An Interactive Approach to Control in Proof Search: Preliminary Results

Alexis Saurin*

Abstract. Logic programming, as well as functional programming, benefitted from a strong connection with logic and proof theory which allowed to provide formal tools to analyze programs through logical tools. However, this logical analysis fails when control operators are concerned.
A informal way to understand this is that the role of those operators is mostly to prune the search space while the search space does not admit a good logical representation in proof-theoretical frameworks. Indeed, proof-search is concerned with partial proofs, proofs being constructed as well as failed searches whereas proo-theory (and in particular sequent calculus) is deals with completed proofs.
Considering the strong relations between proof theory and programming languages theory, one may expect that when making progress in the understanding of the essence of proofs one shall also get better tools to understand computation. Ludics has been introduced about a decade ago by Girard as an interactive approach to logic. In a previous work, we used this framework to introduce a paradigm of interactive proof-search, where proof search in not specified by a sequent to be proved, but rather by a set of tests that shall be passed.
In this short abstract, we discuss how interactive proof-search allow for an interactive analysis of control (or pruning) mechanisms in logic programming.

Keywords: Ludics, Interaction, Game Semantics, Logic Programming, Proof Search, Control, Pruning operators.

1 Introduction: An Interactive Approach to Proof-Search

Proof Theory and Computation. Recent developments in proof theory have led to major advances in the theory of programming languages. The modelling of computation using proofs impacted deeply the foundational studies of programming languages as well as many of their practical aspects. Declarative programming languages have been related mainly in two ways to the mathematical theory of proofs. On the one hand, the "computation as proof normalization" paradigm provided a foundation for functional programming languages through the use of the well-known Curry-Howard correspondence [15] relating typed λ-calculus with intuitionistic natural deduction proofs and reductions of a λ-term with cut-elimination in NJ. On the other hand the "computation as proof search" paradigm stands as a foundation for logic programming.

While functional programming found strong theoretical foundations and very powerful tools for formal analysis thanks to the study of cut-elimination procedures in proof systems such as natural deduction or sequent calculus as well as the development of type theory and its relationships with functional programming languages, logic programming has been built on the paradigm of proof search: the dynamics of a program computation is viewed as the search for a proof in some deductive system.

Computation as proof search. The proof search paradigm was at first founded on the resolution method: computation corresponded to the search for a resolution for first-order Horn clauses. But this approach was uneasy to extend to larger fragments or richer logics.

Later, the use of sequent calculus allowed to overcome this limitation: the introduction of uniform proofs and abstract logic programming languages [23], the introduction of the very well structured sequent calculus for linear logic [12], LL, and the discovery of the focalization property [2] in LL allowed to extend the computational paradigm of proof search to larger fragments of logic that resolution method for Horn Clauses

* Universit degli Studi di Torino, *saurin@di.unito.it*

(Hereditary Harrop formulas for instance) and to richer logics (higher-order logics, linear logic, ...) and to benefit from the structure of sequent calculus. These various aspects enriched the dynamics of proof search.

In the uniform-proof model for instance, a program will be modelled as a sequent $\mathcal{P} \vdash G$ where $\mathcal{P}$ represents the logic program and G is the computation goal. Computation then consists in a search for a cut-free proof of $\mathcal{P} \vdash G$ directed by the goal G, that is the logic program $\mathcal{P}$ can play a rule in the search, through backchaining, only when the goal G has been reduced to an atomic formula.

In addition to the interest of this approach from the fundamental point of view, it allowed to enrich logic programming languages with numerous additional programming primitives while treating them logically (higher-order programming, modules, resource management, concurrent primitives...). Nevertheless some essentials constructions of logic programming languages could not be dealt with logically, in particular when we are concerned with the control of computation and the pruning of the search tree (backtracking, intelligent backtracking, `cut` predicate...) [25, 26]. As a consequence, some parts of the languages do not have a very well established semantics from the logical point of view and they cannot be analyzed in a satisfactory way using the rich logical methods that proof search paradigm provides.

Recalling Kowalski's motto *"Algorithm = Logic + Control"* [17], one can notice that with the development of real implementations of logic programming languages, the correspondence turned out to become much closer to:

$$\textit{Algorithm} = \textit{Logic} + \textit{Control} + \textit{I/O} + \textit{Data Abstraction} + \textit{Modules} + \ldots$$

since those useful programming primitives are needed. One of the long-standing research directions on proof search is to treat the extra-logical primitives in a logical way in order to get closer to the "ideal" correspondence: *"Algorithm = Logic"* [21]. While Kowalski had in mind the importance of separating logic and control in the specification of an algorithm[1], recent advances in proof theory made logic a setting much richer than only an appropriate formalism to speak about truth: proofs are structured objects and proof theory can specify computation in a much richer way than just ensuring correctness with respect to the specification of an algorithm. We believe that the evolved structure of computation is to be found in logic itself. The extension of the proof search paradigm to broader settings allowed to capture some of these components in the logical component (modules, data abstraction, ...) but the control part is still lies outside an satisfying logical treatment.

We can draw a useful and enlightening comparison with functional programming: the extension of the Curry-Howard correspondence to classical logic allowed to capture logically several control operators that were used in practice (like `call/cc`) thanks to typing rules for those operators [14] or thanks to extensions of λ-calculus such as $\lambda\mu$-calculus [27]. In this way, the connection between logic and computation could be extended further, however, corresponding extensions could not be achieved in logic programming in such a satisfactory way.

Why is it so difficult to capture control logically in logic programming? Most probably, the answer to this question is to be found in a sort of mismatch between sequent calculus proof theory and logic programming. While in sequent calculus, the objects that we manipulate are proofs (and the meta-theorems that are demonstrated deal with proofs), the process of searching for and constructing proofs does not deal with proofs until the computation is completed. Instead, the objects of proof search are partial proofs (or open proofs) which may end up not leading to a proof at all but to a failure of the search. Such failed proofs are not part of the proof theory of sequent calculus. Thus, the very dynamics of computation stands outside the theory we work with. In addition, searching for proofs consists essentially in failing often and seldom succeeding.

Moreover, while one often considers that the *state* of the computation is represented by a sequent, the essential element for proof search does not lie in the sequents themselves but in the inference rules which are applied to sequents: the sequent should rather be considered as a constraint on the *actions* that can be performed to progress towards completing the search. Thus, the very dynamics of computation does not fit well with the logical theory we work with.

[1] That is separating the question of *what* is computed by a program from the question of *how* it is computed.

Ludics, a theory of interactions. We noticed earlier the strong relationships between logic and computation. We can thus expect that new developments in proof theory will provide new tools, new methods and maybe new paradigms on the computer science side as well as they could raise new questions. Such impacts can be expected in particular when these new results shed light on concepts that had not been satisfyingly treated before. We think that the recently introduced Ludics [13] should be regarded as such.

In recent years, interaction has become a crucial concept of the theory of computation. The introduction of linear logic [12] surely is partly responsible for this but game theoretical interpretations of logic and computation have been studied prior to linear logic. One can for instance refer to Curien's work on Concrete Data Structure, abstract and environment machines and more recently Abstract Böhm Trees [3, 5–7]. Ludics certainly goes further in this direction. Girard introduced Ludics [13] in which unfinished proofs are given a clear status being at the heart of this theory of interaction. Ludics is a logical theory that attempts to overcome the distinction between syntax and semantics by considering that interaction comes first and by building syntax and semantics afterwards, thanks to interaction. Ludics objects, *designs*, can be seen as intermediate objects "between" syntax and semantics: they are both an abstraction of linear sequent calculus proofs and a concrete version of innocent strategies in a game semantics. Ludics is founded on many concepts of proof theory, especially of linear logic sequent calculus [12] and especially the fundamental result of Focalization [2] that allows for synthetic connectives to be defined. Ludics has lots of connections with game semantics [1, 16] as well since the interaction process in Ludics can be seen as a play [10]. Many concepts have been inspired by proof search [22]. We shall introduce the main definitions of Ludics in section **??** after we provided some more detailed intuitions on Ludics concepts and technics.

We think that Ludics provides lots of tools that should be useful for logic programming especially in order to develop a study of logic programming that would benefit from the concepts and tools from interactive theories, such as an explicit treatment of the computational environment which is uniform to the computational objects [8].

Games and Logic Programming. Game-theoretic approaches to logic programming are fairly natural and however not much developped. Van Emden was the first to notice the connections between logic programming computations and two-person games with $\alpha\beta$-algorithm [31], which was later studied in more details by Loddo et al. [4, 20, 19]. In particular, they extend this analysis to contraint logic programming (CLP). Pym and Ritter [28, 29] proposed a game semantics for uniform proofs and backtracking by relating intuitionnistic and classical provability while the author, Miller and Delande [24, 9] developed a neutral approach to proofs and refutations based on games which was inspired by Prolog search engine [24]. More recently, Galanaki et al. [11] generalized van Emden's games for logic programs with (well-founded) negation.

Searching for proofs interactively The sequent $\mathcal{P} \vdash G$ is the current state of the computation but it is also a way to constrain the future of the computation. In the same way, restrictions on the logical rules that are allowed (like in linear logic) or proof strategies also impose constraints on proof search. But all this is implicit and not done explicitly. In particular, it is fairly difficult to analyze these constraints in proof theory itself, even though that would be useful since important programming primitives precisely rely on these constraints. The interactive approach to proof search we are investigating precisely makes explicit the constraints on proof search: instead of building a proof depending on a given sequent, we shall consider building a proof that shall pass some tests, that shall be opposed to attempts to refute it. The tests will have the form of (para)proofs and thus will be built in the same system as the one in which we are searching for proofs. We propose a computational setting which would correspond to the following (see figure 1):

- We are willing to search for a proof $\mathfrak{D}$ of $\vdash A$;
- Formula A is actually given as a set of tests (which is the set of counter-arguments that one should be able to answer): the tests that shall be successfully passed by the proof we are constructing: $\mathfrak{E}_1, \ldots, \mathfrak{E}_n$;
- Proof construction shall proceed by consensus with the tests: $\mathfrak{D}$ can be extended with rule R only if the extended object interacts well with the tests;
- After some interactions, we may have an object that cannot be extended any more. Either the construction is terminated because $\mathfrak{D}$ cannot pass every test or because all the tests are satisfied and no more

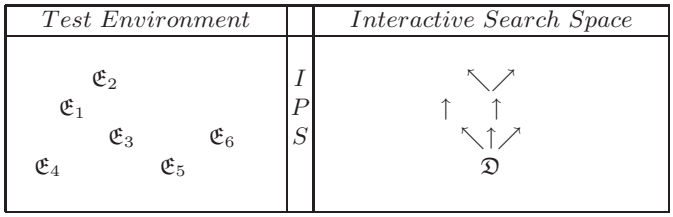


Fig. 1. Interactive Proof Search.

constraint applies to $\mathfrak{D}$ so that there is no need to extend it further. In the first case, we have a failure while in the later case we have a success

– One interesting point with this interactive approach lies in the fact that the setting is symmetrical: $\mathfrak{D}$ is tested by $\mathfrak{E}_i$ but it is also a test for the $\mathfrak{E}_i$s. In particular, even failures are interesting and useful objects. If the result of a computation is a failure $\mathfrak{D}^{\maltese}$, we may be willing to try another search. Indeed, maybe at some point we chose a wrong way to extend $\mathfrak{D}$ and that caused the failure whereas another way to build $\mathfrak{D}$ could have led to a win. There is a standard way to backtrack, that is erase some part of $\mathfrak{D}^{\maltese}$ and try some other construction. But since we are in an interactive setting, there is another option: we can try to use $\mathfrak{D}^{\maltese}$ in order to provide more tests: $\mathfrak{E}_j^{\mathfrak{D}^{\maltese}}$ that will constrain the search to look for a proof that shall be different from the failure $\mathfrak{D}^{\maltese}$.

The present paper will present an ongoing work on providing an interactive study of control, investigating what kind of control can be expressed via interactive means. For details related to the previous works on interactive proof-search which we shall only partially recall in the following, the reader is invited to read our recent paper on the topic [30].

2 Motivations and intuitions for Ludics

In this section, we describe informally Ludics, with an emphasis towards connections with logic programming, in order to motivate the main logical concepts on which this interactive theory is built. For more details on ludics, the reader may see section 3 of [30].

Monism. Ludics has been introduced by Girard [13] as an interactive theory that aims at overcoming the traditional distinction between syntax and semantics by considering that interaction should come first and logic shall be reconstructed afterwards: designs can be viewed both as an abstraction of multiplicative additive linear logic (MALL) sequent proofs *(syntactical viewpoint)* and as a concrete presentation of game semantics innocent strategies [10] *(semantical viewpoint)*.

Thus, one does not start from syntactic objects to which is assigned a semantic interpretation, nor does one begin with a semantic space for which we design an adequate language: There is an interaction theory, objects interacting with each other, properties of which are defined interactively.

Focalization. Andreoli's Focalization [2] is a fundamental property of linear logic which impacted deeply studies on proof search. Moreover it is the root for a polarized approach to logic [18] and allows to define synthetic connectives and synthetic rules (which are clusters of connectives or rules of the same polarity) in MALL. MALL formulas are defined as follows:

$$\begin{array}{rrll} F ::= & a & \mid F \otimes F \mid F \oplus F & \mid \mathbf{1} \mid \mathbf{0} \quad \text{(positive formulas)} \\ & a^{\perp} & \mid F \parr F \mid F \,\&\, F & \mid \perp \mid \top \quad \text{(negative formulas)} \end{array}$$

MALL connectives and logical inference rules (see figure 2) can be classified in two sets of connectives: positive connectives ($\otimes, \oplus, \mathbf{1}, \mathbf{0}$) and negative ones ($\parr, \&, \perp, \top$). When searching for a proof, one alternates

$$\frac{}{\vdash A, A^\perp}\ ini \qquad \frac{\vdash \Gamma, A \quad \vdash \Delta, A^\perp}{\vdash \Gamma, \Delta}\ cut$$

$$\frac{\vdash \Gamma, A \quad \vdash \Delta, B}{\vdash \Gamma, \Delta, A \otimes B}\ \otimes \qquad \frac{\vdash \Gamma, A_0}{\vdash \Gamma, A_0 \oplus A_1}\ \oplus_0 \qquad \frac{\vdash \Gamma, A_1}{\vdash \Gamma, A_0 \oplus A_1}\ \oplus_1 \qquad \frac{}{\vdash \mathbf{1}}\ \mathbf{1}$$

$$\frac{\vdash \Gamma, A, B}{\vdash \Gamma, A \parr B}\ \parr \qquad \frac{\vdash \Gamma, A \quad \vdash \Gamma, B}{\vdash \Gamma, A \,\&\, B}\ \& \qquad \frac{}{\vdash \Gamma, \top}\ \top \qquad \frac{\vdash \Gamma}{\vdash \Gamma, \bot}\ \bot$$

Fig. 2. *MALL* sequent calculus.

between two phases, a negative (or asynchronous) phase and a positive (or synchronous) phase. During the negative phase, we are certain not to lose provability: if a conclusion sequent containing negative formulas is provable, then all the premisses that are obtained by applying a negative rule to the conclusion sequent are provable. On the other hand, during the positive phase provability can be lost: there are choices to make (which rule to use for the $\oplus$ connective, how to split the context for the $\otimes$ rule...) and making a wrong choice we may end up not finding a proof even if the sequent we started with was provable. Thus there is clearly an active phase (positive) and a passive phase (negative); when searching for a proof, one alternates between those two phases.

Focalization is thus a key step towards a game theoretic interpretation of sequent proofs:

- The negative phase is opponent's turn to play (the asynchronous rule gathers all needed information for the player to react to this move);
- The positive phase is player's turn: after a move of the opponent, the player decides what she will play following what her strategy (that is the proof rules at her disposal) tells her to play.

An interesting invariant with proofs in the hypersequentialized calculus for MALL is that there is at most one negative formula in a sequent. Notice that Focalization can be extended to full linear logic [2].

Proof Normalization. The cut elimination procedure reflects this game-theoretic interpretation: a conversion step corresponds to the selection, by the positive rule, of a continuation for the normalization (*e.g.* selection of one of the &-premises by a $\oplus$-rule during cut-elimination). But there is still a problem for an interactive interpretation to hold: we cannot find a proof for both A and $A^\perp$. Notice that if there cannot be proofs for both a formula and its negation, it is perfectly legal to attempt to prove both A and $A^\perp$ [24, 9].

The only important restriction is that at most one of the two formulas can be proved (and maybe none). However, if the proof search fails, the partial object that we have obtained can be used in an interaction with proof attempts of the negation, except where the failure was encountered (here normalization is still undefined). A failed attempt to prove A is a tree with some open branches. Let us add a new rule, the ***daimon***, to mark the fact that the search for a proof has been stopped:

$$\frac{}{\vdash \Gamma}\ \maltese.$$

What is this sequent $\vdash \Gamma$ where we stopped? It would not be very reasonnable to stop if $\vdash \Gamma$ contains a negative formula since decomposing this formula costs nothing. As a consequence, we restrict application of $\maltese$ to sequents that made of only positive formulas. In particular, we now have paraproofs for any sequents, even for the empty sequent, $\vdash$.

Winning and loosing. The normalization between two paraproofs is a ***process through which they test each other***. The one that is caught using $\maltese$ is considered as the loser of the play and the play ends there. Notice that this normalization process is an exploration of the two paraproofs: the cut visits some parts of the paraproofs. When $\maltese$ is reached, the paraproofs are said to be ***orthogonal***. A paraproof that wins an interaction may still contain itself a daimon: it is simply not part of this precise interaction, but would be detected by some other interactions (due to ludics separation theorem [13]).

$$\frac{\vdash \Gamma, A_\xi \quad \vdash \Delta, A_\xi^\perp}{\vdash \Gamma, \Delta}\ cut \qquad \boxed{\frac{}{\vdash\ \Gamma}\ \Pisymbol{psy}{128}}$$

$$\frac{}{\vdash \mathbf{1}_\xi}\ \mathbf{1} \qquad \frac{\vdash \Gamma, A_{\xi 0} \quad \vdash \Delta, B_{\xi 1}}{\vdash \Gamma, \Delta, A_{\xi 0} \otimes_\xi B_{\xi 1}}\ \otimes \qquad \frac{\vdash \Gamma, A_{\xi i}}{\vdash \Gamma, A_{\xi 0} \oplus_\xi A_{\xi 1}}\ \oplus_i,\ i \in \{0,1\}$$

$$\frac{}{\vdash \Gamma, \top_\xi}\ \top \qquad \frac{\vdash \Gamma}{\vdash \Gamma, \perp_\xi}\ \perp \qquad \frac{\vdash \Gamma, A_{\xi 0}, B_{\xi 1}}{\vdash \Gamma, A_{\xi 0} \parr_\xi B_{\xi 1}}\ \parr \qquad \frac{\vdash \Gamma, A_{\xi 0} \quad \vdash \Gamma, B_{\xi 1}}{\vdash \Gamma, A_{\xi 0} \,\&_\xi\, B_{\xi 1}}\ \&$$

In $\Pisymbol{psy}{128}$, Γ contains no negative formula.

Fig. 3. $MALL\Pisymbol{psy}{128}$ sequent calculus.

Locations. Whereas in functional programming it matters to know if the types of functions and arguments match (and thus to know the whole structure of the type formulas), it is not relevant for proof search to know the complete structure of formulas to be proved: we only need to know enough to choose the next rule. In Ludics, we use addresses (or ***loci***): a formula is dealt with through its address ξ and an inference rule R on ξ creates ***subloci*** $(\xi i, \xi j, \dots)$ which refer to *where* the subformulas are (not *what* they are).

Behaviours. A provable formula may be interpreted as the set of its proofs or rather its paraproofs. Actually things are even more drastic in Ludics: formulas are defined interactively by a standard technique of biorthogonality closure which defines the ***behaviours***. Given a paraproof Π in behaviour $\mathbf{A}$ and a paraproof Π' in $\mathbf{A}^\perp$, a part of Π can be explored by normalization with Π'. Sometimes Π is entirely visited by some Π' but usually, there are parts of Π that cannot be explored, whatever $\Pi' \in \mathbf{A}^\perp$ you test it with. However, a class of paraproofs which is highly interesting from our interactive perspective is the class of paraproofs that can be completely visited during normalization with elements of $\mathbf{A}^\perp$, they are said to be ***material***. They are the most interesting elements in A since they can be completely characterized interactively.

3 Searching for proofs interactively in MALL.

In this section we discuss interactive proof-search (or proof-search by cut-elimination) in $MALL\Pisymbol{psy}{128}$. The purpose of this section is mostly to serve as a concrete introduction to interactive proof search that we will not detail in this short abstract (again, the reader shall find detail on interactive proof-search in Ludics in [30]).

If we want to search for proofs by interaction, we need to have enough proof objects to interact with, as noticed in section **??**, but we never have a proof for A and $A^\perp$ for any formula A in $MALL$. This means that we need to extend $MALL$ in order to have more proofs and provable formulas. In the following, we consider $MALL\Pisymbol{psy}{128}$ proofs which are built from unit-only $MALL$-formulas. Moreover we index $MALL\Pisymbol{psy}{128}$ formulas in order to identify there occurrences more easily, anticipating on the use of addresses. Here, indices are finite sequences of 0s and 1s expliciting the subformula relation ($\xi ::= \langle\rangle \mid \xi 0 \mid \xi 1$):

$$\begin{array}{rll} F_\xi ::= F_{\xi 0} \otimes_\xi F_{\xi 1} \mid F_{\xi 0} \oplus_\xi F_{\xi 1} \mid \mathbf{1}_\xi \mid \mathbf{0}_\xi & & \text{(pos)} \\ \quad \mid\ F_{\xi 0} \parr_\xi F_{\xi 1} \mid F_{\xi 0} \,\&_\xi\, F_{\xi 1} \mid \perp_\xi \mid \top_\xi & & \text{(neg)} \end{array}$$

by adding the $\Pisymbol{psy}{128}$ rule to $MALL$ proof system, obtaining unit-only $MALL\Pisymbol{psy}{128}$ proof system which is shown in Figure 3. A $MALL\Pisymbol{psy}{128}$ proof will be referred to as a ***paraproof***.

3.1 An Example of IPS in MALL✠.

Let us consider the two following paraproofs, $\mathfrak{D}_0$ and $\mathfrak{D}_1$, of sequent $\vdash \mathbf{1}_0 \,\&_{\langle\rangle}\, (\bot_{10} \oplus_1 \bot_{11})$:

$$\mathfrak{D}_i = \dfrac{\dfrac{}{\vdash \mathbf{1}_0}\,\mathbf{1} \quad \dfrac{\dfrac{\dfrac{}{\vdash}\,✠}{\vdash \bot_{1i}}\,\bot}{\vdash \bot_{10} \oplus_1 \bot_{11}}\,\oplus_i}{\vdash \mathbf{1}_0 \,\&_{\langle\rangle}\, (\bot_{10} \oplus_1 \bot_{11})}\,\& \qquad \text{with } i \in \{0,1\}$$

We will look for a paraproof $\mathfrak{D}$ that would *pass both tests* $\mathfrak{D}_0$ and $\mathfrak{D}_1$. $\mathfrak{D}$ shall be a MALL✠proof of sequent $\vdash \bot_0 \oplus_{\langle\rangle} (\mathbf{1}_{10} \,\&_1\, \mathbf{1}_{11})$ such that paraproofs built by cutting $\mathfrak{D}$ with any of the $\mathfrak{D}_i$s normalize:

$$\dfrac{\dfrac{\mathfrak{D}_i}{\vdash \mathbf{1}_0 \,\&_{\langle\rangle}\, (\bot_{10} \oplus_1 \bot_{11})} \quad \dfrac{\mathfrak{D}}{\vdash \bot_0 \oplus_{\langle\rangle} (\mathbf{1}_{10} \,\&_1\, \mathbf{1}_{11})}}{\vdash}\,cut \qquad \rightsquigarrow_{cut-elim} \qquad \dfrac{}{\vdash}\,✠$$

Since $\vdash \bot_0 \oplus_{\langle\rangle} (\mathbf{1}_{10} \,\&_1\, \mathbf{1}_{11})$ is a positive sequent, there is an obvious solution:

$$\mathfrak{D} = \dfrac{}{\vdash \bot_0 \oplus_{\langle\rangle} (\mathbf{1}_{10} \,\&_1\, \mathbf{1}_{11})}\,✠$$

But we may also search for an extension of the above paraproof that would possibly not use the ✠. We shall then chose an inference rule following the constraints which are imposed by the cut reduction. Those constraints on $\mathfrak{D}$ can be used as a guide to search for a paraproof on $\vdash \bot_0 \oplus_{\langle\rangle} (\mathbf{1}_{10} \,\&_1\, \mathbf{1}_{11})$. We end up with:

$$\mathfrak{D} = \dfrac{\dfrac{\dfrac{}{\vdash \mathbf{1}_{10}}\,\mathbf{1}^{\star_0} \quad \dfrac{}{\vdash \mathbf{1}_{11}}\,\mathbf{1}^{\star_1}}{\vdash \mathbf{1}_{10} \,\&_1\, \mathbf{1}_{11}}\,\&}{\vdash \bot_0 \oplus_{\langle\rangle} (\mathbf{1}_{10} \,\&_1\, \mathbf{1}_{11})}\,\oplus_1 \qquad \text{or} \qquad \mathfrak{D}' = \dfrac{\dfrac{\dfrac{}{\vdash}\,✠}{\vdash \bot_0}\,\bot}{\vdash \bot_0 \oplus_{\langle\rangle} (\mathbf{1}_{10} \,\&_1\, \mathbf{1}_{11})}\,\oplus_0$$

In $\mathfrak{D}$, the branch ending at $\star_i$ has been built interacting with the right premise of $\mathfrak{D}_i$. On the other hand $\mathfrak{D}'$ uses a ✠ rule and it cannot be developed anymore by interaction: it is a failure. During the construction of $\mathfrak{D}'$, the fact that $\mathfrak{D}'$ uses a ✠ during the interaction can be understood as the fact that both $\mathfrak{D}_i$s were able to find an error in $\mathfrak{D}'$, without letting $\mathfrak{D}'$ discover their logical leakages (the ✠ rule in the right premise).

3.2 Beyond MALL✠.

Since $\mathfrak{D}'$ is a failure, one might wish to avoid searching this paraproof. In order to do so, one can add paraproof $\mathfrak{D}_2$:

$$\mathfrak{D}_2 = \dfrac{\dfrac{}{\vdash \bot_{10} \oplus \bot_{11}}\,✠}{\vdash \mathbf{1}_0 \,\&_{\langle\rangle}\, (\bot_{10} \oplus_1 \bot_{11})}\,\&|_1$$

to the test environment. Indeed, an interactive search with $\mathfrak{D}_2$ would forbid the search of failure $\mathfrak{D}'$ by forcing the selection of $\oplus_1$.

On the other hand, one might wish to forbid search for $\mathfrak{D}$; this can be achieved by adding

$$\mathfrak{D}_3 = \dfrac{\dfrac{}{\vdash \mathbf{1}_0}\,✠}{\vdash \mathbf{1}_0 \,\&_{\langle\rangle}\, (\bot_{10} \oplus_1 \bot_{11})}\,\&|_0$$

to the set of tests instead of $\mathfrak{D}_2$. Indeed, if $\mathfrak{D}_3$ is in the normalization environment then the searched paraproof $\mathfrak{D}$ ***is forced*** to use $\oplus_0$ as a first rule, resulting in the construction of a failure paraproof.

$\mathfrak{D}_2$ and $\mathfrak{D}_3$ can thus be used to forbid some interactions to occur. But one shall actually notice that these two structures are not elements of $MALL✠$ since they use only *partial* & proof rules.

3.3 The Interactive Proof-search Algorithm on an example.

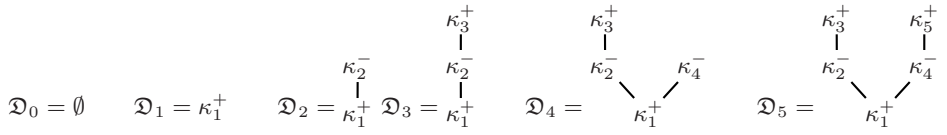
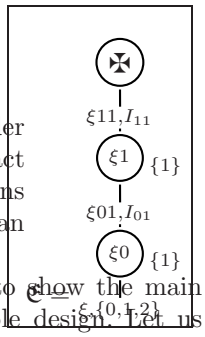


Fig. 4. Interactive search for $\mathfrak{D}$.



In this section, we outline an algorithm for Interactive Proof Search and refer the reader to [30] for more details and in particular for the definition and the study of the abstract machine, the Searching LAM (SLAM) which allows us to interactively build material designs in a behaviour generated by orthogonality to a set of tests. The reader will also find an example of an interactive search.

We will not do more than sketching how IPS works on a simple example in order to show the main structure of the search: consider the Interactive Proof Search driven by one very simple design. Let us proceed with an IPS with environment $\{\mathfrak{E}\}$ in order to build a design $\mathfrak{D}$.

0. To begin with, $\mathfrak{D}_0$ is empty and we have visited an empty path: $Path_0 = \epsilon$;

1. $\mathfrak{E}$ is a negative design so that it is a forest: it may begin with several negative actions on focus ξ, one of which shall be followed during a normalization process. Its initial actions are in $\text{Init}_{\mathfrak{E}} = \{(\xi, \{0,1,2\})^-\}$. We choose some action κ_1^- in $\text{Init}_{\mathfrak{E}}$ and add $\kappa_{1\nu}$ to the normalization path and κ_1^+ as the first action of $\mathfrak{D}$: $Path_1 = \langle(\xi, \{0,1,2\})\rangle = \langle\kappa_{1\nu}\rangle$;

2. Design $\mathfrak{D}$ in construction could have several negative actions above κ_1^+ but at this point, normalization would follow only one action which corresponds to the positive action after κ_1^- in $\mathfrak{E}$. This action is $\kappa_2^+ = (\xi 0, \{1\})^+$ and thus: $Path_2 = \langle\kappa_{1\nu}, \kappa_{2\nu}\rangle$;

3. In $\mathfrak{E}$, κ_2^+ is followed by actions in $\{(\xi 01, I_{01})^-\}$, we choose κ_3^- in this set and we extend $Path_2$ with $\kappa_{3\nu}$ and $\mathfrak{D}$ with κ_3^+: $Path_3 = \langle\kappa_{1\nu}, \kappa_{2\nu}, \kappa_{3\nu}\rangle$;

4. In $\mathfrak{E}$, κ_3^- is followed by the only action $\kappa_4^+ = (\xi 1, \{1\})^+$ and thus, $Path_3$ shall be extended with $\kappa_{4\nu}$. $\mathfrak{D}_4$ must contain κ_4^- in order to interact properly, but this negative action shall be placed right after the positive action justifying it. The branch leading to κ_4^- in $\mathfrak{D}$ is given by: $\ulcorner\kappa_{1\nu}, \kappa_{2\nu}, \kappa_{3\nu}, \kappa_{4\nu}\urcorner^- = \kappa_1^+, \kappa_4^-$ and thus: $Path_4 = \langle\kappa_{1\nu}, \kappa_{2\nu}, \kappa_{3\nu}, \kappa_{4\nu}\rangle$;

5. In $\mathfrak{E}$, $Succ(\kappa_4^+) = \{(\xi 11, I_{11})^-\}$. We extend the dispute: $Path_5 = \langle\kappa_{1\nu}, \ldots, \kappa_{5\nu}\rangle$ and we add κ_5^+;

6. In $\mathfrak{E}$, κ_5^- is followed by a unique action, $\kappa_6^+ = \maltese$. The normalization ends with $\mathfrak{E}$ using a $\maltese$ and the final dispute is $[\mathfrak{D} \rightleftharpoons \mathfrak{E}] = \langle\kappa_{1\nu}, \ldots, \kappa_{5\nu}, \maltese\rangle$.

$\rightarrow$ After the IPS process, we built a design $\mathfrak{D}$ on $\vdash \xi$ such that $[\![\mathfrak{D}, \mathfrak{E}]\!] = \maltese$ with the daimon caused by $\mathfrak{E}$.

4 Interactive Control

Control in interactive proof-search relies on the ability to enrich the test environment with new test by using information gathered during previous interactions. As a result, the test behaviours shall be bigger and the search behaviour (the search space) will be smaller: some parts of the search space will thus have been pruned. We first present the case of backtracking and then discuss other variants of updates.

4.1 Backtracking

We shall consider a final state $\mathcal{S}' = \langle(p_i{\cdot}\maltese \bullet \emptyset)_{i\in I} \mid \mathfrak{D}\rangle$ reached from an initial state $\mathcal{S} = \langle(\epsilon \bullet (\{\mathfrak{E}_j\})_{j\in J}) \mid \emptyset\rangle$. If $I \neq \emptyset$, then $\mathfrak{D}$ is a failure (it contains $\maltese$ at $\ulcorner p_i \cdot \maltese\urcorner^+$). One shall use those paths (or disputes) $p_i, i \in I$ to enrich the test environment with new designs (see figure 5).

Definition 1 ($\mathfrak{Test}(p)$). *If p a dispute generated by SLAM-n, $\mathfrak{Test}(p)$ is:*
(i) $\mathfrak{Test}(\epsilon) = \emptyset$*;*

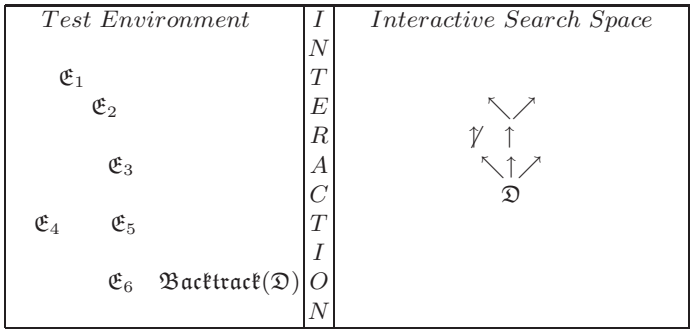


Fig. 5. Backtracking in IPS

(ii) $\mathfrak{Test}(\kappa) = \{\kappa^+\}$;
(iii) $\mathfrak{Test}(s \cdot \kappa \cdot \kappa') = \{\ulcorner s \cdot \kappa \cdot \kappa' \urcorner^+, \ulcorner s \cdot \kappa \urcorner^-\} \cup \mathfrak{Test}(s)$.

Proposition 1. *$\mathfrak{Test}(p_i), i \in I$ is a slice. Moreover, $\mathfrak{Test}(p_i)$ is the smallest design (when designs are considered as sets of chronicles) realizing interaction $p_i \cdot \maltese$ with the final design $\mathfrak{D}$.*

In order to have tests forbidding to reach the failure point, we shall both add and remove chronicles from $\mathfrak{Test}(p)$:

- $\mathfrak{Test}(p)$ does not allow enough other interaction paths. In particular, $\mathfrak{Test}(p)$ is a slice whereas the choices in the interactive search come from the fact that there are multiple negative actions on the same address. As a consequence, for each positive chronicle $\chi \in \mathfrak{Test}(p)$ but for $\ulcorner p \urcorner^+$ ending at a proper action $(\xi, I)^+$, for any $i \in I$ and $J \in \mathcal{P}_f(\omega)$, if $\chi \cdot (\xi i, J)^- \notin \mathfrak{Test}(p)$, then one shall extend χ with $(\xi i, J)^- \cdot \maltese$.
- $\mathfrak{Test}(p)$ allows to reach the $\maltese$ which is not satisfying. One shall shorten the chronicle $\ulcorner p \urcorner^+$ in such a way that it disables the visit of the branch causing the failure. Thus one shall remove the chronicle $\ulcorner p \urcorner^+$ as well as the negative chronicle it immediately extends: if $\ulcorner p \urcorner^+ = s \cdot \kappa_1^- \cdot \kappa_2^+$, then one removes $s \cdot \kappa_1^- \cdot \kappa_2^+$ and $s \cdot \kappa_1^-$ from $\mathfrak{Test}(p)$

Definition 2 ($\mathfrak{Backtrack}(p)$). *$\mathfrak{Backtrack}(p)$ is the smallest design such that:*
1. *$\mathfrak{Backtrack}(p)$ contains all positive chronicles of $\mathfrak{Test}(p)$ except $\ulcorner p \urcorner^+$;*
2. *if $\chi \in \mathfrak{Backtrack}(p)$ has last action $(\xi, I)^+$, then for any $i \in I$ and $J \in \mathcal{P}_f(\omega)$ such that $\chi \cdot (\xi i, J)^- \notin \mathfrak{Test}(p)$, one has $\chi \cdot (\xi i, J)^- \cdot \maltese \in \mathfrak{Backtrack}(p)$.*

Remark 1. In definition 2, $\mathfrak{Backtrack}(p)$ is drastically infinite because of all the $\chi \cdot (\xi i, J)^- \cdot \maltese$ that are added. However, by collecting information during the search (at step 1. of SLAM-n, when considering the set of initial positive actions in the environment), one may retain the needed actions and build a finite $\mathfrak{Backtrack}(p)$ if original test-environments are made of finitely branching designs.

4.2 A Classification of Interactive Control.

In the same spirit of backtrack, one may use the disputes $(p_i)_{i \in I}$ to build other designs to be added to the test environment before restarting the search and one may use ludics tools to analyze the effect of those new tests, this is what we briefly outline in an high-level way in this last paragraph.

An interactive control operator ϕ shall thus be a some function taking as inputs a test environment $\mathbf{E}$, a design $\mathfrak{D}$ and an interaction path $p_{\mathfrak{D}}$ and outputing a set of designs $\{\mathfrak{E}_i^{\phi,\mathfrak{D},p_{\mathfrak{D}}}, i \in I\}$ used to update $\mathbf{E}$: $\mathbf{E}$ becomes $\mathbf{E}' = \mathbf{E} \cup \{\mathfrak{E}_i^{\phi,\mathfrak{D},p_{\mathfrak{D}}}, i \in I\}$[2]

[2] Satisfying some additional conditions, for instance that material designs in $\mathbf{E}'^{\perp}$ shall be material in $\mathbf{E}^{\perp}$.

Consider a test environment $\mathbf{E} = \{\mathfrak{E}_i, i \in I\}$ and consider the successive answers $\langle \mathfrak{D}_0, \mathfrak{D}_1, \ldots, \mathfrak{D}_n, \ldots \rangle$ which are constructed by successive interactions using the backtracking as in the previous section, all of them being material designs in $\mathbf{E}^{\perp}$. There are several cases when applying a control operator ϕ after the search produced $\mathfrak{D}_0$. For instance, one may consider the set

$$J = \{i \in \omega, \mathfrak{D}_i \in |(\mathbf{E} \cup \{\mathfrak{E}_i^{\phi, \mathfrak{D}, p_{\mathfrak{D}}}, i \in I\})^{\perp}|\}$$

In the case J is infinite, then the role of the control operators was to skip some of the searched results (it may speed up the search for instance), whereas if J is finite then the control operators will allow to find new solutions (that were not reachable at first). Another analysis that can be looked for concerning an interactive control operator is whether or not there are winning designs among the designs indexed by elements of $I \setminus J$.

References

1. Samson Abramsky and Radha Jagadeesan. Games and full completeness for multiplicative linear logic. *J. of Symbolic Logic*, 59(2):543–574, 1994.
2. Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *J. of Logic and Computation*, 2(3):297–347, 1992.
3. G. Berry and Pierre-Louis Curien. Sequential algorithms on concrete data structures. *Theoretical Computer Science*, 20:265–321, 1982.
4. Roberto Di Cosmo, Jean-Vincent Loddo, and Stephane Nicolet. A game semantics foundation for logic programming (extended abstract). In *PLILP/ALP*, pages 355–373, 1998.
5. Pierre-Louis Curien. Abstract böhm trees. *Mathematical Structures in Computer Science*, 8(6):559–591, 1998.
6. Pierre-Louis Curien. Abstract machines, control, and sequents. In *Proceedings of APPSEM Summer School*, pages 123–136. Springer, September 2000.
7. Pierre-Louis Curien. Symmetry and interactivity in programming. *Bulletin of Symbolic Logic*, 9(2):169–180, 2003.
8. Pierre-Louis Curien and Hugo Herbelin. The duality of computation. In *ICFP*, pages 233–243, 2000.
9. Olivier Delande and Dale Miller. A neutral approach to proof and refutation in MALL. In F. Pfenning, editor, *23th Symp. on Logic in Computer Science*. IEEE Computer Society Press, 2008.
10. Claudia Faggian and Martin Hyland. Designs, disputes and strategies. In *CSL*, pages 442–457, 2002.
11. Chrysida Galanaki, Panos Rondogiannis, and William W. Wadge. An infinite-game semantics for well-founded negation in logic programming. *Annals of Pure and Applied Logic*, 151(2):70–88, 2008.
12. Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.
13. Jean-Yves Girard. Locus solum. *Mathematical Structures in Computer Science*, 11(3):301–506, June 2001.
14. Timothy Griffin. A formulae-as-types notion of control. In *POPL'90*, pages 47–58, 1990.
15. William A. Howard. The formulae-as-type notion of construction, 1969. In J. P. Seldin and R. Hindley, editors, *To H. B. Curry: Essays in Combinatory Logic, Lambda Calculus, and Formalism*, pages 479–490. Academic Press, New York, 1980.
16. J. M. E. Hyland and C.-H. L. Ong. On full abstraction for PCF: I. models, observables and the full abstraction problem, II. dialogue games and innocent strategies, III. A fully abstract and universal game model. *Information and Computation*, 163:285–408, 2000.
17. R. Kowalski. Algorithm = logic + control. *Communications of the Association for Computing Machinery*, 22:424–436, 1979.
18. Olivier Laurent. *Etude de la polarisation en logique*. Thèse de doctorat, Université Aix-Marseille II, March 2002.
19. Jean-Vincent Loddo. *Généralisation des Jeux Combinatoires et Applications aux Langages Logiques*. PhD thesis, Université Paris VII, 2002.
20. Jean-Vincent Loddo and Roberto Di Cosmo. Playing logic programs with the alpha-beta algorithm. In *LPAR*, pages 207–224, 2000.
21. Dale Miller. Sequent calculus and the specification of computation. In Ulrich Berger and Helmut Schwichtenberg, editors, *Computational Logic*, volume 165 of *Nato ASI Series*, pages 399–444. Springer, 1999.
22. Dale Miller. Overview of linear logic programming. In Thomas Ehrhard, Jean-Yves Girard, Paul Ruet, and Phil Scott, editors, *Linear Logic in Computer Science*, volume 316 of *London Mathematical Society Lecture Note*, pages 119–150. Cambridge University Press, 2004.
23. Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

24. Dale Miller and Alexis Saurin. A game semantics for proof search: Preliminary results. In *Proceedings of the Mathematical Foundations of Programming Semantics (MFPS05)*, number 155 in Electr. Notes Theor. Comput. Sci, pages 543–563, 2006.
25. Lee Naish. *Negation and Control in Prolog*, volume 238 of *Lecture Notes in Computer Science*. Springer, 1986.
26. Lee Naish. Pruning in logic programming. Technical Report 95/16, Department of Computer Science, University of Melbourne, Melbourne, Australia, june 1995.
27. Michel Parigot. $\lambda\mu$-calculus: an algorithmic interpretation of classical natural deduction. In *Proceedings of International Conference on Logic Programming and Automated Deduction*, volume 624 of *Lecture Notes in Computer Science*, pages 190–201. Springer, 1992.
28. David Pym and Eike Ritter. *Reductive Logic and Proof-search: proof theory, semantics, and control*, volume 45. Oxford Logic Guides, Oxford, 2004.
29. David Pym and Eike Ritter. A games semantics for reductive logic and proof-search. In Dan Ghica and Guy McCusker, editors, *GaLoP 2005: Games for Logic and Programming Languages*, pages 107–123, 2005.
30. Alexis Saurin. Towards ludics programming: Interactive proof search. In *24th International Conference on Logic Programming (ICLP2008)*, Lecture Notes in Computer Science. Springer, 2008.
31. Maarten H. van Emden. Quantitative deduction and its fixpoint theory. *Journal of Logic Programming*, 3(1):37–53, 1986.