Playing Logic Programs with the Alpha-Beta Algorithm

Jean-Vincent Loddo and Roberto Di Cosmo

Laboratoire Preuves Programmes et Systèmes (PPS)
Université Paris 7 - France
{loddo,dicosmo}@pps.jussieu.fr

Abstract. Alpha-Beta is a well known optimized algorithm used to compute the values of classical combinatorial games, like chess and checkers . The known proofs of correctness of Alpha-Beta do rely on very specific properties of the values used in the classical context (integers or reals), and on the finiteness of the game tree. In this paper we prove that Alpha-Beta correctly computes the value of a game tree even when these values are chosen in a much wider set of partially ordered domains, which can be pretty far apart from integer and reals, like in the case of the lattice of idempotent substitutions or ex-equations used in logic programming. We do so in a more general setting that allows us to deal with infinite games, and we actually prove that for potentially infinite games Alpha-Beta correctly computes the value of the game *whenever it terminates.* This correctness proofs allows us to apply Alpha-Beta to new domains, like constraint logic programming.

1 Introduction

Game theory has found various applications in the research field of programming languages semantics, so that game theory is a very active research subject in computer science. After the preliminary works of Lamarche [11], Blass [2] and Joyal [9] in the early 90s, the works of Abramsky, Malacaria and Jagadeesan [1] lead to the first fully abstract semantics for functional (PCF) or imperative (Idealized Algol) languages. Then, more recently, specialists of Linear Logics got interested in links between games and the geometry of interaction [13], whereas Curien and Herbelin showed that certain classical abstract machines could be interpreted in terms of games [4].
But these relevant works use more the *vocabulary* of games (player, move, game, strategy) than the results and the techniques of traditional Game Theory: typically, nobody is interested to know, in those games, if there is a winner, and what he wins; the focus there is on the dynamic aspect of player *interaction*, and *game composition*, not on the only interesting notion of classical game theory, *the gain.* This should not be taken as a criticism, but as proof of the richness of Game Theory, which can be useful even when one only takes its vocabulary: the generality of the concepts it manipulates (arenas, multiple and independent agents, strategies of cooperation or of non-cooperation, quantification of

the remuneration after each game) and their intuitive nature, already provide a powerful metalanguage that allows us to tackle many of the aspects of modern programming languages.
In a paper written with S. Nicolet [5], the authors showed for the first time that classical notions like payoff, propagation functions and evaluation of a game tree are not sterile in the semantics of programming languages, by introducing a two player combinatorial game whose *value*, defined by means of von Neumann's MiniMax theorem [16], is the result of the execution of a logic program. In that work, we had to introduce an ad hoc framework to deal with substitutions as game values, and to prove the correctness of this game semantics with respect to the traditional semantics of logic programs [3].
In this paper, we present a general framework for classical two player games, but relaxing many traditional restrictions: infinite plays are allowed, values are no longer required to be totally ordered, and propagation functions can be chosen from a wide set of candidates. In this setting, the formal definition of the value of a game is given.
Then, we focus on the problem of *efficiently computing* the value of the game, by using the Alpha-Beta *algorithm*, not just on the existence of the value as formally defined.
Surprisingly enough, we can show that the Alpha-Beta algorithm computes the correct value of the game under a few general assumptions on the domain of values, thus greatly broadening its applicability: it can be, for example, used as a computational engine for logic programming.
But we do not content ourselves with proving correctness of Alpha-Beta on games whose value domain is more abstract than the usual integers and reals; we go much further by introducing the notions necessary to deal with potentially infinite games, and we prove that Alpha-Beta is (partially) correct on potentially infinite games: whenever it terminates, it computes the value of the game. This step takes us out of the usual domain of combinatorial game theory, where game trees can be huge but not infinite, and this is, to our best knowledge, the first correctness proof in this setting.
Then, we present an application to logic programming, and exhibit an example to show how Alpha-Beta can give us a significant gain in performance, or even terminate where other engines may loop. Finally, we conclude with a selection of future directions for research and application.

2 An abstract theory of two players combinatorics games

In this section, we introduce our formal framework for two player games, together with some fundamental notions, like that of a game-tree and the value of a finite game [14, 15, 6]. We then present the notion of an *approximation* of a game value, as is found in the theory of combinatorial games like chess or go, which are too big to be fully developed, and use it as a key notion to extend the framework to deal with infinite games too.

2.1 The rules of the game

A two player game is the simplest game in combinatorial Game Theory: we find two players, Player and Opponent, each opposed to the other, that play in turn one after the other and that must behave *rationally* (i.e. their move are deterministic, and they both seek to win). The game is assumed to be finite, and once one knows the terminal position in a play, the gain (or loss) of each player is known also; besides, what one looses, the other wins, so there always is precisely one winner.

A game is given once one knows its "syntax", that is the set of all possible plays, and its "semantics", that is the *value* of each of these plays (what Player gains or looses towards Opponent). We will formalize each of these aspects in turn, starting here from the "syntax".

2.2 Basic definitions: syntax

There are two ways of knowing all possible plays, either by giving them extensively as a set, an approach quite inadequate to handle real-world games where this set can be enormous, even if finite; or by giving a set of "positions" and "rules" that allow to produce all possible plays (like in chess or go) [12].
We will take in what follows this second approach: the following definitions are essentially the traditional ones.

Definition 1 (Syntax of a game). *The syntax of a game is formally defined as a tuple*

$$G = (WPOS,\ BPOS,\ IPOS\ ,\ \rightarrow_{\mathcal{R}})$$

Here $WPOS$ is the set of Player position, while $BPOS$ is the set of opponent positions and we write POS for the disjoint sum $WPOS \oplus BPOS$, and π for a generic position in POS. The third component, $IPOS \subseteq POS$ is the set of initial positions in the game. Finally, $\rightarrow_{\mathcal{R}} \subseteq POS \times POS$ is a *locally finite* (i.e. only a finite number of pairs in the relation can share their first component) transition relation that represents all the possible moves in a play as transitions between positions.
We require that the moves in the game are *alternating*, that is to say that whenever $(\pi, \pi') \in \rightarrow_{\mathcal{R}}$ we have that if π is a player position, then π' is an opponent position and vice-versa.
Defining $\rightarrow_{\mathcal{R}}$ is equivalent to defining a function $moves : POS \rightarrow 2^{POS}$, whose value is $moves(\pi) = \{\pi' \mid \pi \rightarrow_{\mathcal{R}} \pi'\}$ and that explicitly gives the possible moves out of a given position.

Once the syntax of a game is known, we have all the necessary information to determine when a game is finished (we have reached a terminal position, or not).

Definition 2 (Terminal, non terminal positions). *Given a game G, we identify the following derived notions*

terminal positions $TPOS = \{\pi \in POS \mid not\, \exists \pi' \; such\, that\, \pi \rightarrow_{\mathcal{R}} \pi'\}$
player terminal positions $WTPOS = TPOS \cap WPOS$
opponent terminal positions $BTPOS = TPOS \cap BPOS$
non terminal positions $NTPOS = \{\pi \in POS \mid \exists \pi' \; such\, that\, \pi \rightarrow_{\mathcal{R}} \pi'\}$
player non terminal positions $WNTPOS = NTPOS \cap WPOS$
opponent non terminal positions $BNTPOS = NTPOS \cap BPOS$

Finally, all the possible plays starting from a given initial position can be represented as a tree, known as a *game tree.*

Definition 3 (Game tree [14]). *A game tree Γ_π, for an initial position π, is a tree having positions as nodes, and representing all possible plays starting at π in the traditional way. Since a game tree can be infinite, it should be formally defined as a fix-point of a suitable monotone function, but we do not enter into the details here. Since the $\rightarrow_{\mathcal{R}}$ relation is locally finite, the game tree is finitely branching. A game is* finite *if its game tree is, infinite otherwise.*

2.3 Basic definitions: semantics

It is now time to turn to the essential aspect of a game in classical game theory: its *value*. To each terminal position, which is reached when a play is complete, is associated a *gain* for Player, taken out of some domain D (traditionally, the integers), given by an evaluation function h. Of course, this gain for Player is actually a *loss* for opponent, and given a set of possible moves, Player chooses the move that maximizes its gain, while Opponent chooses the move that minimizes its loss; this rational choice can be abstracted by two functions $\uparrow$ and $\downarrow$ on the domain of values. These functions usually operate on the *set* of possible choices to provide a value, but we prefer to see the choices presented orderly as a tuple, not a set, and we require that the result of the choice function is independent of the order. All these elements give us the semantics part of the game.

Definition 4 (Evaluation structure). *An* evaluation structure *$\mathcal{E}$ is a tuple*

$$(D\,,\,\uparrow,\downarrow\,,\,h)$$

*where D is the domain of values, $\uparrow: D^n \rightarrow D$ and $\downarrow: D^n \rightarrow D$ are functions of all finite arities $n \geq 1$ materializing the rational choices of Player and Opponent, and $h : TPOS \rightarrow D$ is the evaluation function giving the value (*gain*) of each terminal position. We require the $\uparrow$ and $\downarrow$ functions to be order-independent, that is, for any permutation $\sigma : n \rightarrow n$, $\uparrow(a_1, \ldots, a_n) = \uparrow(a_{\sigma 1}, \ldots, a_{\sigma n})$ (and similarly for $\downarrow$).*

Remark 1. The requirement that the choice-functions be order-independent is not necessary for the proofs, that go through seamlessly without it. But the traditional framework of combinatorial games always enforces this condition, and we keep it here to make the presentation of the result more intuitive.

Given an evaluation structure, it is possible to compute the *value* of any finite game (this definition mirrors the traditional one).

Definition 5 (Value of a finite game (Minimax propagation)). *Given a game G and an* evaluation structure, *$\mathcal{E}$ the function $Val : POS \rightarrow D$ defined as follows associates to each position in G its* value*:*

$$
\begin{aligned}
Val\,\pi &= h(\pi) && if\,\pi \in TPOS \\
Val\,\pi &= \mathop{\uparrow}_{\pi' \in moves(\pi)} Val\,\pi' && if\,\pi \in WNTPOS \\
Val\,\pi &= \mathop{\downarrow}_{\pi' \in moves(\pi)} Val\,\pi' && if\,\pi \in BNTPOS
\end{aligned}
$$

2.4 From huge to infinite games

Many finite games, like chess and go and unlike tic-tac-toe, are so huge that computing their value is not feasible. This is why one needs sometimes to try to compute an approximation of the value of a game from a given position. For that, we simply stop exploring the huge tree at some internal nodes, whose value is arbitrarily provided by some heuristic function. We will see that, while heuristics are simply useful to approximate the value of huge finite games, they are essential to *define* the value of an infinite game.
But to compute approximations, one needs to be able to compare values, that is, in what follows we assume that D is actually equipped with a partial order relation $\leq$. Also, for the approximations to be useful, it is necessary that the choice functions $\uparrow$ and $\downarrow$ be monotone w.r.t. the product partial order induced on D^n by $\leq$.
In what follows, we will always assume this monotonicity, which is always satisfied by the max and min functions used for traditional combinatorial games.

Heuristics and approximations A heuristic function is just a function of type $NTPOS \rightarrow D$ assigning arbitrary values to nonterminal positions in a game, but only some heuristics are interesting, and one usually distinguish between optimistic and pessimistic heuristics according to the ability of the heuristic to provide an approximation greater of, or inferior to, the actual value.

Definition 6 (Admissible heuristics). *A heuristic function $\varphi : NTPOS \rightarrow D$ is an* admissible pessimistic heuristic *(resp.* admissible optimistic heuristics *) for a finite game iff $\forall \pi \in NTPOS.\ \varphi(\pi) \leq Val\,\pi \qquad (resp.\ \geq)$.*

Unfortunately, determining if a heuristic is admissible can be quite hard, and another definition can be more useful in practice: we say a heuristic is monotone if the approximation it provides are consistent among themselves.

Definition 7 (Monotone heuristics). *A heuristic function* $\varphi : NTPOS \to D$ *is* monotone pessimistic *(resp.* monotone optimistic*) iff :*

$$\forall \pi \in NTPOS. \quad \begin{cases} \varphi(\pi) \leq h(\pi') \ \ \forall \pi' \, s.t. \, \pi \to_{\mathcal{R}} \pi' \, and \, \pi' \in TPOS & (resp. \geq) \\ \varphi(\pi) \leq \varphi(\pi') \ \forall \pi' \, s.t. \, \pi \to_{\mathcal{R}} \pi' \, and \, \pi' \in NTPOS & (resp. \geq) \end{cases}$$

Once we have a heuristic, we can build out of our evaluation structure a structure useful to compute approximations.

Definition 8 (Approximation structure). *A* pessimistic approximation structure *(resp.* optimistic*) is a tuple*

$$(D \ , \ \leq , \ \uparrow , \ \downarrow , \ h \ , \ \varphi)$$

where $(D \ , \ \uparrow \ , \ \downarrow \ , \ h)$ *is an evaluation structure, and such that*

- $\uparrow$ *and* $\downarrow$ *are monotone w.r.t. the product order induced on* D^n *by* $\leq$
- *the function* $\varphi' : POS \to D$ *defined as* $\varphi' = \varphi \oplus h$ *(i.e. the disjunctive union of relations* $\varphi : NTPOS \to D$ *and* $h : TPOS \to D$*) satisfies:*

$$\varphi'(\pi) \leq \underset{\pi' \in moves(\pi)}{\uparrow} \varphi'(\pi') \qquad if \, \pi \in WNTPOS \qquad (resp. \geq)$$

$$\varphi'(\pi) \leq \underset{\pi' \in moves(\pi)}{\downarrow} \varphi'(\pi') \qquad if \, \pi \in BNTPOS \qquad (resp. \geq)$$

If D contains a minimal element $\perp$, the heuristic $\lambda\pi.\perp$ will give us a *canonical* pessimistic approximation structure. Similarly, if D contains a maximal element $\top$, the heuristic $\lambda\pi.\top$ will give us a *canonical* optimistic approximation structure.

The monotonicity property allows us to easily prove the following result.

Proposition 1 (Monotonicity and approximations).
If $(D, \leq)$ *is a* distributive lattice*, and we take* $\uparrow = \vee$ *(the sup) and* $\downarrow = \wedge$ *(the inf) on* D*, then any* monotone pessimistic heuristics *(resp.* optimistic*) gives raise to a* pessimistic approximation structure *(resp.* optimistic*).*

Once we have an approximation structure at hand, we can compute an approximation of the value of a game, by cutting the tree branches at some internal nodes, obtaining another (smaller) tree, and computing the value of this cut tree. Of course, the approximation thus computed depends on where the cut actually take place, so an approximation structure really gives rise to a whole set of approximations. This can be put more formally as follows:

Definition 9 (Set of approximations of a game ($SetOfVal_\varphi$)).
Given a game G *and an evaluation structure* $(D \ , \ \uparrow \ , \downarrow \ , \ h)$*, and heuristic* $\varphi : NTPOS \to D$*, we define the approximation function for the game* G*, relative to* φ*, written* $SetOfVal_\varphi : POS \to 2^D$*, as the smallest function* $S : POS \to 2^D$ *(w.r.t. to the point-wise partial order on* $POS \to 2^D$ *derived from the partial order* $\subseteq$ *in* 2^D *) such that:*

1. $S(\pi)$ *contains* $h(\pi)$ *if* $\pi \in TPOS$
2. $S(\pi)$ *contains the element* $\varphi(\pi)$ *if* $\pi \in NTPOS$
3. $S(\pi)$ *contains the element* $\overset{n}{\underset{i=1}{\uparrow}} v_i$ *if* $\pi \in WNTPOS$ *, where* $moves(\pi) = \{\pi_1, \pi_2, \ldots, \pi_n\}$ *and* $v_i \in S(\pi_i)$
4. $S(\pi)$ *contains the element* $\overset{n}{\underset{i=1}{\downarrow}} v_i$ *if* $\pi \in BNTPOS$ *, where* $moves(\pi) = \{\pi_1, \pi_2, \ldots, \pi_n\}$ *and* $v_i \in S(\pi_i)$

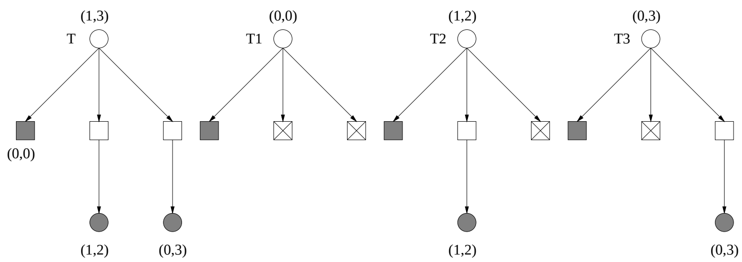
We remark here that $SetOfVal_{\varphi}(\pi)$ is well defined for all positions π as its value is the least fixed point of a monotone function over the complete lattice 2^D, which always exists due to Knaster-Tarski's fix-point theorem.
Actually, this really defines a function $SetOfVal : (NTPOS \rightarrow D) \rightarrow POS \rightarrow 2^D$, which we will apply to monotone heuristic functions in order to obtain a set of approximations having a reasonable algebraic structure.

Proposition 2 (Algebraic structure of the monotone approximations). *If* $(D\ ,\ \leq\ ,\ \uparrow\ ,\ \downarrow\ ,\ h\ ,\ \varphi)$ *is a* pessimistic approximation structure *(resp.* optimistic*), then for all position* $\pi \in POS$ *the set* $SetOfVal_{\varphi}(\pi) \in 2^D$ *is an* upper-semi-lattice *(i.e.* $\forall x, y \in S. x \vee y \in S$*) (resp.* lower-semi-lattice*).*
Moreover, if the game is finite*, then* $SetOfVal_{\varphi}(\pi)$ *contains its upper (resp lower) limit* $Val\,\pi$*.*

To put it in other terms, for finite games, given any monotone pessimistic heuristic h_{pess} and any monotone optimistic approximation h_{opt}, we have that

$$SetOfVal_{h_{pess}}(\pi) \cap SetOfVal_{h_{opt}}(\pi) = Val\,\pi$$

Example 1. In the following picture, we give the game tree of a finite game (on the left), and three smaller game trees obtained by cutting the full tree at some positions. The evaluation domain D is $Nat \times Nat$ with $\leq$ the product order and we use the trivial heuristic $h_{pess} = \lambda\pi.(0,0)$. Each tree is labelled with its valuation (the values of the trees on the right approximations in $SetOfVal_{h_{pess}}(\pi)$: notice that the set of approximations is not totally ordered. Terminal nodes are grey, no terminal are white; player nodes are circles, opponent nodes are squares, and cut nodes are crossed.



2.5 Infinite games

The framework set up to approximate the value of huge games can be used as is to handle infinite games too. Indeed the $SetOfVal$ function, defined as we did, is well defined both on finite *and* infinite games. This allows us to formally define two canonical approximations of the value of an infinite game, as partial functions assigning to each position the limit of its approximations, if it exists.

Definition 10 (Limit values of a game, value). *If* $(D\ ,\ \leq\ ,\ \uparrow\ ,\ \downarrow\ ,\ h_{pess}\ ,\ \varphi)$ *is a* pessimistic *approximation structure, then*

$$Val_{h_{pess}}\pi \doteq \sup\ SetOfVal_{h_{pess}}(\pi)$$

If $(D\ ,\ \leq\ ,\ \uparrow\ ,\ \downarrow\ ,\ h_{opt}\ ,\ \varphi)$ *is an* optimistic *approximation structure, then*

$$Val_{h_{opt}}\pi \doteq \inf\ SetOfVal_{h_{opt}}(\pi)$$

where we write $x \doteq y$ *for "*x *is equal to* y *if* y *defined".*
If they coincide, that defines the value $Val\ \pi$ of the infinite game *on* π*.*

Notice that $Val_{h_{pess}}$ is defined everywhere if the domain is a complete partial order (CPO), and $Val_{h_{opt}}$ is defined everywhere if if the domain is a co-complete partial order (co-CPO). Also, on finite games both approximations coincide with the value of the finite game.

Proposition 3 (Relating approximations). *Given a pessimistic approximation structure and an optimistic approximation structure sharing the same evaluation structure (hence different only for the heuristic functions,* h_{pess} *and* h_{opt}*), such that* $h_{pess}(\pi) \leq h_{opt}(\pi)$ *for all non terminal positions* π*, then we have that*

$$h_{pess}(\pi) \leq y \qquad \forall y \in SetOfVal_{h_{opt}}(\pi) \quad ; \quad h_{opt}(\pi) \geq x \qquad \forall x \in SetOfVal_{h_{pess}}(\pi)$$

for all non terminal positions π*.*

Proof. We give the proof of the first inequality by induction on the definition of $y \in SetOfVal_{h_{opt}}$. The second inequality is proved similarly.

- Base case: $y = h_{opt}(\pi)$ and $h_{pess}(\pi) \leq y$ by hypothesis
- Inductive step: suppose $\pi \in WNTPOS$ (the case $\pi \in BNTPOS$ is proved similarly, using the monotonicity of $\downarrow$); we have that

$$\begin{aligned}
h_{pess}(\pi) &\leq \mathop{\uparrow}_{i=1}^{n} h_{pess}(\pi_i) && \text{because } (\uparrow, \downarrow, h_{pess}) \text{ is an approximation structure} \\
&\leq \mathop{\uparrow}_{i=1}^{n} y_i && \text{by induction hypothesis and monotonicity of } \uparrow \\
&= y
\end{aligned}$$

Corollary 1 (Finite computations for infinite games). *Under the hypothesis of the previous proposition, we only have two possible cases*

1. $SetOfVal_{h_{pess}}(\pi) \cap SetOfVal_{h_{opt}}(\pi) = \{v\}$ *where* $v = Val\,\pi$
2. $SetOfVal_{h_{pess}}(\pi) \cap SetOfVal_{h_{opt}}(\pi) = \emptyset$

The first case gives us a sufficient condition to stop the computation of the approximations on an infinite game tree: as soon as the intersection is non empty, we know we have the value of the game, without needing to fully compute the set of approximations.

We will use these properties in our analysis of the Alpha-Beta algorithm on infinite games.

3 Correctness of the Alpha-Beta algorithm

We turn now to the central result of the paper: the correctness proof of the Alpha-Beta algorithm on approximation structures that are distributive lattices.
Let us first recall the definition of the Alpha-Beta algorithm, extended with heuristic functions (see for example [14] for an excellent introduction to Alpha-Beta), where the α and β parameters provide a lowest and an upper bound on the value that we want the algorithm to provide us with.

Definition 11. *Sequential Alpha-Beta algorithm The Alpha-Beta algorithm is defined, in pseudo-language, as follows*

```
function AlphaBeta ( π : Pos ; α ,β : D ): D
begin
if π ∈ TPOS then return(h(π));
if π ∈ WNTPOS and moves(π) = {π_1,...,π_n} then begin
        v := α ↑ h_pess(π);
        i := 1;
        while (not v ≥ β) and (i ≤ n) do begin
                  v := v ↑ AlphaBeta(π_i, v, β);
                  i := i + 1;
                  end;
        end;
if π ∈ BNTPOS and moves(π) = {π_1,...,π_n} then begin
        v := β ↓ h_opt(π);
        i := 1;
        while (not v ≤ α) and (i ≤ n) do begin
                  v := v ↓ AlphaBeta(π_i, α ,v) ;
                  i := i + 1 ;
                  end;
        end;
return( v );
end;
```

We will prove in the following a key result of *relative* correctness of AlphaBeta: if the algorithm is called with an α and a β parameters, it will never return anything outside the interval α, β, so we can only expect to prove that the value it computes is correct *up to* the interval we gave. But this result is a sort of induction loading needed in the proof: the interesting point is the corollary: when we call it with $\alpha = \bot$ and $\beta = \top$, it gives *the* correct result whenever it terminates.

Let us first establish a simple property of distributive lattices

Lemma 1 (Insertion). *Let* $(D, \leq)$ *be a* distributive lattice *and* $\uparrow = \vee$ *(the* sup*)* $\downarrow = \wedge$ *(the* inf*) on* D*. Then for all* α, β, x *we have that* $\alpha \uparrow [\beta \downarrow x] = \alpha \uparrow [\beta \downarrow (\alpha \uparrow x)]$ *and* $\beta \downarrow [\alpha \uparrow x] = \beta \downarrow [\alpha \uparrow (\beta \downarrow x)]$

Proof.

$$\begin{array}{rcll} \alpha \uparrow [\beta \downarrow (\alpha \uparrow x)] & = & \alpha \uparrow [(\beta \downarrow \alpha) \uparrow (\beta \downarrow x)] & \text{by distributivity} \\ & = & \alpha \uparrow (\beta \downarrow x) & \text{because } (\beta \downarrow \alpha) \leq \alpha \end{array}$$

The second equation is proved similarly.

Definition 12 (Equality modulo $\alpha\beta$)). *We will write* $AlphaBeta(\pi, \alpha, \beta) =_{\alpha\beta} z$*, where* $z \in D$*, as an abbreviation of one of the following equations over* D*, according to the type of the position* π*:*

$$\begin{array}{ll} \alpha \uparrow [\beta \downarrow AlphaBeta(\pi, \alpha, \beta)] = \alpha \uparrow [\beta \downarrow z] & if\, \pi \in WPOS \\ \beta \downarrow [\alpha \uparrow AlphaBeta(\pi, \alpha, \beta)] = \beta \downarrow [\alpha \uparrow z] & if\, \pi \in BPOS \end{array}$$

Theorem 1 (Relative correctness of the $AlphaBeta$ method)). *Let* $(D, \leq)$ *be a* distributive lattice *and suppose to apply the method* $AlphaBeta$ *with* $\uparrow = \vee$ *(the* sup*) and* $\downarrow = \wedge$ *(the* inf*) of* D*, and with a couple of* monotone heuristics h_{pess} *and* h_{opt}*, respectively* pessimistic *and* optimistic *and such that* $h_{pess}(\pi) \leq h_{opt}(\pi)$ *for all position* $\pi \in NTPOS$*. Then, for all position* $\pi \in POS$*, and for all* $\alpha, \beta \in D$*, if the function* $AlphaBeta(\pi, \alpha, \beta)$ *terminates, then there exist* $x \in SetOfVal_{pess}(\pi)$ *and* $y \in SetOfVal_{opt}(\pi)$ *such that:*

$$\begin{cases} AlphaBeta(\pi, \alpha, \beta) =_{\alpha\beta} x \\ AlphaBeta(\pi, \alpha, \beta) =_{\alpha\beta} y \end{cases}$$

Proof. This is a long induction on the structure of the finite part of the tree visited by the algorithm whenever it terminates. Full details are given in the appendix.

Corollary 2 (Correctness of the $AlphaBeta$ method)). *In the hypothesis of the theorem 1, suppose also that* D *contains a minimal element* $\bot$ *and a maximal element* $\top$*. Then, if the method* $AlphaBeta(\pi, \alpha, \beta)$ *terminates returning the value* v *then*

$$v = Val_{h_{pess}}(\pi) = Val_{h_{opt}}(\pi) = Val\,\pi$$

4 An application to logic programming

We believe that, while the core result of the paper is really the proof of correctness we provided, it would not be satisfactory to conclude it without providing a significant example of application outside the scope of traditional game-playing like chess. So we turn now to our favorite example, that was also the starting point of [5]: the world of logic programming. In what follows, we radically improve w.r.t. [5] by actually treating *constraint* logic programming (that includes traditional logic programming via the special case of Herbrand constraints).
We suppose that the constraints C come equipped with an intersection operator $\wedge$ (for Herbrand terms, this is the usual most general unifier for substitutions), and a disjunction operator $\vee$. We also suppose that the relation of logical implication of constraints $\Rightarrow$ is a partial order on C, and we assume the empty constraint $true$, and the never satisfied constraint $false$.
We now give the syntax and semantics of the game of a constraint logic program, but we need to assume, due to lack of space, familiarity with constraint logic programming (CLP) (see [8] for an introduction).

4.1 Syntax of the game

Definition 13 (Positions). *The set of positions $\pi \in POS$ is defined as follows*

$$WPOS \ni \pi ::= (A, c)$$
$$BPOS \ni \pi ::= [G, c]$$

where G is a positive (conjunctive) goal, A is a positive atom and c is a (conjunctive) constraint in C.

We follow the traditional notation in game theory that uses circles (here, parentheses) for player positions, and squares (here square parentheses) for opponent positions.

Definition 14 (Initial Positions). *The game starts from an opponent position having an empty constraint:*

$$IPOS \ni \pi ::= [G, true]$$

We now define the transition relation by giving explicitly the *moves* function.

Definition 15 (Rules of a CLP game, $\rightarrow_{\mathcal{CLP}}$). *The possible moves, that define the transition relation $\rightarrow_{\mathcal{CLP}}$ for the CLP game for a given CLP program P, are as follows:*

player moves *in a player position (A, c), player can choose any (renaming of) a rule $A \leftarrow c' \mid A_1, A_2, ..., A_n$ of P, s.t. $c \wedge c'$ is satisfiable in the constraint algebra of C, and reach an opponent position $[(A_1, A_2, ..., A_n), c \wedge c']$*

opponent moves *in an opponent position $[(A_1, A_2, ..., A_n), c]$, opponent can choose any atom A_i and reach a player position (A_i, c).*

4.2 Semantics of a CLP(C) game.

The game of a CLP program can easily be infinite, so we need an approximation structure and some heuristic functions to obtain an approximation of the game value. The values we have to handle are "disjunctions" of constraints, represented as sets of constraints, hence subsets of the powerset of C. Such disjunctions denotes a solution space in any model of C, and then we say that a set of constraint is greater than another when its denotation is bigger (covers) the denotation of the other. More formally

Definition 16 (The partially ordered domain of values). *The evaluation domain is $D = 2^C$, the powerset of the constraints. We equip D with a partial order $\leq$ which is the* covering extension *[7] of the partial order induced on the constraint by the logical implication of constraints:*

$$d_1 \leq d_2 \text{ iff } \forall c_1 \in d_1 \exists e_1 \dots e_n \in d_2 \text{ t.q. } c_1 \Rightarrow \bigvee_{i=1,\dots,n} e_i$$

Definition 17 (Evaluation of terminal positions). *The evaluation function on terminal nodes is defined as*

$$\begin{cases} h\,(A, c) = \emptyset \\ h\,[\Diamond, c] = c \end{cases}$$

where $\Diamond$ is the empty goal.

Player takes the set theoretic union of values, while opponent takes the intersection of the denotations of the constraints, more formally

Definition 18 (Player and opponent choice functions). *The functions $\uparrow, \downarrow: (2^C)^n \to 2^C$, are defined as $\uparrow = \cup$ and $\downarrow = \wedge$ where $\wedge$ is defined as $d_1 \wedge \dots \wedge d_n = \{c_1 \wedge \dots \wedge c_n \mid c_1 \in d_1, \dots, c_n \in d_n\}$. Both functions are monotone w.r.t. the order on D.*

We define now two heuristics as follows

Definition 19 (Heuristics). *For any nonterminal position $\pi = (A, c)$ or $\pi = [G, c]$*

$$\begin{cases} h_{pess}(\pi) = \emptyset \\ h_{opt}(\pi) = \{c\} \end{cases}$$

In this game, if $\pi \to_{\mathcal{CLP}} \pi'$, then the constraint component c' of π' is either the constraint component c of π, or $c \wedge c''$ for some of the constraint c''. In both case, $c' \Rightarrow c$, hence $h_{opt}(\pi) \geq h_{opt}(\pi')$. Also, trivially $h_{pess}(\pi) \leq h_{pess}(\pi')$, so both heuristics are monotone, and we have at hand both an optimistic and a pessimistic approximation structure for the game of $CLP(C)$.

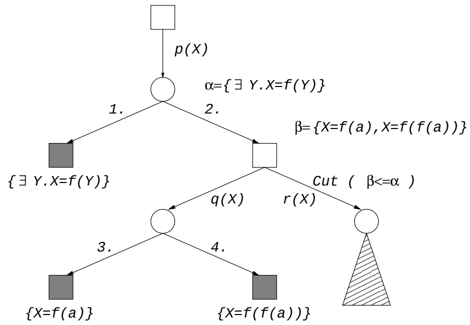
This general construction yields an interesting object when the evaluation domain 2^C satisfies the condition for applying Alpha-Beta; this depends in general on C, but there are many examples that satisfy the requirement, like the case of *ex-equations* Eqn (see [10]), for which it is easy to verify that $\uparrow = \cup$ and $\downarrow = most\ general\ unifier\ (for\ ex-equations)$ are the sup and inf on the domain 2^{Eqn} and that they satisfy the hypothesis of the Alpha-Beta theorem, that can then be used to compute the answers.

Let us see how Alpha-Beta works on an example

Example 2. Consider the (classical) logic program

```
1. p(f(Y)).
2. p(X) :- q(X),r(X).
3. q(f(a)).
4. q(f(f(a))).
5. r(X) :- ...
```

Here, if we consider the goal $p(X)$, once we use rule 1, and we get the answer $\exists Y.X = f(Y)$, it is useless to try rule 2, as rule 2 finds answers for $q(X)$ of the shape $\{X = f(a), X = f(f(a))\}$. Since the most general unifier gives always a result which is less than its arguments, we cannot find anything better than $\{X = f(a), X = f(f(a))\}$. Without even knowing the answer for $r(X)$, we can give up the search right there. This is what Alpha-Beta does, by cutting the subtree rooted at $r(X)$ and exiting with the answer $\{\exists Y.X = f(Y)\}$:



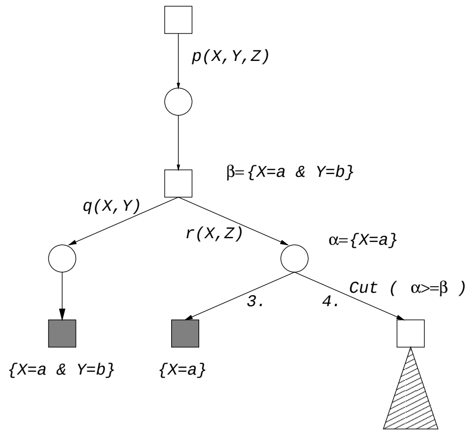
We provide a second example where the cut takes place at a player node.

Example 3. Consider the program

```
1. p(X,Y,Z) :- q(X,Y),r(X,Z).
2. q(a,b).
3. r(a,Z).
4. r(X,Z) :- ...
```

Starting with a goal $p(X, Y, Z)$, we use rule 1, then we evaluate the sub-goal $q(X, Y)$, that forces $X = a$, and we use rule 3 to solve $r(X, Z)$. After that, we have a solution where $X = a$ and we force no constraint on Z. That means that there is no interest in looking for other solutions to $r(X, Z)$, as $X = a$ is forced by $q(X, Y)$. Again, this is what AlphaBeta really does, with a cut condition

$\alpha \geq \beta$:



5 Conclusions and future work

We have introduced a formal framework to extend the traditional notion of value of a combinatorial game in order to deal with games whose value can now range on arbitrary partially ordered domains. Also, by means of the notion of approximation structure (pessimistic and optimistic), we can now give a precise meaning to the value of a potentially infinite game. In this general framework, we have generalized the well-known Alpha-Beta algorithm to compute values in an arbitrary distributive lattice, and formally proved that this algorithm is partially correct in general on infinite games, and correct on finite ones. This is, in our opinion, a significant achievement, as the algorithm can now be confidently applied to many different fields, quite far apart from the traditional fields of application of combinatorial game theory. As an example, we have provided an instantiation of the game framework for (constraint) logic programming, that includes the minimax characterization of the semantics of logic programming formally introduced by the authors in [5], and we have shown how the Alpha-Beta method can be used to compute (efficiently) the set of answers for a given goal.
Another very promising direction of research is in the field of abstract interpretation, where the very same algorithm could be used to compute the abstract value of a program, by just changing the value domain, thus providing a more efficient means of performing static analysis.
Finally, we believe that this approach can be interesting also for the study of logic programming in the presence of negative atoms in the clauses.

References

1. S. Abramsky and R. Jagadeesan. Games and full completeness for multiplicative linear logic. *The Journal of Symbolic Logic*, 59(2):543–574, 1994.

2. A. Blass. A game semantics for linear logic. *Annals of Pure and Applied Logic*, 56:pages 183–220, 1992.
3. A. Bossi, M. Gabbrielli, G. Levi, and M. Martelli. The s-semantics approach: Theory and applications. *Journal of Logic Programming*, 19-20:149–197, 1994.
4. P. Curien and H. Herbelin. Computing with abstract bohm trees. 1996.
5. R. Di Cosmo, J.-V. Loddo, and S. Nicolet. A game semantics foundation for logic programming. In C. Palamidessi, H. Glaser, and K. Meinke, editors, *PLILP'98*, volume 1490 of *Lecture Notes in Computer Science*, pages 355–373, 1998.
6. G. Diderich. A survey on minimax trees and associated algorithms. *Minimax and its Applications, Kluwer Academic Publishers*, 1995.
7. F. Fages. Constructive negation by pruning. *Journal of Logic Proigramming*, 2(32):85–118, 1997.
8. J. Jaffar and M. J. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19/20:503–581, 1994.
9. A. Joyal. Free lattices, communication and money games. *Proceedings of the 10th International Congress of Logic, Methodology, and Philosophy of Science*, 1995.
10. N. J. Kim Marriot, Harald Søndergaard. Denotational abstract interpretation of logic programs. *ACM Transaction on Programming Languages and Systems*, 16(3):607–648, 1994.
11. F. Lamarche. Game semantics for full propositional linear logic. *Proceedings of the 10th Annual IEEE Symposium on Logic in Computer Science*, pages 464–473, 1995.
12. G. Owen. *Game Theory*. W.B. Saunders, 1968.
13. V. D. P. Baillot and T. Ehrhard. Believe it or not, AJM's games model is a model of classical linear logic. *Proceedings of the 12th Annual IEEE Symposium on Logic in Computer Science*, pages pages 68–75, 1997.
14. J. Pearl. *Heuristics. Intelligent Search Strategies for Computer Problem Solving.* Addison Wesley, 1984.
15. A. Plaat. *Research Re: search & Re-search.* PhD thesis, Tinbergen Institute and Department of Computer Science, Erasmus University Rotterdam, 1996.
16. J. von Neumann. Zur Theorie der Gesellschaftsspiele. *Mathaematische Annalen*, (100):195–320, 1928.

A Proof of the relative correctness theorem

Here is the proof of Theorem 1

Proof. If the method terminates this means that it computes the value visiting only a finite part of the game tree (possibly infinite). So we can proceed by induction on the structure of this finite part. We give the proof of the case $\pi \in WPOS$, i.e. the player positions. For opponent positions the proof is similar.

Base case. The method visit an unique node and return the value $v = AlphaBeta(\pi, \alpha, \beta) \in D$. There are only two possibilities:

– The node is terminal: $\pi \in TPOS$

Implies $v = h(\pi)$ so $x = h(\pi) \in SetOfVal_{pess}(\pi)$ and $y = h(\pi) \in SetOfVal_{opt}(\pi)$ verify trivially the equality modulo $\alpha\beta$ with v.

– The node was immediately cut because $v \geq \beta$

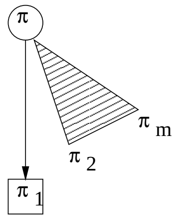
Let $x = h_{pess}(\pi) \in SetOfVal_{pess}(\pi)$ and $y = h_{opt}(\pi) \in SetOfVal_{opt}(\pi)$. Then

$$\begin{array}{rcll} \alpha \uparrow [\beta \downarrow v] & = & \alpha \uparrow [\beta \downarrow (\alpha \uparrow h_{pess}(\pi))] & because\, v = \alpha \uparrow h_{pess}(\pi) \\ & = & \alpha \uparrow [\beta \downarrow h_{pess}(\pi)] & by\, lemma\ 1 \\ & = & \alpha \uparrow [\beta \downarrow x] & \end{array}$$

So $v =_{\alpha\beta} x$. Moreover, since $\alpha \uparrow h_{pess}(\pi) \geq \beta$ we have also $\alpha \uparrow [\beta \downarrow v] = \alpha \uparrow \beta$. Using the hypothesis $h_{pess}(\pi) \leq h_{opt}(\pi)$, we obtain, by monotonicity of the operator $\uparrow$, that also $\alpha \uparrow h_{opt}(\pi) \geq \beta$ so $\alpha \uparrow [\beta \downarrow y] = \alpha \uparrow [\beta \downarrow (\alpha \uparrow h_{opt}(\pi))] = \alpha \uparrow \beta$, that means $v =_{\alpha\beta} y$.

Structural induction. In the position π, we suppose that the algorithm has analyzed n moves of the player among the m possibles moves (with $m \geq n$) before returning the result. We now proceed by a sub-induction on n.

– Case $n = 1$



Implies $v = AlphaBeta(\pi, \alpha, \beta) = \alpha \uparrow h_{pess}(\pi) \uparrow AlphaBeta(\pi_1, v_0, \beta)$ $(*)$ where $v_0 = \alpha \uparrow h_{pess}(\pi)$. By inductive hypothesis on structure of tree we have that exist $x_1 \in SetOfVal_{pess}(\pi_1)$ and $y_1 \in SetOfVal_{opt}(\pi_1)$ such that:

$$\begin{cases} AlphaBeta(\pi_1, v_0, \beta) =_{v_0\beta} x_1 \\ AlphaBeta(\pi_1, v_0, \beta) =_{v_0\beta} y_1 \end{cases}$$

Since $\pi_1 \in BPOS$ is an opponent position, that means:

$$\beta \downarrow [v_0 \uparrow AlphaBeta(\pi_1, v_0, \beta)] = \beta \downarrow [v_0 \uparrow x_1] = \beta \downarrow [v_0 \uparrow y_1] \qquad (**)$$

We distinguish now between two reasons of termination of the algorithm: the case $n = m = 1$ (all the moves have been evaluated) and the case $n = 1 < m$ (some moves have been cut).

– Sub-case $n = m = 1$

We define the approximations: $x =\uparrow x_1 = x_1$ and $y =\uparrow y_1 = y_1$. By definition $x \in SetOfVal_{pess}(\pi)$ et $y \in SetOfVal_{opt}(\pi)$, so:

$$\begin{array}{rcll} \alpha \uparrow [\beta \downarrow v] & = & \alpha \uparrow [\beta \downarrow (v_0 \uparrow AlphaBeta(\pi_1, v_0, \beta))] & by\, (*) \\ & = & \alpha \uparrow [\beta \downarrow (\alpha \uparrow h_{pess}(\pi) \uparrow x_1)] & by\, (**) \\ & = & \alpha \uparrow [\beta \downarrow (h_{pess}(\pi) \uparrow x)] & by\, Lemma\ 1\, and\, x = x_1 \\ & = & \alpha \uparrow [\beta \downarrow x] & because\, h_{pess}(\pi) \leq x \end{array}$$

The last fact in proof $h_{pess}(\pi) \leq x$ is insured because h_{pess} is a monotone pessimistic. With the same logical steps we prove $\alpha \uparrow [\beta \downarrow v] = \alpha \uparrow [\beta \downarrow y]$ where the necessary condition $h_{pess}(\pi) \leq y$ is insured by prop. 3 using the hypothesis $h_{pess}(\pi) \leq h_{opt}(\pi)$.

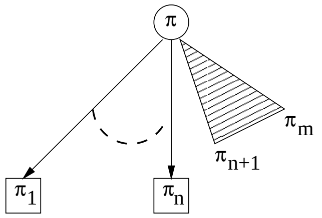
- Sub-case $n = 1 < m$

If the algorithm has cut the $m-1$ remaining moves, this means that the condition $v_0 \uparrow v_1 \geq \beta$, where $v_1 = AlphaBeta(\pi_1, v_0, \beta)$, was true. This implies $\alpha \uparrow [\beta \downarrow v] = \alpha \uparrow [\beta \downarrow (v_0 \uparrow v_1)] = \alpha \uparrow \beta$. We can define the approximations $x = x_1 \uparrow x_1'$ where $x_1' = h_{pess}(\pi_2) \uparrow \ldots \uparrow h_{pess}(\pi_m)$ and $y = y_1 \uparrow y_1'$ where $y_1' = h_{opt}(\pi_2) \uparrow \ldots \uparrow h_{opt}(\pi_m)$. By definition $x \in SetOfVal_{pess}(\pi)$ and $y \in SetOfVal_{opt}(\pi)$. Then

$$\begin{array}{rcll}
\alpha \uparrow [\beta \downarrow x] & = & \alpha \uparrow [\beta \downarrow (\alpha \uparrow x)] & by\, lemma\ 1 \\
& = & \alpha \uparrow [\beta \downarrow (\alpha \uparrow h_{pess}(\pi) \uparrow x)] & because\, h_{pess}(\pi) \leq x \\
& = & \alpha \uparrow [\beta \downarrow (v_0 \uparrow x_1 \uparrow x_1')] & by\, def.\, of\, v_0\, and\, x \\
& = & \alpha \uparrow [\beta \downarrow (v_0 \uparrow x_1)] \uparrow [\beta \downarrow x_1'] & by\, distributive\, law \\
& = & \alpha \uparrow [\beta \downarrow (v_0 \uparrow v_1)] \uparrow [\beta \downarrow x_1'] & by\, (**) \\
& = & \alpha \uparrow \beta \uparrow [\beta \downarrow x_1'] & because\, v_0 \uparrow v_1 \geq \beta \\
& = & \alpha \uparrow \beta & because\, \beta \downarrow x_1' \leq \beta
\end{array}$$

Using the prop. 3 we can infer $h_{pess}(\pi) \leq y$ and use it to prove $\alpha \uparrow [\beta \downarrow y] = \alpha \uparrow \beta$ with these same logical steps.

- Case $n > 1$



By definition of the method, if we consider a fake position π' with only the first $n-1$ moves of π, and another fake position π'' having only the nth move, we can write $v = AlphaBeta(\pi, \alpha, \beta) = AlphaBeta(\pi', \alpha, \beta) \uparrow AlphaBeta(\pi'', v_1, \beta)$ where $v_1 = AlphaBeta(\pi', \alpha, \beta)$, considering $h_{pess}(\pi') = h_{pess}(\pi'') = h_{pess}(\pi)$. Then, by inductive hypothesis on n there exists $x' \in SetOfVal_{pess}(\pi')$ and $y' \in SetOfVal_{opt}(\pi')$ such that

$$(\#) \begin{cases} AlphaBeta(\pi', \alpha, \beta) =_{\alpha\beta} x' \\ AlphaBeta(\pi', \alpha, \beta) =_{\alpha\beta} y' \end{cases}$$

and exist $x'' \in SetOfVal_{pess}(\pi'')$ and $y'' \in SetOfVal_{opt}(\pi'')$ such that

$$(\#\#) \begin{cases} AlphaBeta(\pi'', v_1, \beta) =_{v_1\beta} x'' \\ AlphaBeta(\pi'', v_1, \beta) =_{v_1\beta} y'' \end{cases}$$

Then

$$\begin{array}{rcl}
\alpha \uparrow [\beta \downarrow v] = & \alpha \uparrow [\beta \downarrow (v_1 \uparrow AlphaBeta(\pi'', v_1, \beta))] & by\, def.\, of\, v\, and\, v_1 \\
= & \alpha \uparrow [\beta \downarrow (v_1 \uparrow (\beta \downarrow AlphaBeta(\pi'', v_1, \beta)))] & by\, lemma\ 1 \\
= & \alpha \uparrow [\beta \downarrow (v_1 \uparrow (\beta \downarrow x''))] & by\ (\#\#) \\
= & \alpha \uparrow [\beta \downarrow (v_1 \uparrow x'')] & by\, lemma\ 1 \\
= & \alpha \uparrow [\beta \downarrow v_1] \uparrow [\beta \downarrow x''] & by\, distributive\, law \\
= & \alpha \uparrow [\beta \downarrow x'] \uparrow [\beta \downarrow x''] & by\ (\#) \\
= & \alpha \uparrow [\beta \downarrow (x' \uparrow x'')] & by\, distributive\, law
\end{array}$$

We have proved $\alpha \uparrow [\beta \downarrow v] = \alpha \uparrow [\beta \downarrow (x' \uparrow x'')]$ $(*)$ and, with the same arguments, we can prove $\alpha \uparrow [\beta \downarrow v] = \alpha \uparrow [\beta \downarrow (y' \uparrow y'')]$ $(**)$. We distinguish now between two reasons of termination of the algorithm: the case $n = m$ (all the moves have been evaluated) and the case $n < m$ (some moves have been cut).

- Sub-case $1 < n = m$

The moves of π are exactly the set of moves of π' together with the moves of π''. Then, defining the approximations: $x = x' \uparrow x''$ and $y = y' \uparrow y''$ we have $x \in SetOfVal_{pess}(\pi)$ and $y \in SetOfVal_{opt}(\pi)$. Hence conditions $(*)$ and $(**)$ give our thesis.

- Sub-case $1 < n < m$

If the algorithm has cut the $m - n$ remaining moves, this means that condition $v_0 \uparrow v_1 \geq \beta$ where $v_0 = AlphaBeta(\pi', \alpha, \beta)$ was true. Then $\alpha \uparrow [\beta \downarrow v] = \alpha \uparrow [\beta \downarrow (v_0 \uparrow v_1)] = \alpha \uparrow \beta$ $(***)$. We define the approximations: $x = x' \uparrow x'' \uparrow x'''$ where $x''' = h_{pess}(\pi_{n+1}) \uparrow \ldots \uparrow h_{pess}(\pi_m)$ and $y = y' \uparrow y'' \uparrow y'''$ where $y''' = h_{opt}(\pi_{n+1}) \uparrow \ldots \uparrow h_{opt}(\pi_m)$. By definition $x \in SetOfVal_{pess}(\pi)$ and $y \in SetOfVal_{opt}(\pi)$. Then

$$\begin{array}{rcl}
\alpha \uparrow [\beta \downarrow x] = & \alpha \uparrow [\beta \downarrow (x' \uparrow x'' \uparrow x''')] & by\, def.\, of\, x \\
= & \alpha \uparrow [\beta \downarrow (x' \uparrow x'')] \uparrow [\beta \downarrow x'''] & by\, associative\, and\, distributive\, laws \\
= & \alpha \uparrow [\beta \downarrow v] \uparrow [\beta \downarrow x'''] & by\ (*) \\
= & \alpha \uparrow \beta \uparrow [\beta \downarrow x'''] & by\ (***) \\
= & \alpha \uparrow \beta & because\, \beta \downarrow x''' \leq \beta
\end{array}$$

With the same logical steps, simply using $(**)$ instead of $(*)$, we obtain $\alpha \uparrow [\beta \downarrow y] = \alpha \uparrow \beta$.