [5, 10] is used. This first-order logic language based on Datalog has enough expression power to describe all finite, turn-based, deterministic games with full information, simultaneous moves and a fixed number of players. By "finite" we mean that the set of possible game states, and the set of actions (moves) which players can choose in each state should be finite. Also every match should end after a finite number of turns. Players perform actions simultaneously, which means that in each turn all players select their moves, without knowing the decision of the others. Sequential games can be simulated by explicit adding some `noop` move for the players which should normally wait. Another strong restrictions are that no game element can be random and all players should have the full information about the game state. Extension of the GDL called GDL-II [18] removes these limits, but this leads to a more complicated system where general playing is even harder.

Syntactically GDL is very similar to Prolog. It is purely axiomatic, so there is no arithmetic or other complex game concepts (like pieces or boards) included, every such thing must be explicitly stated in the code. GDL is rule based which means that gaining information about a game state is equivalent to applying rules and extending the set of holding (true) facts. As example, in listing 1.1, we show some rules of the game *Goldrush* from Dresden GGP Server [6].

Listing 1.1. Part of the *Goldrush* game GDL code.

```
1  (role Green) (role Red)
2  (init (OnMap Green 1 1)) (init (OnMap Red 7 7))
3  (init (OnMap Obstacle 1 6)) (init (OnMap Obstacle 2 4)) ...
4  (init (OnMap (Gold 2) 1 7)) (init (OnMap (Gold 1) 3 6)) ...
5  (init (OnMap (Item Blaster 3) 7 4)) ...
6  (init (OnMap (Item Stoneplacer 3) 1 4)) ...
7  (<= (legal ?r (Move ?nx ?y))
8       (role ?r) (true (OnMap ?r ?x ?y)) (InBoard ?nx))
9       (or (+ ?x 1 ?nx) (- ?x 1 ?nx))
10 (<= (next (OnMap ?r ?x ?y))
11      (role ?r) (does ?r (Move ?x ?y)))
12 (+ 0 0 0) (+ 1 0 1) (+ 2 0 2) (+ 3 0 3) ...
13 (<= (- ?x ?y ?z) (+ ?y ?z ?x))
14 (InBoard 1) (InBoard 2) (InBoard 3) ... (InBoard 7)
```

Predicates that are arguments of `init`, `true` and `next` can be considered as a minimal set of predicates enough to restore all information about the state, which we will call as the *base predicates*. This means that the full game state (the *view* of a state) is a set of facts closed under application on the *base facts* and the next state is computed based on the previous full state and the players actions.

This leads us to the notion of the reasoner. This is an essential part of every player, allowing it to shift the state based on information from the game con-

troller. During competition, the game controller sends to a player only moves made by all players, so computing the next state, legal moves and so forth should be made at the player's side. In other words the reasoner is an implementation of the game loop (Fig. 1) described by game rules.

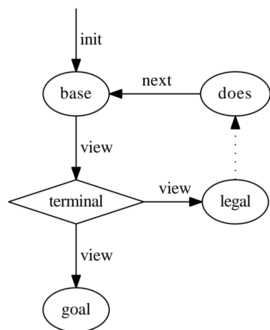

Fig. 1. Game loop: the *view* of a state is computed based on the current *base* facts. Then either game ends or the program should wait for actions of the players and then compute a next *base*.

2.1 Reasoner Implementations

An efficient reasoner implementation, while not connected with playing algorithms, is one of the most important part of a general game player. This applies to currently dominating simulation-based approach [3], but also some knowledge based ([7, 8, 15]) players make benefits from performing game simulations to tune their heuristic evaluation functions [1]. The problem of playing general games is so computationally difficult, that most of players use distributed architecture to support parallel computations on many machines [11, 13]. Complementary approach to parallelism is speeding up the process of reasoning itself. Most common implementations of the reasoner are based on prolog engines built into another programming language. The benefit from this approach is that it requires only syntactic modifications of GDL, so it results in simplicity of implementation. But using a very general inference engine to compute rules of strictly given form is a drawback causing lack of computations speed.

The most obvious method of avoiding this is compiling GDL to other language. Straightforward rewriting GDL code to C++ language, done in a top-down manner was described in [20] and Java version of this approach can be found in [13]. On the other hand, the method of forward chaining GDL to OCaml compilation with some optimizations was proposed in detail in [14]. Other approaches to make reasoning faster contains usage of propositional networks [2] and instantiating games to use binary decision diagrams [9].

In the following section we will describe in details our method of constructing GDL compiler. We use combination of a few ideas such us careful ordering of computations, optimizing control flow and designing dedicated structures to perform queries. All of these, result in a significant improvement in efficiency compared to the methods used before.

3 Compilation

Our compiler takes as an input the game rules written in GDL and outputs a structure called *compilation plan*, decoding the computation strategy for the

reasoner. The plan can be then translated to some efficient low-level programming language like C/C++. In this section we describe all major steps of constructing the plan. Our general aim is to achieve better efficiency of computing game states by the resulted reasoner. Looking to a process of playing GDL game, it can be considered as an usage of a database. The predicates are containers with some facts and we have to perform reading (queries) and writing (insertions) to these containers. Our method uses such techniques as flattening domains, optimizing data structures for containers, reordering operations, which are mostly apart from the target language.

3.1 Calculating Domains and Flattening

The first thing we need to do, after parsing GDL to some abstract tree structure, is to compute domains of the predicates' arguments. Let P be a predicate. Because predicates in GDL can be nested, every occurrence of P form a tree of its arguments. In that case we can describe such occurrence as a function from vectors encoding positions in tree to arguments symbols, so e.g. position of `3` in `OnMap (Item Blaster 3) 7 4` can be described as $\langle 0, 2\rangle$ (positions at every tree level are enumerated from 0). We want to calculate domain as a function which takes a pair of a predicate P and a tree position $\tilde{p}$ (possible for P), and returns a set of symbols that can occur in this position. Such domains are in fact a supersets of the real predicates' domains and also lose information about dependencies between arguments. However this is enough for the further calculations.

The method proposed in [9] requires computing set of dependencies where $(P, \tilde{p}) \triangleright (Q, \tilde{q})$ ($\triangleright$ is "depends on" relation) if and only if there exists a rule with P in the head and Q in the body, where at the positions $\tilde{p}$ and $\tilde{q}$ respectively, the same variable occurs. This means that every symbol in domain of $(Q, \tilde{q})$ should be also in domain of $(P, \tilde{p})$. In this approach calculating domains means resolving dependencies by extending appropriate domains until a fixpoint is obtained.

We improved this method to handle nested predicates and compute smaller domains. In our case extending domains include also domains of every subtree of variable occurrence. If $(P, \tilde{p}) \triangleright (Q, \tilde{q})$ then for every $\tilde{q}''$ which has $\tilde{q}$ as a prefix (so $\tilde{q}'' = \tilde{q} + \tilde{q}'$ for some position vector $\tilde{q}'$), also $(P, \tilde{p} + \tilde{q}') \triangleright (Q, \tilde{q}'')$. These dependencies must be dynamically computed because during the algorithm new predicates positions can be found.

Instead of $\triangleright$ we use relation $\triangleright_R$ where $(P, \tilde{p}) \triangleright_R (Q, \tilde{q})$ if dependency is created by rule R in CNF form (note that every game described in GDL can be easily converted do CNF). Let $\odot$ be the operator of domain conjunction defined as $(\odot\, d_1 \ldots d_n)\ v = d_1(v) \cap \ldots \cap d_n(v)$. For every rule R we create set $\pi^R_{(P,\tilde{p})}$ containing every $(Q, \tilde{q})$ such that $(P, \tilde{p}) \triangleright_R (Q, \tilde{q})$ holds. Then for every $\pi^R_{(P,\tilde{p})}$ we extend domain of $(P, \tilde{p})$ by $\odot\, d_i$ for $d_i \in \pi^R_{(P,\tilde{p})}$. This simulates conjunction which takes place in GDL rules and prevent domains from containing symbols unused in practice. The procedure loops for every pair $(P, \tilde{p})$ and finishes when a fixpoint is found.

To illustrate this algorithm, consider a subset of game rules shown in Listing 1.1. Calculating domains based on this example appoints the following domain of predicate `OnMap`:

$$(\texttt{OnMap}, \langle 0 \rangle) \to \{\texttt{Green,Red,Obstacle}\} \qquad (\texttt{OnMap}, \langle 0,3 \rangle) \to \{\texttt{3}\}$$
$$(\texttt{OnMap}, \langle 0,1 \rangle) \to \{\texttt{Gold,Item}\} \qquad (\texttt{OnMap}, \langle 1 \rangle) \to \{\texttt{1,2,3,4,5,6,7}\}$$
$$(\texttt{OnMap}, \langle 0,2 \rangle) \to \{\texttt{1,2,Blaster,Stoneplacer}\} \qquad (\texttt{OnMap}, \langle 2 \rangle) \to \{\texttt{1,2,3,4,5,6,7}\}$$

Because of arguments nesting, the number of leaves in parse tree can vary for one predicate. But to effectively perform queries, we need to have predicates without nesting and with a fixed arity. To achieve this, we developed a notion of *flattened* predicate and algorithms to convert standard (nested) predicate to flattened form and to perform reversed conversion.

The arity of a flattened predicate is the number of leaves in the widest of arguments assignments found in domain calculating phase. Tighter occurrences of the predicate are then stretched using special non-GDL symbol `#nil`, and each variable occurrence is extended by introducing new variables with added suffixes to avoid ambiguity. From now on, each mentioned predicate is flattened. Conversion from flattened predicate to its standard GDL form is necessary when the player needs to send a move to the game controller, having a flattened move given by the reasoner. As we made proper algorithms, we can stand:

Lemma 1. *For every occurrence of a valid GDL predicate with a fixed domain, there exist its unique flattened form. There exists an algorithm that converts these forms.*

A small part of flattened *Goldrush* game is shown in Listing 1.2 as an example. As it shows at line 10, it can create rules with unbound variables, but the values of these variables are explicitly set to `#nil` during further calculations.

Listing 1.2. Flattened GDL code.

```
2   (init (OnMap Green #nil #nil 1 1)) ...
3   (init (OnMap Obstacle #nil 1 6)) ...
4   (init (OnMap Gold 2 #nil 1 7)) ...
5   (init (OnMap Item Blaster 3 7 4)) ...
6   (init (OnMap Item Stoneplacer 3 1 4)) ...

10  (<= (next (OnMap ?r0 ?r1 ?r2 ?x0 ?y0))
11      ( role ?r0 ) ( does ?r0 ( Move ?x0 ?y0 #nil ) ) )
```

3.2 Predicates Dependency Graph and Layering

Let say that a predicate P depends on Q if there exists a game rule R such that P is the head of R and the body of R contains Q. We consider the *dependency graph*, which is a directed graph representing the dependency relation of predicates.

After the complete dependency graph is built, it is split up to subgraphs representing each of the game *phases*, depending on what we are going to compute.

The phases are: *init*, *term*, *goal*, *legal* and *next* and they correspond to the solid arrows from the game loop visualization (Fig. 1), where *term*, *goal* and *legal* belong to *view*. In such a way, the dependency graph gives us information about the predicates usage.

We can get rid of all predicates that are not needed to compute any of `legal`, `terminal`, `goal`, `next`. *Constant* predicates can be fully precomputed during the initialization phase and they stay unchanged during the rest of the game. These and base predicates belong to the *init* phase. The predicates reachable in the reversed dependency graph from `terminal`, `goal`, `legal` and `next` belong to corresponding phases respectively. There is one exception: predicates reachable from both `goal` and `legal` are put in the *term* phase. We note that it is not necessary required to compute the `goal` predicate while the state is non-terminal. This loses possible information about scores in non-terminal states, but it saves computation time and in Monte Carlo approach simulations go to the end anyway, so checking the `goal` values in non-terminals can be avoided.

All proper GDL games must be stratified, which means that for all predicates P and Q if P depends on `not` Q then P must be in a higher stratum, and all facts of a lower stratum should be deducted before deducting the upper stratum starts (which is always possible). This mechanism allows to treat GDL deduction as continuously adding facts to a database, without worrying of withdrawing them if computations are made in the proper order. In a top-down approach right computation order is for free, but in a bottom-up ordering is more flexible and can lead to better efficiency.

Despite stratification as a result of negations placement, we consider *layering*. This is a more general and a more complex approach based on dependency graphs. Each layer corresponds to a set of strongly connected components of the dependency graph. There are two types of layers:

Acyclic layer is a set of predicates such that there is no path in the dependency graph between any two predicates from this set. This means that, if only all the lower layers are computed, all the rules with these predicates in the head can be computed simultaneously and only once.

Cyclic layer is a set of predicates that are reachable from any other from this set (by using at least one edge). In this case the number of rules applications to deduct these predicates is unknown, and computations must take place until a fixpoint is reached (no new fact is added after an iteration).

Partitioning of the dependency graph to layers should be done in a way, that acyclic layers should be as large as possible, and cyclic layers as small as possible (which reduces number of computations). Currently we create the layers incrementally from the nodes without ingoing edges (so the first layer contains all "leaves" of graph). If there is a choice which layer cyclic or acyclic should be considered as a lower, the lower (first to compute) goes cyclic one.

3.3 Defining the Rules Computation Order

Mapping from predicates to layers does not make ordering of rule computation unambiguous. Consider a rule R and let L_h^R be the layer where the head of the

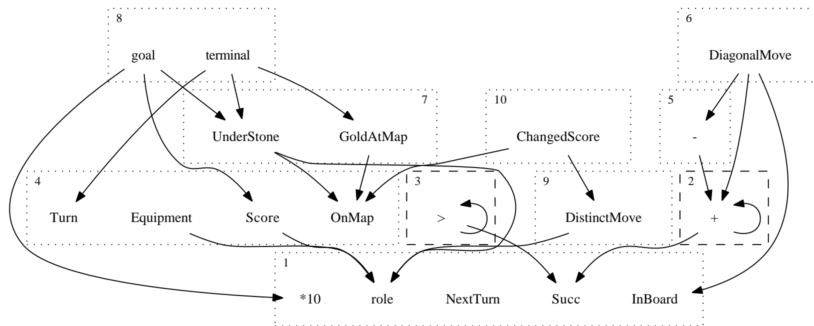

Fig. 2. Dependency graph of the game *Goldrush* (predicates `legal` and `next` are omitted due to visibility) with cyclic layers marked with dashed border.

rule belongs, and let $L_{b\ \max}^{R}$ be the maximal (the highest) layer of predicates in body of R. This means that R must be computed before layer $L_h^R + 1$ and after layer $L_{b\ \max}^{R}$, so the rule can be placed in any layer between these values. An exceptional situation is when $L_h^R = L_{b\ \max}^{R}$, which happens only for some rules from cyclic layers. In this case rule placing is unambiguous.

Intelligent rule placement can lead to some speed improvement in two main cases. First, when the predicate in the head of a rule is from a cyclic layer but the body is not ($L_h^R < L_{b\ \max}^{R}$) then the rule can be computed cheaper, because it is computed only once within a lower (assuming non cyclic) layer. Sometimes even special layers for such rules can be created. The second reason to move rules is that in some layers in admissible range there are rules similar in construction and some sharing computations between them can be done.

3.4 Filter Trees

With computed order of the rule computations we can produce the final plan. In such a plan the symbols and predicates get their unique id, and each predicate has bound information about its (flattened) domain and container type. To appoint exact ordering of game state computations, structures called *filter trees* are created.

Filter trees contains nodes, which can have child nodes. The whole computation process is just traversing the tree and performing actions according to the types of nodes. During computation a set of local variables and a set of containers are maintained. In the root the sets are empty. A variable (similarly container) defined in a node has scope bounded to the children nodes. A variable defined in a parent node is bound in the children and cannot change its value. The nodes have the following types:

- *Sequence* A node with many children which should be executed in the order.
- *Query* Queries the specified container for a given subset of facts. There can be either symbols or bound and unbound variables. For each fact in

the container which matches the query, define unbound variables by setting the values to match the fact and go into the child node. For optimization purposes a query can have additional explicit domain filter.
- *Accept* Inserts a fact to the specified container. All of the variables used in an insertion must be bound. If a new fact is added, a special *repeat* flag is set to inform a cyclic layer to be repeated.
- *Repeat* Repeats computation of its subtree until the *repeat* flag is unset. The flag is checked and cleared each time between iterations.
- *If* It has three children. The child called *test* is executed first until it is finished or a special *Return* node is reached. If *Return* was reached the child *true* is executed, otherwise *false*.

A GDL rule can be simply transformed to a filter tree without any significant modifications. A conjunction is simply nested children, an alternative can be decoded as *Sequence* and negated terms can be put in *If* filter with *Accept* in *false* subtree. Distinct between two variables is converted to a *Query* with unspecified container but with a list of distinct variables, while distinct with a variable and a symbol is just *Query* with reduced variable's domain.

A careful way of constructing filter trees can reduce much of computations. At first, if the same query occurs in two rules in the same layer it can be shared. Because every nested query can potentially cut out variable domains, the right order of nested queries can also improve efficiency. The last main optimization takes place when all variables from the heads of the rules are already defined in some query. Since the added fact is fully defined it remains only to check if the rest of queries can be satisfied. This can be realized by putting into *If* the rest nested queries, so a single positive pass through them is sufficient to immediately return and insert a fact.

We observe that only the values of variables which can reach *Insert* or *Return* node are necessary to be considered. We can restrict the domain in advance in queries defining these variables, instead of filtering them by nested queries.

We create five filter trees, one for each of the phases. An example filter tree for *next* phase of the game *blocker* [6] is presented in Fig. 3. Creating the filter trees finishes our construction, allowing generation of the final code.

3.5 Data Structures for Containers

Queries can have very different shapes. They consist of a predicate and a fixed number of arguments, depending on the predicate's arity. The arguments can be constants or variables. The simplest are those asking about existence of a particular fact like `cell 1 3 b`. More complicated are queries mixing both constants and variables, including bound variables like in `cell ?x ?x ?t`. Efficiency of performing queries depends on the data structure used to implement the container for a given predicate. An elementary analysis can be used to estimate query and insert costs for various data structures. Although more deep optimizations can take care of the proportion between queries and insertions, and occurrences of query shapes.

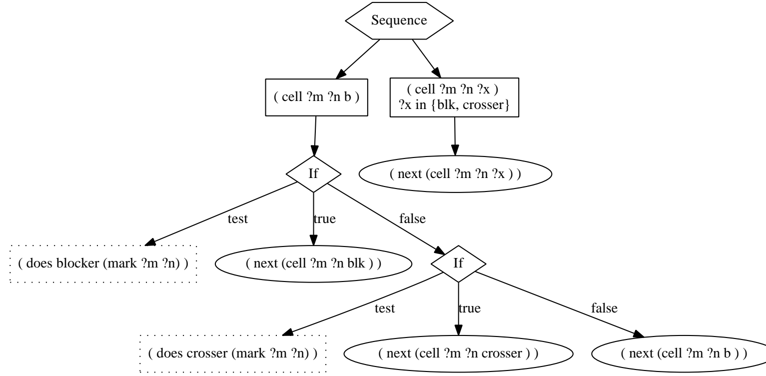

Fig. 3. *Next* filter tree of the game *blocker*. Box nodes represent *Queries* while ellipses are *Accept* filters. Every *test* branch of *If* ends with unmarked *Return* node.

In our compiler we use a few different data structures. We describe them here and perform a simple efficiency analysis. Consider a container and assume that d is the arity of facts in the container. Then let $c_1, \ldots, c_d$ be the sizes of the domains of the arguments, that is the i-th argument can take one of c_i possible values. Thus the container can hold at most $C = \prod_{i=1}^{d} c_i$ facts. Assume that n is the number of currently stored facts in the container. We assume that d is a fixed constant and comparing any two values (symbols) takes $O(1)$ time.

Querying a subset of facts in the container takes time at least $\Omega(s)$, where s is the size of the queried subset, because of further processing these facts. For simplicity we do not consider the cases with duplicated unbound variables in a single query, which are also rare cases. Thus an optimal data structure would take $O(s)$ time performing a query, and $O(1)$ time performing an insertion. A special careful should be taken for querying for a particular fact, because such an operation is often required during insertions to prevent storing duplicated facts in the container.

One of the simplest data structures is a standard dynamically-sized *vector*. Inserting to a vector takes $O(1)$ time, but querying for a particular fact or a subset of facts can take $O(n)$ time in the worst case. In opposition to the vector there is a complete *lookup array* with the fixed size C. Each allowed fact has a fixed position in the array storing a flag indicating if the fact is in the container. Insertion to an array, as well as querying for a particular fact, takes $O(1)$ time. However querying for a larger subset of facts can take $O(C)$ time, depending on the number of possible facts matching the query. Thus vectors are better for larger domains and smaller number of stored facts, and arrays conversely.

Hash and tree sets are quite efficient for insertions and querying for a particular fact, which take $O(1)$ time in a hash set and $O(\log n)$ in a balanced tree set with lexicographically ordering of facts. However querying for a subset of facts may take $O(n)$ in a hash set, and as well in a tree set if the first argument is an unbound variable.

A *trie* is a more complex tree-like structure. Levels correspond to the arguments. At the level h, each node contains d_h pointers to nodes at the height $h+1$ (except the last). These correspond to all d_h allowed values of the h-th argument. A fact is encoded by a path from the root to a leaf. A pointer is *null* if there are no facts with the corresponding value. A trie grows as more facts are added. An insertion takes $O(\sum_{i=1}^{d} c_i)$ time in the worst case (empty trie), because we must create a node at each level. Querying a particular fact takes $O(1)$ time. Efficiency of a trie in querying for a specified subset depends on the order of the arguments. A query can cost from $O(s)$ time when the constant arguments are at the beginning, up to $O(n)$ time when they are at the end. We consider tries as an universal balanced structure, since it performs quite well in most cases.

With an assumption that we have a constant number of query shapes, we developed two nearly optimal *composed structures*. Composed structure consist of a set of other structures made especially for efficient maintaining of different query shapes. The first such structure is the *trie-composed* structure based on tries, and the second is the *tree-composed* structure based on balanced tree sets. It seems that the *trie-composed* structure is better, because queries usually occur more frequently than insertions, and it generally has a lower constant factor.

Lemma 2. *The* trie-composed *structure takes $O(s)$ time for a query and $O(c_1 + \ldots + c_d)$ time for an insertion.*

Lemma 3. *The* tree-composed *structure takes $O(s + \log n)$ time for a query and $O(\log n)$ time for an insertion.*

3.6 Final Code Generation

We have implemented GDL compilation to C++. The compiled reasoner is just a module allowing to maintain game states. In particular the phases are functions initializing the reasoner and computing the *init state*, computing a *view state* given a *base state*, or computing a next *base state* given a *base state*, *view state* and moves of the players. It also provides an interface for answering if a state is terminal, getting the goal or the legal moves.

Each node of the filter tree is directly inlined in the code. In this way many technical optimizations are possible by using the context, since for example, each query can have different set of domains for the variables and we can perform explicit iteration through them.

4 Experimental Results

We have implemented the compiler as a Java program producing a reasoner in C++ as output. Our benchmark results are presented in Table 1. The resulted reasoners were compiled by g++. The main program performed uniformly random simulations of the games. Comparative reasoners uses *ECLiPSe Prolog* system. The benchmarks were done on Intel(R) Core(TM) i7-3610QM 2.3GHz with 8GB of RAM.

Table 1. The numbers of performed random simulations and visited game states per second for different games.

Game	The reasoner of Dumalion			Prolog	
	Compilation	Simulations	States	Simulations	States
Tic-Tac-Toe	0.736 s	331,647	2,860,888	2,076	15,829
Blocker	0.645 s	194,628	1,729,674	1,020	8,049
Connect Four	0.857 s	15,092	353,283	287	6,424
Breakthrough	1.074 s	3,086	200,901	55	3,553
Checkers	10.482 s	211	21,291	12	1,186
Skirmish	7.810 s	71	7,114	5	518

Although differences between computation speed between a simple Prolog engine and the optimized and compiled code are outstanding as expected, an interesting observation is that the improvement factor is less for more complicated games (such a tendency is also visible in benchmarks from [13, 14]). In other hand, while the improvement factor is smaller, the numbers of performed simulations and visited states were increased several time, and this is especially crucial for very difficult games when computing states is hard and even small speed up can give a big advantage.

Because of hardware differences and chosen method of benchmarking it is hard to make a straightforward comparison between our compiler and other approaches described in [13, 14, 20]. But after recalculating all the results to a common base simulations over a second, a roughly comparison shows that the reasoner of Dumalion can compute simulations from 2 to about 10 times faster (depending on the game) than the compiling methods described so far, and from 10 to 160 times faster than a standard Prolog engine.

5 Conclusions and Future Work

Using a compiler generator to create reasoners requires far more work than running a Prolog engine on syntactically changed GDL code, but the benefit in computation speed is significant. We mention here a few of inconveniences in our method. At first the produced reasoner must be compiled into a native code. This can take quite long if the game is complicated. The second problem is that we lose all the structure information, for example we cannot ask about a specified predicate defined in the original GDL, since it is possible that the corresponding container does not exist at all due to optimizations. Another drawback is that the process of compilation itself make the whole GGP system more complicated and harder to handle, especially if it should support parallelism.

There are many other ways of further optimizations of the plan. They include introducing new temporary containers, reordering of arguments, more careful selection of container types, reordering of queries and splitting them. As the future work, we have plan to construct GGP architecture with the aim of efficient maintain compiled code in a scalable, parallel system.

References

1. J. Clune. Heuristic Evaluation Functions for General Game Playing. In *AAAI*, pages 1134–1139. AAAI Press, 2007.
2. E. Cox, E. Schkufza, R. Madsen, and M. Genesereth. Factoring General Games using Propositional Automata. In *Proceedings of the IJCAI Workshop on General Game Playing (GIGA'09)*, 2009.
3. H. Finnsson and Y. Bjornsson. Simulation-based Approach to General Game Playing. In *AAAI*. AAAI Press, 2008.
4. H. Finnsson and Y. Bjornsson. CadiaPlayer: Search-Control Techniques. *KI*, 25(1):9–16, 2011.
5. M. Genesereth, N. Love, and B. Pell. General game playing: Overview of the AAAI competition. *AI Magazine*, 26:62–72, 2005.
6. M. Gunther and S. Schiffel. Dresden General Game Playing Server. `http://ggpserver.general-game-playing.de`.
7. S. Haufe, D. Michulke, S. Schiffel, and M. Thielscher. Knowledge-Based General Game Playing. *KI*, 25(1):25–33, 2011.
8. S. Haufe and M. Thielscher. Pushing the Envelope: General Game Players Prove Theorems. In *Proceedings of the Australasian Joint Conference on Artificial Intelligence*, volume 6464, pages 1–10, 2010.
9. P. Kissmann and S. Edelkamp. Instantiating General Games Using Prolog or Dependency Graphs. In *KI 2010: Advances in Artificial Intelligence*, volume 6359 of *LNCS*, pages 255–262. 2010.
10. N. Love, T. Hinrichs, D. Haley, E. Schkufza, and M. Genesereth. General Game Playing: Game Description Language Specification. Technical report, Stanford Logic Group, 2008.
11. J. Mehat and T. Cazenave. A Parallel General Game Player. *KI*, 25(1):43–47, 2011.
12. D. Michulke and M. Thielscher. Neural Networks for State Evaluation in General Game Playing. In *Machine Learning and Knowledge Discovery in Databases*, volume 5782 of *LNCS*, pages 95–110. 2009.
13. M. Möller, M. Schneider, M. Wegner, and T. Schaub. Centurio, a General Game Player: Parallel, Java- and ASP-based. *KI*, 25(1):17–24, 2011.
14. A. Saffidine and T. Cazenave. A Forward Chaining Based Game Description Language Compiler. In *IJCAI Workshop on General Intelligence in Game-Playing Agents (GIGA'11)*, pages 69–75, 2011.
15. S. Schiffel and M. 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.
16. S. Sharma, Z. Kobti, and S. Goodwin. General Game Playing with Ants. In *Simulated Evolution and Learning*, volume 5361 of *LNCS*, pages 381–390. 2008.
17. S. Sharma, Z. Kobti, and S. Goodwin. Knowledge Generation for Improving Simulations in UCT for General Game Playing. In *AI 2008: Advances in Artificial Intelligence*, volume 5360 of *LNCS*, pages 49–55. 2008.
18. M. Thielscher. A General Game Description Language for Incomplete Information Games. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 994–999. AAAI Press, 2010.
19. M. Thielscher. General Game Playing in AI Research and Education. In *KI 2011: Advances in Artificial Intelligence*, volume 7006 of *LNCS*, pages 26–37. 2011.
20. K. Waugh. Faster State Manipulation in General Games using Generated Code. In *IJCAI Workshop on General Game Playing (GIGA'09)*, 2009.