In *Proceedings of the Workshop on Knowledge Compilation and Speedup Learning*, at Machine Learning Conference, Amherst, Mass., 1993.

Logic Programming for General Game-Playing

Barney Pell[1]
Computer Laboratory
University of Cambridge
Cambridge, CB2 3QG, UK
E-mail: bdp@cl.cam.ac.uk

Abstract

The attempt to construct a general game-playing program is made difficult by the opposing goals of generality and efficiency. This paper shows how application of standard techniques in logic-programming (abstract interpretation and partial evaluation) makes it possible to achieve both of these goals. Using these techniques, we can represent the semantics of a large class of games in a general and declarative way, but then have the program transform this representation into a more efficient version once it is presented with the rules of a new game. This process can be viewed as moving some of the responsibility for game analysis (that concerned with efficiency) from the researcher to the program itself.

1 Introduction

Meta-Game Playing [Pell, 1992a] is a new approach to games in Artificial Intelligence, in which we construct programs to play new games (from a well-defined class) which are output by an automatic game generator. As only the *class* of games is known in advance, a degree of human bias is eliminated, and playing programs are required to perform any game-specific optimisations without human assistance.

Other work [Pell, 1992b] has defined, analysed, and evaluated the various components necessary for formalising the problem of playing Metagame in

[1]Parts of this work have been supported by RIACS, NASA Ames Research Center [FIA], the American Friends of Cambridge University, Trinity College (Cambridge), and the Cambridge University Computer Laboratory.

the class of symmetric chess-like games, which includes chess, chinese-chess, checkers, and noughts and crosses, among others. That work provides a definition of the class of games which will be played, formal grammars defining the language in which both specific games and moves within these games will be encoded, and a transparent game generator to produce new problem instances. This paper discusses how these abstract definitions can be realised in a program to actually play Metagame.

1.1 Representation and Efficiency

Of particular importance in developing more general problem solving systems, such as a Metagame-playing program, are the linked issues of *representation* and *efficiency*. As discussed in [Russell and Wefald, 1992], AI is concerned with making reasonable decisions with limited resources. If we neglect time concerns, then all the games in this class can be seen, from the pure game-theoretic perspective, as "trivial", in that their value can be calculated perfectly [von Neumann and Morgenstern, 1944]. If we take time resources into consideration, however, it is clear that programs which perform their basic computations (such as move generation) more efficiently than others have a marked competitive advantage, other factors being equal. This explains why much current research in computer game playing focusses on extremely efficient implementations of the basic computations, to the extent that each idiosyncrasy of the rules of a particular game are optimised in advance by the designers of the playing program (for example, [Ebeling, 1986]).

However, an approach relying on a highly-efficient but special-purpose representation encounters difficulties when applied to developing a Metagame player. First, as the class itself is fairly general, [2] it is difficult to see a way of hand-optimising the entire class of games in advance. Second, we would like to represent the rules in a general and declarative fashion, so that the program can explicitly reason about them for a variety of purposes.

A natural way to reconcile these two opposing goals to some extent is to automate the process by which a general program is specialised to handle specific sets of problems. Two well-understood techniques for doing this are *partial evaluation* [Sahlin, 1991; van Harmelen and Bundy, 1988] and *abstract interpretation* [Cousot and Cousot, 1977; Cousot and Cousot, 1992]. Using these techniques, we can represent the semantics of the class of games in a general and declarative way, but then have the program transform this representation into a more efficient version once it is presented with the rules of a new game. Consistent with the philosophy of Metagame, this process can be viewed as moving some of the responsibility for game analysis (that concerned with efficiency) from the researcher to the program itself.

[2] As shown in [Pell, 1993], the class of symmetric chess-like games contains all finite two player games of perfect information, though not all such games can be represented compactly as instances of it.

The rest of this paper elaborates on the issues of representation and efficiency in our construction of a Metagame player. The first section discusses our representation of the semantics, and the second section illustrates the methods used to specialise this general representation into one more optimised for particular games.

2 Declarative Representation

The input to a playing program is a *game definition* in a formal grammar presented in [Pell, 1992b]. The program must interpret this game as an instance of symmetric chess-like games, and then play it according to its interpretation of these rules. There are essentially two straightforward ways to bring this about.

One approach would be to write a program to convert the grammatical game definition directly into a program in some language which plays according to the interpretation of that definition. Another approach, and that taken here, is to view the class itself as a meta-game: the class represents the total set of possible moves which could be made in any position, in any game which is an instance of it. A legal move (or any other property) in a position in a *particular* game G, then, can be seen as an instance of all the possible moves (or other properties) such that G satisfies the conditions on the game which validate that move (or property). This has the advantage over the first approach, in that the relationship between all particular games in the class is here specified declaratively, in addition to the relationship between positions in a particular game. This might facilitate analogical reasoning across games (see [Collins and Birnbaum, 1988]).

2.1 Game description language

To this end, we have represented the rules for the entire class of games in a *game description language* (*gdl*). The syntax and semantics of this language is very similar to Prolog, with the addition of constructs which access an implicit *current state*, *current player*, and *current game*. These constructs are as follows:

- `true(P)`: P is a state-dependent property, which is true in the current state of the game.
- `add(P)`: add state-dependent property P to the current state.
- `del(P)`: delete state-dependent property P from the current state.
- `control(Player)`: true if Player is the player on move in the current state of the game.

- game:Pred: *Pred* is a game-dependent property, which is true of the current game, from the perspective of the current player.
- transfer_control: transfers control from the player currently in control to the opponent.

As an example, Figure 1 displays a portion of the domain theory for symmetric chess-like games, as represented in *gdl*. Our encoding of this theory represents a legal move as a sequence of legal sub-moves, or pseudo_operators, and the rules in the figure are a subset of those defining a *moving* portion of a move. Thus, these rules say that if the current game has a global must_capture rule, the current player must make a capturing movement if he has one available, or a non-capturing movement if not (as in the game of checkers). Otherwise, he can make a locally-constrained move, which allows him to move a piece of his choice, and then capture or move normally based on local constraints on the piece.

The important point to note about this representation is that all the rules are expressed without *explicit* reference to the current game or to the current state, i.e. the game and state do not appear as arguments in any of the rules. In some ways, this is syntactically cleaner than representations which have state as an explicit argument in their domain axioms (for example, the *situation calculus* [Flann and Dietterich, 1989; Genesereth and Nilsson, 1987; Hölldobler, 1992]), as we can transform from implicit to explicit representation quite easily, whereas the opposite direction requires giving a particular argument of certain predicates (the state predicates) a special status throughout any routines which operate on a theory so expressed.

In our representation, the meanings of these indexical predicates are expressed in the interpreter for this language. This interpreter has explicit arguments for these predicates and interprets indexical expressions in *gdl* by binding these variables appropriately.

The *gdl* meta-interpreter is displayed in Figure 2. The first six clauses are standard for implementing a Prolog interpreter in Prolog, and the rest (starting with the case for true(GIn)) define the special constructs used for *gdl*, as discussed above.

2.2 Flexibility

One advantage to this declarative representation of the interpreter and indexical predicates, as well as the rules defining the class, is that they can all be processed and modified by a program in a variety of ways, and it is easy to use different *state representations* for different purposes. For example, implementing true_in, add_in, and del_in as relations between bags of properties corresponds to a STRIPS-like representation, whereas implementing them as situational fluents corresponds to a situation-calculus representation [Genesereth and Nilsson, 1987].

```
% MOVE Operator:
% If global must_capture rule,
% and player can capture, then he must.
% Otherwise, player can move any piece,
% subject to local must_capture constraints.

pseudo_op(move(Piece,Player,SqF,SqT)) ==>
        control(Player),
        if( game:global_must_capture,
            global_prefer_capture(Piece,Player,SqF,SqT),
            local_move(Piece,Player,SqF,SqT)).

global_prefer_capture(Piece,Player,SqF,SqT) ==>
        if( capturing(Piece,Player,SqF,SqT),
            true,
            moving(Piece,Player,SqF,SqT)).

local_move(Piece,Player,SqF,SqT) ==>
        true(on(Piece,Player,SqF)),
        if( game:piece_must_capture(Piece),
            local_prefer_capture(Piece,Player,SqF,SqT),
            general_moving(Piece,Player,SqF,SqT)).

local_prefer_capture(Piece,Player,SqF,SqT) ==>
        if( capturing(Piece,Player,SqF,SqT),
            true,
            moving(Piece,Player,SqF,SqT)).

general_moving(Piece,Player,SqF,SqT) ==>
        capturing(Piece,Player,SqF,SqT).
general_moving(Piece,Player,SqF,SqT) ==>
        moving(Piece,Player,SqF,SqT).
```

Figure 1: Some rules expressed in the Game Description Language

```
gseval((A,B),SIn,SOut,Game) :- !,
        gseval(A,SIn,S1,Game),
        gseval(B,S1,SOut,Game).

gseval((A;B),SIn,SOut,Game) :- !,
        ( gseval(A,SIn,SOut,Game)
        ; gseval(B,SIn,SOut,Game)).

gseval(if(Cond,Then,Else),SIn,SOut,Game) :- !,
        if(gseval(Cond,SIn,S1,Game),
           gseval(Then,S1,SOut,Game),
           gseval(Else,SIn,SOut,Game)).

gseval((not Goal),SIn,SOut,Game) :- !, SIn=SOut,
        not gseval(Goal,SIn,_,Game).

gseval(setof(X,Test,Xs),SIn,SOut) :- !, SIn=SOut,
        setof(X,
              S1^seval(Test,SIn,S1),
              Xs).

gseval(X^Test,SIn,SOut,Game) :- !,
        X^gseval(Test,SIn,SOut,Game).

gseval(true(GIn),SIn,SOut,_) :- !, SIn=SOut, true_in(GIn,SIn).

gseval(add(GIn),SIn,SOut,_) :- !, add_in(GIn,SIn,SOut).

gseval(del(GIn),SIn,SOut,_) :- !, del_in(GIn,SIn,SOut).

gseval(control(P),SIn,SOut,_) :- !, SIn=SOut, in_control(P,SIn).

gseval(transfer_control,SIn,SOut,_) :- !,
        transfer_control(SIn,SOut).

gseval(game:Pred,SIn,SOut,Game) :- !,
        in_control(Player,SIn),
        player_game(Player,Game,GameP),
        true_for_game(Pred,GameP),
        SIn=SOut.

gseval(H,SIn,SOut,Game) :- H ==> B,
        gseval(B,SIn,SOut,Game).

gseval(GIn,SIn,SIn) :- operational(GIn), call(GIn).
```

Figure 2: A Meta-Interpreter for the Game Description Language

2.3 Bidirectionality

In addition, these predicates are all *logical*, in that state is represented as a relation between two variables, `StateIn` and `StateOut`, instead of a global structure which is changed by side-effects (as in a *current board* array used in many traditional playing programs). This allows a program to use the predicates in the domain theory in both directions. For example, by constraining `SOut` in Figure 2 instead of `SIn`, a program can determine possible predecessor states, thus using the rules "in reverse" to find all the positions which would have been legal before a given move.

3 Automated Efficiency Optimisation

Given this declarative representation of the domain theory for symmetric chess-like games, we created a program which can thus take as input the grammatical specification of a particular game, and play the game by interpreting the rules for legal moves and goal achievement with respect to this particular game. This program is the initial Metagame-playing program.

Unfortunately, this generality of representation does have its costs in terms of efficiency. First, there is a high overhead to interpreting a theory instead of using a compiled version of a theory. Second, it is inefficient to consider possibilities which have no connection to the definition of a given game. For example, in interpreting the game of chess, which has no `must_capture` rules at all, there is no need for a program to spend any time checking for the presence of these rules when playing a game: it would be preferable if they could somehow be eliminated altogether when the program is playing chess.

Fortunately, both of these problems can be overcome by standard techniques from logic programming. Since both the interpreter and the domain rules are declaratively expressed, it is possible to write a program to transform them automatically into a much more efficient version of the same theory, specialised to the particular game.

3.1 Partially-evaluating game-specific properties

Partial evaluation is a technique for specialising a logic program to run more efficiently on a class of queries which is a constrained subset of those on which the program is defined. As this technique is well described in the literature (for example [Sahlin, 1991; van Harmelen and Bundy, 1988]), we will here only illustrate its application to specialising a playing program for a particular game.

As noted in Section 2.3 above, our meta-interpreter is defined in a general manner to handle any modes of instantiation in its variables (e.g. , `SIn` could be ground, `SOut` un-instantiated, or vice versa). However, when playing a particular game, we know that the `Game` variable, for example, will always be instantiated to a particular value upon invocation, so we can propagate this

```
chess_pseudo_op(move(Piece,Player,SqF,SqT)) ==>
        control(Player),
        true(on(Piece,Player,SqF)),
        chess_general_moving(Piece,Player,SqF,SqT).

chess_general_moving(Piece,Player,SqF,SqT) ==>
        chess_capturing(Piece,Player,SqF,SqT).
chess_general_moving(Piece,Player,SqF,SqT) ==>
        chess_moving(Piece,Player,SqF,SqT).
```

Figure 3: Specialised chess domain theory after partial evaluation.

information by symbolic execution. By this method, each rule in the domain theory which depends only on the current game (i.e. not on any properties of the current state) can be executed at compile-time, after which we replace (a copy of) the existing rule with the results of that execution. In the simplest case, if the rule R fails to apply to the particular game, we can be sure that any other rules R', which are conditional on R succeeding, will not be called. Thus, this entire conditional branch can be eliminated from the theory (for this particular game).

Figure 3 shows the results of applying this simple example of partial evaluation to the part of the domain theory in Figure 1. As the global and local `must_capture` predicates both fail when applied to the definition of chess, these predicates, and those which are only called after their success, can be entirely removed from the specialised theory. As the number of goal reductions is dramatically reduced, the result is a much more efficient program.

3.2 Folding the Interpreter into the Domain Theory

Although this partial evaluation has greatly simplified the domain theory, a playing program using even the new theory so far would still be inefficient, due to the overhead of interpreting the theory. However, it is in fact possible to eliminate this interpretation overhead altogether, by folding the meta-interpreter directly into the clauses in the domain theory.

This transformation rewrites the now game-specific domain theory of Figure 3 into the Prolog program in Figure 4. It works by replacing the primitive constructs of Section 2.1 (like `true(P)` and `add(GIn)`) with their interpreted counterparts from Figure 2 (like `true_in(P,SIn)` and `add_in(P,SIn,SOut)`), and threading the `SIn` and `SOut` variables through all defined (i.e. non-primitive) goals which *possibly require* them.

Thus, in Figure 4, the goal `chess_pseudo_op` is converted into a new goal which contains both an input and output state, as some of its subgoals can change state. Those defined goals which, if called, could possibly lead to calling a subgoal which *tests* state, but not to one which could possibly *change* it, get extended by a *SIn* variable only.

```
chess_pseudo_op(move(Piece,Player,SqF,SqT),SIn,SOut) :-
        in_control(Player,SIn),
        true_in(on(Piece,Player,SqF),SIn),
        chess_general_moving(Piece,Player,SqF,SqT,SIn,SOut).

chess_general_moving(Piece,Player,SqF,SqT,SIn,SOut) :-
        chess_capturing(Piece,Player,SqF,SqT,SIn,SOut).
chess_general_moving(Piece,Player,SqF,SqT,SIn,SOut) :-
        chess_moving(Piece,Player,SqF,SqT,SIn,SOut).
```

Figure 4: Optimised chess-specific program after abstract interpretation.

3.2.1 Abstract interpretation for state-dependency analysis of domain predicates

In performing this transformation, it is necessary to know, for each defined goal, whether it has a recursive subgoal that could possibly change, or at least test, state. This is a question which partial evaluation alone does not answer, as it is concerned with specialising a theory, not gathering information about it. However, this question can be answered using *abstract interpretation* [Cousot and Cousot, 1977; Cousot and Cousot, 1992].

Abstract interpretation is a technique by which we "generalise" a program to make a new approximate program. It consists of mapping the *objects* in a program into *abstract objects*, mapping the *operators* into *abstract operators*, and then executing the program, over all abstract inputs, in the *abstract space* defined by these mappings. As a result of this abstract execution, we will have an approximate characterisation of its behaviour.

An example of abstract interpretation, from [Cousot and Cousot, 1977], is the use of the *rule of signs* to determine that the result of $12638 * -156$ is negative, without actually doing the multiplication. In this rule, we replace integers with (-),0,(+), and ANY, and replace the multiplication operator with a table of how it maps pairs of these abstract objects into new ones. Two entries from this table would be:

$$
\begin{array}{lll}
(+) & *(-) & \mapsto (-) \\
(-) & *(0) & \mapsto (0)
\end{array}
$$

The abstract execution $12638 * -156 \Longrightarrow (+) * -(+) \Longrightarrow (+) * (-) \Longrightarrow (-)$ proves that $12638*-156$ is a negative number. Although this example is simple, the method is very powerful, and has been used in applications ranging from *data-flow analysis* to *mode inference* in logic programs [Warren, 1992]. As this technique, like partial evaluation, is thoroughly described elsewhere, we shall not discuss the formal foundations here, but shall instead detail its application to the game analysis problem concerning us in this section.

Abstract Space Thus, our program needs to know the state-dependency of each defined goal in the domain theory. This depends on the definition of that goal, which will either be one of the primitive constructs of Section 2.1, an operational goal, another defined goal, or a compound structure relating two or more goals (this is just a summary of the language defined by the meta-interpreter for *gdl* in Figure 1). In the framework of abstract interpretation, we thus take the *concrete objects* in our theory to be the primitive constructs and operational goals, and the *concrete operators* in our theory to be the defined and compound goals. Our *abstract space* groups objects into classes based on their state-dependency requirements, and thus consists of the following three *abstract objects*, with the associated abstract interpretations:

2 : we know that calling this goal could possibly lead to calling a goal which changes state.

1 : we know that calling this goal could possibly lead to calling a goal which tests state, but not to one which possibly changes state.

0 : we do not know that calling this goal could possibly lead to calling a goal which tests or changes state.

Abstract Objects In terms of this abstract space, it is easy to define the mapping α from concrete objects to abstract objects:

$$
\begin{array}{lcl}
\alpha : & \texttt{true(P)} & \mapsto [1] \\
\alpha : & \texttt{add(P)} & \mapsto [2] \\
\alpha : & \texttt{del(P)} & \mapsto [2] \\
\alpha : & \texttt{control(P)} & \mapsto [1] \\
\alpha : & \texttt{transfer\_control(P)} & \mapsto [2] \\
\alpha : & \texttt{G} & \mapsto [0], \text{for operational(G)}
\end{array}
$$

This mapping formalises the notion that primitive goals which are quantified solely upon `SIn` have abstract value [1], those quantified on `SOut` as well, have abstract value [2], and operational predicates, which are state-independent, have abstract value [0]. Note that we do not need to define a mapping for `game:Pred`, as by this point all such predicates have been partially evaluated away.

Abstract Operators We then map our concrete operators, the defined goals and logical constructs, into abstract operators. We shall denote an abstract operator as $\alpha(Op)$. These abstract operators are then defined as:

$$\alpha((\texttt{A,B})) = \max(A, B)$$

$$
\begin{aligned}
\alpha((\mathtt{A};\mathtt{B})) &= \max(A, B)\\
\alpha(\mathtt{if}(\mathtt{A},\mathtt{B},\mathtt{C})) &= \max(A, B, C)\\
\alpha(\mathtt{not}\,\mathtt{A}) &= \min(A, 1)\\
\alpha(\mathtt{setof}(\mathtt{X},\mathtt{A},\mathtt{Xs})) &= \min(A, 1)\\
\alpha(\mathtt{X} \wedge \mathtt{A}) &= A\\
\alpha(\mathtt{H} : \exists(B, \mathtt{H} ==> \mathtt{B})) &= \max(B : \mathtt{H} ==> \mathtt{B})
\end{aligned}
$$

These abstract operators represent the fact that a compound or defined goal can possibly cause any of its components or subgoals to be called, in which case their abstract value is the maximum of any of their components. Note that in the case of the logical operators `not` and `setof`, these would never lead to the state being changed, even if their arguments could cause a modified state to be envisioned. Thus they never map to an abstract value greater than 1.

Abstract Execution Now that we have defined the relation between our abstract program and the concrete program, we can execute the abstract program in order to determine the abstract values of each of our defined goals. Our analysis proceeds in a sequence of iterations. We start in the most abstract space, in which we know nothing about the abstract values of any of our defined goals, and execute the program in this abstract space, to determine if we necessarily gain any information which forces us to move to a less abstract space. If the execution in an abstract space leaves us in the same space, then clearly all further executions will leave us in this space also, implying that we have reached a *least fixed point* in our approximation. At this point we are finished with the abstract interpretation, and by the construction of our abstract mapping we are guaranteed that we have correctly classified all of our defined goals in terms of their state-dependency number (a proof of this can be found in [Cousot and Cousot, 1977]).

An Example In case the above description was too abstract, this analysis can be interpreted algorithmically, as follows: we begin by assuming that all defined goals have state-dependency (henceforth *stativity*) 0, which means we know nothing about their stativity. At each iteration, we update our assumptions on stativity for each defined predicate based on our assumptions from the previous iteration.

For example, our theory might consist of the rules:

$$
\begin{aligned}
A &\Longrightarrow (B, C). &\quad (1)\\
B &\Longrightarrow \mathit{true}(P). &\quad (2)\\
C &\Longrightarrow D. &\quad (3)\\
D &\Longrightarrow \mathit{add}(Q). &\quad (4)
\end{aligned}
$$

We would begin our analysis assuming that all predicates A, B, C and D have stativity 0. Based on this knowledge, our first iteration reveals that

B has stativity 1 (as calling B immediately calls a `true(P)`, which performs a test on state), and D has stativity 2. As we assumed at the start that all predicates had stativity 0, at the end of this first iteration we have not determined anything new about A or C. Then on the next iteration, from ($B = 1$ and $C = 0$) we conclude that A is at least 1. And from $D = 2$, we conclude that $C = 2$. On our third iteration, from ($B = 1$ and $C = 2$), we conclude $A = 2$. On our fourth iteration, we gain no new knowledge (our assumptions about all the goals stays the same), which means that every further iteration will have the same result. Thus, our final knowledge of the stativity of this theory is: $(A = 2, B = 1, C = 2, D = 2)$, which is correct.

The result: A chess-specific program Having completed the state dependency analysis, the program then transforms the theory to eliminate the interpreter, thus yielding the final efficient and specialised Prolog chess program in Figure 4. As all of the transformations performed here were logic-preserving, this program is still logical and bidirectional, but is now optimised to generating moves and evaluating goals only in chess positions: most of the inefficiency due to the general representation has been eliminated automatically.

Thus, the net result of all this optimisation can be viewed as a game-specific program generator which takes as input the rules of any game within the class of symmetric chess-like games and, by analysing the rules of this game as an instance of the general class, produces a special-purpose, efficient program to play just that game.

3.2.2 Implementation Details

Under Sicstus Prolog 2.1 patch 6 on a SUN4, finding all the legal moves in the initial position of Chess under the original (general) representation takes 314 seconds. On the specialised representation resulting from the processing discussed above, the same computation takes 1 second. This speedup allows our program to search 5 half-ply deep in chess in 45 minutes.

We also anticipate further speedup from other optimisations using the automatic partial evaluator in MIXTUS-PROLOG [Sahlin, 1991]. As this is one of the largest applications of MIXTUS to date, more work on it appears necessary before it can be applied usefully to this domain theory [Sahlin, 1992].

4 Summary

This paper dealt with the initial issues involved in realizing a Metagame-playing program, given the rigorous definition of the problem developed in [Pell, 1992b]. As Metagame is a more general problem, it is natural that one of the most pressing issues in the construction of a program to address this problem should be the tradeoff between the representational goals of generality and flexibility, on the one hand, and the operational goals of specialisation and efficiency on the other.

While these goals seemed incompatible at first, we show that it was possible to achieve them both to some extent, by shifting some of the work of building special-purpose programs, normally the task of the human researcher, onto the program itself. It is interesting to note that this was achieved by first developing a naive game player which was *extremely* general, flexible, and inefficient, and only then automatically transforming this program into a specialised player of a particular game.

To summarise, the approach taken in this paper to realize a practical Metagame-player consisted of the following steps:

1. **gdl:** design a general-purpose language for describing games, and represent the entire class of games as one big meta-game, where a legal move is one which could be legal in any possible situation in any game in the class.

2. **gseval:** implement a naive player as a declarative and flexible meta-interpreter for this language.

3. **peval:** partially evaluate this interpreter to specialise (a copy of) the domain theory for a specific game.

4. **stativity:** use abstract interpretation to determine the state-dependency requirements of the predicates in the specialised domain theory.

5. **transform:** use the results of this interpretation to fold the interpreter into the specialised domain theory, to eliminate the overhead of meta-interpretation. The result is an efficient Prolog program to play a specific game, without the inefficiencies due to the generality of the class definition or the flexibility of meta-interpretation.

In concluding this paper, it is interesting to observe that the issue of efficient representations in game-playing systems is often seen as an engineering concern which is somehow separate from the scientific work of playing the game in a given representation. However, the issue of *automatic* efficient change of representation is a very important scientific problem [Benjamin, 1990]. This paper has shown how techniques from logic programming can greatly assist in this endeavour.

5 Acknowledgements

Thanks to Regis Cridlig for help with the intricacies of Abstract Interpretation, to Dan Sahlin for making Mixtus available, and to Mats Carlson for help with Sicstus Prolog. Thanks to Siani Baker and Ellen Germain, and especially

to my supervisors, Steve Pulman and Manny Rayner, for careful reading of drafts of this work.

References

[Benjamin, 1990] D. Paul Benjamin, editor. *Change of Representation and Inductive Bias.* Kluwer, 1990.

[Collins and Birnbaum, 1988] Gregg Collins and Lawrence Birnbaum. Learning strategic concepts in competitive planning: An explanation-based approach to the transfer of knowledge across domains. Technical Report UIUCDCS-R-88-1443, Univesrity of Illinois, Dept. of Computer Science, Urbana, IL, 1988.

[Cousot and Cousot, 1977] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the Fourth ACM Symposium on Principles of Programming Languages*, 1977.

[Cousot and Cousot, 1992] Patrick Cousot and Radhia Cousot. Abstract Interpretation and Application to Logic Programs. *Journal of Logic Programming*, 13(2–3):103–179, 1992.

[Ebeling, 1986] C. Ebeling. *All the Right Moves: A VLSI Architecture for Chess.* PhD thesis, Carnegie-Mellon University, 1986.

[Flann and Dietterich, 1989] Nicholas S. Flann and Thomas G. Dietterich. A study of explanation-based methods for inductive learning. *Machine Learning*, 4:187–226, 1989.

[Genesereth and Nilsson, 1987] Michael R. Genesereth and Nils J. Nilsson. *Logical Foundations of Artificial Intelligence.* Morgan Kaufmann, 1987.

[Hölldobler, 1992] Steffen Hölldobler. On deductive planning and the frame problem. In A. Voronkov, editor, *Logic Programming and Automated Reasoning.* Springer-Verlag, 1992.

[Pell, 1992a] Barney Pell. Metagame: A New Challenge for Games and Learning. In H.J. van den Herik and L.V. Allis, editors, *Heuristic Programming in Artificial Intelligence 3 – The Third Computer Olympiad.* Ellis Horwood, 1992. Also appears as University of Cambridge Computer Laboratory Technical Report No. 276.

[Pell, 1992b] Barney Pell. Metagame in Symmetric, Chess-Like Games. In H.J. van den Herik and L.V. Allis, editors, *Heuristic Programming in Artificial Intelligence 3 – The Third Computer Olympiad.* Ellis Horwood, 1992. Also appears as University of Cambridge Computer Laboratory Technical Report No. 277.

[Pell, 1993] Barney Pell. *Strategy Generation and Evaluation for Meta Game-Playing*. PhD thesis, Computer Laboratory, University of Cambridge, 1993. Forthcoming.

[Russell and Wefald, 1992] Stuart Russell and Eric Wefald. *Do the Right Thing*. MIT Press, 1992.

[Sahlin, 1991] Dan Sahlin. An automatic partial evaluator for full prolog. Technical Report Research Report No. SICS/D–91/04–SE, Swedish Institute of Computer Science, 1991.

[Sahlin, 1992] Dan Sahlin. Personal communication, November 1992.

[van Harmelen and Bundy, 1988] F. van Harmelen and A. Bundy. Explanation-based generalisation = partial evaluation. *Artificial Intelligence*, 36(3):401–412, 1988.

[von Neumann and Morgenstern, 1944] J. von Neumann and O. Morgenstern. *Theory of Games and Economic Behavior*. Princeton University Press, 1944.

[Warren, 1992] David Scott Warren. Memoing for Logic Programs. *Communications of the ACM*, 35(3), March 1992.