A Forward Chaining Based Game Description Language Compiler

Abdallah Saffidine and **Tristan Cazenave**
LAMSADE, Université Paris-Dauphine
Paris, France

Abstract

We present a first attempt at the compilation of the Game Description Language (GDL) using a bottom-up strategy. GDL is transformed into a normal form in which rules contain at most two predicates. The rules are then inverted to allow forward chaining and each predicate will turn into a function in the target language. Adding a fact to the knowledge base corresponds to triggering a function call.

Permanent and persistent facts are detected and enable to speed up the deductions. The resulting program performs playouts at a competitive level. Experimental results show that a speed-up of more than 40% can be obtained without sacrificing scalability.

1 Introduction

General Game Playing (GGP) has been described as a *Grand Artificial Intelligence (AI) Challenge* (Genesereth and Love 2005; Thielscher 2009) and it has spanned research in several directions. Some works aim at extracting knowledge from the rules (Clune 2007), while GGP can also be used to study the behavior of a general algorithm on several different games (Méhat and Cazenave 2010). Another possibility is studying the possible interpretation and compilation of the Game Description Language (GDL) (Waugh 2009; Kissmann and Edelkamp 2010) in order to process game events faster.

While the third direction mentioned does not directly contribute to AI, it is important for several reasons. It can enable easier integration with playing programs and let other researchers work on GGP without bothering with interface writing, GDL interpreting and such non-AI details. Having a faster state machine may move the speed bottleneck of the program from the GGP module to the AI module and can help performance distinction between different AI algorithms. Finally, as GDL is a high level language, compiling the rules to a fast state machine and extracting knowledge from the rules are sometimes similar things. For instance, factoring a game (Günther, Schiffel, and Thielscher 2009) can be considered a part of the compilation scheme.

GDL is compiled into forward chaining rules that contain only two predicates. As an optimization, a so-called *temporization* procedure is carried on in which permanent and persistent predicates are detected. The whole compilation process is very quick even and the resulting program performs playouts at a competitive level.

1.1 Game Automaton

The underlying model of games assumed in GGP is the Game Automaton (GA). It is a general and abstract way of defining games to encompass puzzles and multiplayer games, be it turn-taking or simultaneous. Informally, a GA is a kind of automaton with an initial state and at least one final state. The set of outgoing transitions in a state is the cross-product of the legal moves of each player in the state (non-turn players play a *no-operation* move). Final states are labelled with a reward for each player.

1.2 Objective and previous work

We focus on the compilation of rulesheets in the GDL into GAs. More precisely, the compiler described in this work takes a game rulesheet written in GDL as an input and outputs an OCAML module that can be interfaced with our playing program also written in OCAML. The module and the playing program are then compiled to native code by the standard OCAML compiler (Leroy et al. 1996) so that the resulting program runs in reasonable time. The generated module exhibits a GA-like interface. Figure 1 sums up the normal usage of our compiler and Figure 2 details the various steps between a rulesheet and an executable.

The usual approach to the compilation of GDL is to compile it to PROLOG and to use resolution to interpret it (Méhat and Cazenave 2011). Other approaches have been tried such as specialization and compilation of the resolution mechanism to the C language (Waugh 2009).

1.3 Outline of the paper

The remaining of this article is organized as follows: we first recall the definition of the Game Description Language, then we present the organization of the knowledge base. Section 4 describes the various passes used by our compiler to generate target code from GDL source code. Finally, we briefly present some experimental considerations.

2 Game Description Language

The Game Description Language (Love, Hinrichs, and Genesereth 2006) is based on DATALOG and allows to define

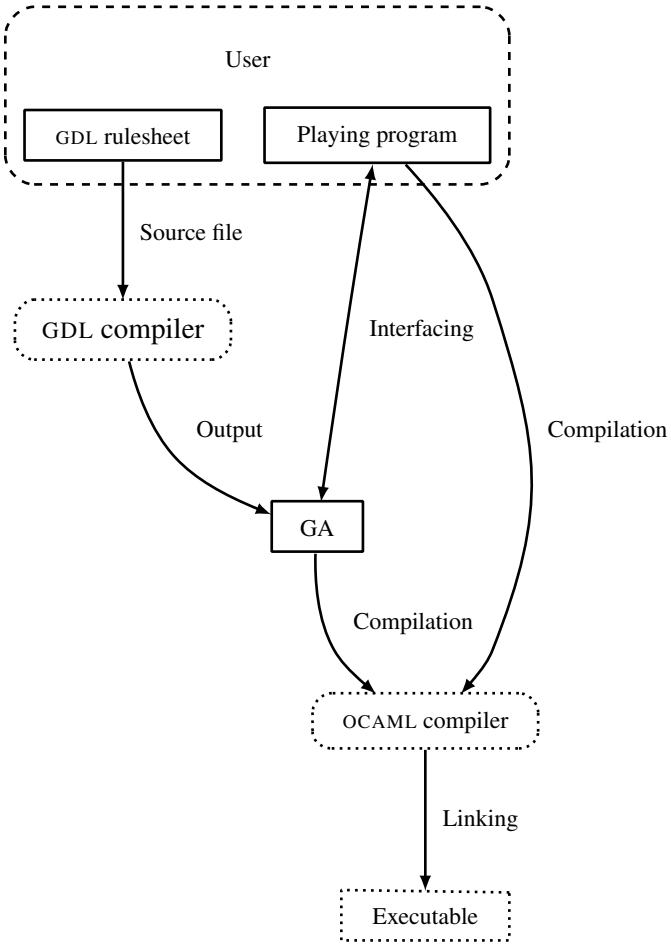

Figure 1: Interactions between a user and the GDL compiler.

Name	Arity	Appearance
`does`	2	body
`goal`	2	base, body, head
`init`	1	base, head
`legal`	2	base, body, head
`next`	1	head
`role`	1	base
`terminal`	0	base, body, head
`true`	1	body

Table 1: Predefined predicates in GDL with their arity and restriction on their appearance.

a large class of GAs (see Section for a formal definition of a GA). It is a rule based language that features function constants, negation-as-a-failure and variables. Some predefined predicates confer the dynamics of a GA to the language.

2.1 Syntax

A limited number of syntactic constructs appear in GDL.[1] Predefined *predicates* are presented in Table 1. Function constants may appear and have a fixed arity determined by context of the first appearance. Logic operators are simply `or`, `and` and `not`; they appear only in the body of rules. Existentially quantified variables may also be used bearing some restrictions defined in Section 2.1. Rules are compose of a head term and a body made of logic terms.

A GDL source file is composed of a set of grounded terms that we will call B for *base facts* and a set of rules. The Knowledge Interchange Format is used for the concrete syntax of GDL.

The definition of GDL (Love, Hinrichs, and Genesereth 2006) makes sure that each variable of a negative literal also appears in a positive literal. The goal of this restriction is probably to make efficient implementations of GDL easier. Indeed, it is possible to wait until every variable in a negative literal are bound before checking if the corresponding fact is in the knowledge base. Put another way, it enables to deal with the negation by only checking for ground terms in the knowledge base. This property is called *safety*.

2.2 Semantics

The base facts B defined in the source file are always considered to hold. The semantics also make use of the logical closure over the rules defined in the files, that is at a time τ, the rules allow to deduce more facts that are true at τ based on facts that are known to hold at τ.

The semantics of a program in the GDL can be described through the GA formalism as follows

- The set of players participating to the game is the set of arguments to the predicate `role`.
- A state of the GA is defined by a set of facts that is closed under application of the rules in the source files.

[1] We depart a bit from the presentation in (Love, Hinrichs, and Genesereth 2006) to ease the sketch of our compiler.

- The initial state is the closure over the facts that are arguments to the predicate `init`.
- Final states are those in which the fact `terminal` holds.
- For each player p and each final state s, exactly one fact of the form $\texttt{goal}(p, o_p)$ holds. We say that $0 \leq o_p \leq 100$ is the reward for player p in s. The outcome o in the final state is the tuple $(o_{p_1}, \ldots, o_{p_k})$.
- For each player p and each state s, the legal moves for p in a state s are $L_s(p) = \{m_p | \texttt{legal}(p, m_p) \text{ holds in } s\}$
- The transition relation is defined by using the predicates `does` and `next`. For a move $m = (m_{p_1}, \ldots, m_{p_k})$ in a state s, let q be the closure of the following set of facts : $s \cup \{\texttt{does}(p_1, m_{p_1}), \ldots, \texttt{does}(p_k, m_{p_k})\}$. Let n be the set of fact f such that $\texttt{next}(f)$ holds in n. The resulting state of applying m to s is the closure of the set $\{\texttt{true}(f) | f \in n\} \cup B$.

2.3 Stratification

Game rules written in GDL are stratified. It means that all the facts of the lower strate have to be deduced before starting to deduce the facts of the upper strate. Stratification is a way of coping with negation as a failure. When using backward chaining the user does not need to pay much attention to the stratification of GDL. However when using forward chaining, stratification is important since the absence of a fact cannot be concluded until every deduction on the lower strates have been carried on.

3 Runtime environment

3.1 Managing the knowledge base

In a bottom-up evaluation scheme, it is necessary to keep a data structure containing the ground facts that were found to hold in the current state. This knowledge base constitutes the biggest part of the runtime of our system. It needs to satisfy the following interface.

- An `add` procedure taking a ground fact f and adding f to the knowledge base.
- An `exists` function taking a ground fact f and indicating whether f belongs to the knowledge base.
- A `unify` function taking a fact f with free variables and returning the set of ground facts belonging to the knowledge base that can be unified with f.

It is possible to implement such a module based on a hashing of the ground facts. It allows an implementation of `add` and `exists` in constant time, but `unify` is linear in the number of present ground facts as it consists in trying to unify every fact present in the knowledge base one after another.

It is also possible to implement such a module based on a trie by first linearizing the ground facts into lists. `unify` is in this case logarithmic, but `add` and `exists` are slower.

We implemented both approaches and the first one performed slightly better. It is probably far from optimal but a better alternative could not be found yet.

3.2 Temporization of the facts

Permanent facts are facts that are present at the beginning of a game in the rule and that never change during a game. For example in some games there is a need to compare small numbers. A set of facts describes the comparison of all pairs of numbers. These facts never change and they are always present. It is not necessary to remove them and deduce them again after playing a move. Separating them from the set of facts that can change and considering them as always true can gain some time.

Persistent facts (Thielscher and Voigt 2010) are facts that always remain true once they are deduced. They are not present at the beginning of a game. They are deduced after some moves, and once they have been deduced they are present in all the following states. An example of a persistent fact is the alignment of two pieces of the same color in the game `doubletictactoe`. Once an alignment has been deduced, it will remain valid until the end of the game. So avoiding to deduce it again can gain some time.

We call *ephemeral facts* the remaining facts. That is, ephemeral facts can appear at some point in the knowledge base and then disappear as the game proceeds.

3.3 Labelling the predicates

Let $\mathbb{B}$, $\mathbb{P}$, and $\mathbb{E}$ be respectively the sets of permanent, persistent, and ephemeral facts that can appear in game. In the following, we assume the sets are disjoint and every possible fact belongs to either set.

It is not straightforward to compute theses sets at compile time without instantiating the rules. On the other hand, it is possible to use approximation for theses sets. We will call $\mathbb{B}'$, $\mathbb{P}'$, and $\mathbb{E}'$ the approximated sets. An approximation is *conservative* if the following relations hold : $\mathbb{E} \subseteq \mathbb{E}'$, $\mathbb{E} \cup \mathbb{P} \subseteq \mathbb{E}' \cup \mathbb{P}'$, and $\mathbb{E} \cup \mathbb{P} \cup \mathbb{B} \subseteq \mathbb{E}' \cup \mathbb{P}' \cup \mathbb{B}'$.

By default, it is possible to consider that every fact is ephemeral. This approximation takes $\mathbb{E}' = \mathbb{E} \cup \mathbb{P} \cup \mathbb{B}$ and $\mathbb{P}' = \mathbb{B}' = \emptyset$ and is clearly conservative. Distinguishing persistent facts can accelerate the resulting engine since it acts as some sort of tabling or memoization. Similarly, when a game is restarted, persistent facts are removed from the knowledge base but permanent facts can be kept.

A more elaborate approximation consists in labelling each predicate with b, p, or e and assigning a fact f to $\mathbb{B}'$, $\mathbb{P}'$ or $\mathbb{E}'$ depending on the predicate it is based on. We use the fixpoint procedure presented in Algorithm 1. The algorithm begins with pattern matching to have a few starting persistent labels and asserts that predicates in the upper strates are considered ephemeral. The fixpoint part then deduces the label of the predicate in a conclusion part of a rule based on the labels of the hypotheses. It uses the following order on temporal labels : $b < p < e$. Permanent labels are first obtained from rules with no hypothesis. The update operator in the fixpoint procedure is increasing which ensures its termination. Finally, predicates that could not be proved to belong to $\mathbb{B}$ or $\mathbb{P}$ are labelled ephemeral.

Input: set of rules Γ
Result: A mapping from predicates to $\{b, p, e\}$
Start with an empty mapping;
Bind `next` to e;
for *each rule* $\texttt{next}(q(args)) \Leftarrow q(args)$ **do**
    Bind `q` to p;
end
for *each rule* $q \leftarrow hyps$ **do**
    if *hyps contains a negative literal* **then**
        Bind `q` to e;
    end
end
while *A fixpoint has not bee reached* **do**
    for *each rule* $q \leftarrow r \wedge \cdots \wedge r'$ **do**
        if $r \wedge \cdots \wedge r'$ *are bound* **then**
            let β be the set of bindings for $\texttt{r} \wedge \cdots \wedge \texttt{r}'$;
            Add the current binding of `q` to β if it exists;
            Bind `q` to the maximal element of β or to b if β is empty;
        end
    end
end
for *each rule* $q \leftarrow \ldots$ **do**
    if q *is not bound* **then**
        Bind `q` to e;
    end
end
return *the completed mapping*

Algorithm 1: Fixpoint to label predicate with temporal status. We sometimes omit predicate arguments for the sake of presentation.

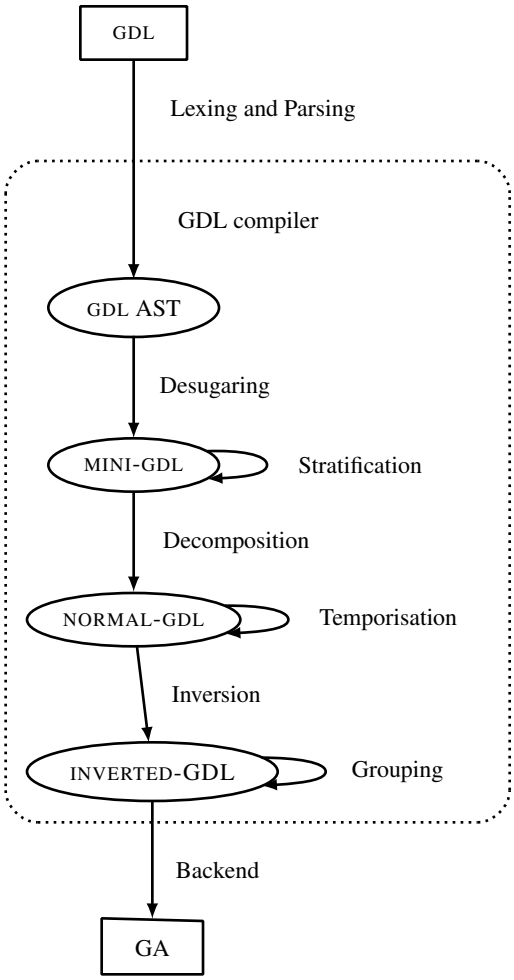

Figure 2: Steps and transformations between a GGP program written in GDL and the GA written in the output language.

4 Intermediate languages

Translating GDL programs to programs in the target language can be decomposed into several steps (see Figure 2). Each of these steps corresponds to the translation from one language to another. We used three intermediate languages in this work. The first one, MINI-GDL, is a version of GDL without syntactic sugar. In the second intermediate language, NORMAL-GDL, the rules are decomposed until a normal form with at most two hypotheses per rule is reached. The transition between a declarative language and an imperative one takes place when the program is transformed into INVERTED-GDL. Finally the program in the INVERTED-GDL is transformed in an abstract syntax tree of the target language.

4.1 Desugaring

MINI-GDL is a subset of GDL that has the same expressive power. For instance, there is no equal predicate in GDL and many rulesheets use the negation of the `distinct` pred-

icate to express equality. On the other hand, a literal in a MINI-GDL rule is therefore either a regular predicate, the negation of a regular predicate or the special `distinct` predicate. Disjunctions in rules are no longer possible.

The right hand side of a rule in GDL contains a logical formula made of an arbitrary nesting of conjunctions, disjunctions and negations.[2] The first step in transforming a rule from GDL to MINI-GDL is to put it in Disjunctive Normal Form (DNF).

A rule in DNF can now be split over as many subrules as the number of disjunctions it is made of. Indeed a rule with a conclusion c and a right hand side made of the disjunction of two hypotheses h_1 and h_2 is logically equivalent to two rules with h_1 and h_2 as hypotheses and the same conclusion $c : \{c \leftarrow h_1 \vee h_2\} \equiv \{c \leftarrow h_1, c \leftarrow h_2\}$.

A rule involving equalities can be turned into an equivalent rule without any equality. The transformation is made of two recursive processes, a substitution and a decomposition. When we are faced with an equality between t_1 and t_2 in a rule r, either at least one of the two terms is a variable (wlog. we will assume t_1 is a variable) or both are made of a function constant and a list of subterms. In the former case the substitution takes place: we obtain an equivalent rule by replacing every instance of t_1 in r by t_2 and dropping the equality. In the latter case, if the function constants are different then the equality is unsatisfiable and r cannot fire. Otherwise, we can replace the equality between t_1 and t_2 by equalities between the subterms of t_1 and the subterms of t_2. Note that function constants with different arities are always considered to be different. We can carry this operation until the obtained rule does not have any equality left.

4.2 Ensuring stratification

We have seen in Section 2.3 that it was necessary to take care of stratification in our bottom-up approach. We start by labelling each predicate with its strate number. The labels can be obtained with a simple fixpoint algorithm that we do not detail. Then, we can use the following trick: we create a new predicate $\texttt{strate}(s)$ for each possible strate s, we modify slightly the rules so that before the negation of each predicate `p` labelled with strate s the predicate $\texttt{strate}(s)$ appears.

For instance, assume in the rule $\texttt{foo} \leftarrow \texttt{bar} \wedge \neg\, \texttt{baz}$ that the predicate `baz` is labelled with strate 1. The transformation results in the rule $\texttt{foo} \leftarrow \texttt{bar} \wedge \texttt{strate(1)} \wedge \neg\, \texttt{baz}$.

After the rules are thus explicitly stratified, the evaluation scheme becomes straightforward. Apply the rules to obtain every possible fact, then add the fact corresponding to the first strate to the knowledge base, apply the rules again, then add the fact corresponding to the second strate and so on until the last strate fact is added.

4.3 Decomposition

GDL is built upon DATALOG, therefore techniques applied to DATALOG are often worth considering in GDL. Liu and Stoller (2009) presented a decomposition such that each rule in normal form is made of at most two literals in the right hand side.

Let $r = c \leftarrow t_1 \wedge t_2 \wedge t_3 \wedge \cdots \wedge t_n$ be a rule with $n > 2$ hypotheses. We create a new term t_{new} and replace r by the following two rules. $r_1 = t_{\text{new}} \leftarrow t_1 \wedge t_2$ and $r_2 = c \leftarrow t_{\text{new}} \wedge t_3 \wedge \cdots \wedge t_n$. Since variables can occur in the different terms and in c, t_{new} needs to carry the right variables so that c is instantiated with the same value when r is fired and when r_1 then r_2 are fired. This is achieved by embedding in t_{new} exactly the variables that appear on the one hand in t_1 or t_2 and on the other hand in c or any of $t_3, \ldots, t_n$. The fact that variables that appear in t_1 or t_2 but not in $t_3, \ldots, t_n$ or c do not appear in t_{new} ensures that the number of intermediate facts is kept relatively low.

The decomposition of rules calls for an order of the literals, the simplest such order is the one inherited from the MINI-GDL rule. However, it is necessary that the safety property (see Section 2.1) holds after the rules are decomposed. Consequently, literals might need to be reordered so that every variable appearing in a negative literal m appears in a positive literal before m. The programmer who wrote the game in Knowledge Interchange Format (KIF) might have ordered the literals to strive for efficiency or the literals might have been reordered by optimizations at the MINI-GDL stage. In order to minimize interferences with the original ordering, only negative literals are moved.

4.4 Inversion

After the decomposition is performed, the *inversion* transformation takes place. Each predicate `p` will generate a function in the target language. This function would in turn trigger the functions corresponding to head of rules in the body of which `p` appeared. The arguments of the target function correspond to the arguments of the predicate in NORMAL-GDL.

The inversion transformation must also take into account the fact that a given predicate can naturally appear in several rule bodies. Such a predicate need still to be translated into a single function in the target language. Therefore, an important step of the inversion transformation is to associate to each function constant f the couples (rule head, remaining rule body) of the rules that can be triggered by f.

4.5 Target language

Once the game has been translated to INVERTED-GDL, it can be processed by the back-end to have a legitimate target language program. Our implementation generates OCAML code, but it is relatively straightforward to extend it to other target languages, provided the appropriate runtime is written.

OCAML (Leroy et al. 1996) is a compiled and strongly typed functional programming language supporting imperative and object oriented styles. Some key features of OCAML simplify the back-end, particularly algebraic data types and pattern matching.

[2] Although there are some restriction on the negation possibilities.

Game	YAP	GaDeLaC	Factor
Breakthrough	**1395**	340	24%
Connect4	**3249**	990	30%
Nim1	**26066**	17500	67%
Sum15	36329	**37300**	103%
Roshambo2	68252	**70900**	104%
Bunk_t	17620	**22950**	130%
Doubletictactoe	17713	**23600**	133%
Tictactoeserial	8248	**11200**	135%
Tictactoe	31370	**45800**	146%

Table 3: Comparison of the number of random playouts performed in 30 seconds by YAP and GaDeLaC based engines.

5 Experimental results

The usual interpretation of GDL is done through an off the shelf PROLOG interpreter such as YAP (Costa et al. 2006). We implemented the proposed compiler using an OCAML back-end. We named it GaDeLaC and performed two sets of experiments described hereafter. The experiments were run on a 2.5 GHz Intel Xeon.

Table 2 shows the time needed in seconds (s) to transform various GDL source files to the corresponding OCAML files and the time needed by the OCAML compiler to compile the resulting files into an executable. We also provide the size of the main files involved in kilo-byte (KB). Namely the original GDL source file with extension `.gdl`, the translated OCAML file with extension `.ml` and the executable file.

As can be seen from Table 2, the proposed compilation scheme is pretty efficient since the whole process takes less than one second for typical games. The compilation scheme proposed in this article scales very-well as can be seen from the time needed to compile the most stressing games `Duplicatestatelarge` and `Statespacelarge` which is less than thirty seconds.

We then need to measure the speed of the resulting executable. A simple benchmark to test the speed of a GGP engine is to play a large number of games using a random policy. The second set of experiments (see Table 3) consists in counting the number of random playouts that can be played from the initial state during 30 seconds. Jean Méhat was kind enough to provide us with comparative data from his state of the art player Ary (Méhat and Cazenave 2011) which uses YAP to interpret GDL. The comparative data was obtained on a 2 GHz Intel CPU.

To gain reasonable confidence in the data presented in Table 3, the following care was taken. Each number in the GaDeLaC column is the average of ten runs of 30 seconds each. As can be expected, GaDeLaC is not bug free, however, we recorded the average playout length as well as the frequency of each possible game outcome and compared these statistics to the equivalent ones provided in the Ary data set. The playout statistics of GaDeLaC matched those of Ary for each game in Table 3.

GaDeLaC is only a proof-of-concept implementation but the results presented in Table 3 are very encouraging. A GaDeLaC based engine is significantly faster than a PROLOG based engine on several games but it is also slower on a couple of other games. Unfortunately we do not have any definitive characterization of game rules that would allow to decide whether a resolution-based engine would be quicker than a bottom-up engine.

6 Discussion and future work

The translation in GaDeLaC keeps the first-order nature of the rules, as a consequence the improvement factor is less than what Kevin Waugh could obtain in Waugh (2009), on the other hand using an instantiation pass to preprocess the rules such as suggested by Kissmann and Edelkamp (2010) remains a possibility and is likely to lead to further acceleration.

Several extensions to this work are anticipated. Devising a more adapted runtime data structure would surely allow a considerable speed gain. Partial instantiation of well-chosen predicate would allow more predicates to be considered persistent without compromising scalability. To the best of our knowledge this work is the first first-order forward chaining approach to GGP and GDL rulesheets are tested and optimized with resolution based engines, it could therefore be interesting to see if optimizing the GDL rules of a game towards bottom-up based engines would change much the rulesheet and accordingly the performance. It might also be interesting to test methods to direct the bottom-up evaluation from the deductive databases community, for instance *magic sets* (Kemp, Stuckey, and Srivastava 1991) or *demand transformation* (Tekle and Liu 2010, Section 4) could prove appropriate. Finally, writing back-ends for different languages is envisioned, for instance, generating C code might improve the performance even further and would enhance compatibility with other artificial players.

7 Conclusion

We have presented a bottom-up based Game Description Language compiler. It transforms GDL into rules that have only two conditions. It then uses OCAML to perform forward chaining with these rules. It performs playouts at a speed competitive with Ary for most of the games we tested. This is a promising result since more optimizations are still possible.

Acknowledgements

The authors would like to thank Jean Méhat for contributing comparative data from his competitive player, Peter Kissmann for his insights on the intricacies of the Game Description Language, and Bruno De Fraine for his various advice. The authors would also like to thank the anonymous reviewers for their detailed comments.

References

[1] Clune, J. 2007. Heuristic evaluation functions for general game playing. In *AAAI*, 1134–1139. AAAI Press.

[2] Costa, V.; Damas, L.; Reis, R.; and Azevedo, R. 2006. YAP Prolog user's manual. *Universidade do Porto*.

Game	Size of the `.gdl` file (KB)	Size of the `.ml` file (KB)	Size of the object file (KB)	Transformation time GDL → OCAML (s)	Compilation time OCaml → object file (s)
Breakthrough	3.6	48	115	0.024	0.46
Tictactoe	3.2	22	55	0.012	0.27
Duplicatestatesmall	3.7	47	108	0.022	0.43
Duplicatestatemedium	23	337	755	0.170	3.6
Duplicatestatelarge	73	1100	2500	0.980	28
Statespacesmall	2.0	34	76	0.018	0.33
Statespacemedium	13	236	516	0.108	2.3
Statespacelarge	45	825	1700	0.620	16

Table 2: Measuring the compilation times and file sizes.

[3] Genesereth, M., and Love, N. 2005. General game playing: Overview of the AAAI competition. *AI Magazine* 26:62–72.

[4] Günther, M.; Schiffel, S.; and Thielscher, M. 2009. Factoring general games. In *Proceedings of the IJCAI-09 Workshop on General Game Playing (GIGA'09)*, 27–34.

[5] Kemp, D.; Stuckey, P.; and Srivastava, D. 1991. Magic sets and bottom-up evaluation of well-founded models. In *Proceedings of the 1991 Int. Symposium on Logic Programming*, 337–351.

[6] Kissmann, P., and Edelkamp, S. 2010. Instantiating general games using prolog or dependency graphs. In *KI 2010: Advances in Artificial Intelligence*, 255–262. Springer.

[7] Leroy, X.; Doligez, D.; Garrigue, J.; Rémy, D.; and Vouillon, J. 1996. The Objective Caml system. *Software and documentation available from http://pauillac.inria.fr/ocaml*.

[8] Liu, Y. A., and Stoller, S. D. 2009. From datalog rules to efficient programs with time and space guarantees. *ACM Trans. Program. Lang. Syst.* 31(6):1–38.

[9] Love, N. C.; Hinrichs, T. L.; and Genesereth, M. R. 2006. General Game Playing: Game Description Language specification. Technical report, LG-2006-01, Stanford Logic Group.

[10] Méhat, J., and Cazenave, T. 2010. Combining UCT and nested Monte-Carlo search for single-player general game playing. *IEEE Trans. on Comput. Intell. and AI in Games* 2(4):271–277.

[11] Méhat, J., and Cazenave, T. 2011. A Parallel General Game Player. *KI-Künstliche Intelligenz* 25(1):43–47.

[12] Tekle, K., and Liu, Y. 2010. Precise complexity analysis for efficient datalog queries. In *Proceedings of the 12th international ACM SIGPLAN symposium on Principles and practice of declarative programming*, 35–44. ACM.

[13] Thielscher, M., and Voigt, S. 2010. A temporal proof system for general game playing. In *AAAI*.

[14] Thielscher, M. 2009. Answer set programming for single-player games in general game playing. In *ICLP*, 327–341. Springer.

[15] Waugh, K. 2009. Faster state manipulation in general games using generated code. In *Proceedings of the IJCAI-09 Workshop on General Game Playing (GIGA'09)*.