Ludics Programming I:
Interactive Proof Search

Alexis Saurin
INRIA Futurs École Polytechnique
saurin@lix.polytechnique.fr

Abstract

Proof theory and Computation are research areas which have very strong relationships: new concepts in logic and proof theory often apply to the theory of programming languages. The use of proofs to model computation led to the modelling of two main programming paradigms which are functional programming and logic programming. While functional programming is based on proof normalization, logic programming is based on proof search. This approach has shown to be very successful by being able to capture many programming primitives logically. Nevertheless, important parts of real logic programming languages are still hardly understood from the logical point of view and it has been found very difficult to give a logical semantics to control primitives.

Girard introduced Ludics [12] as a new theory to study interaction. In Ludics, everything is built on interaction or in an interactive way.

In this paper, which is the first of a series investigating a new computational model for logic programming based on Ludics, namely computation as interactive proof search, we introduce the interactive proof search procedure and study some of its properties.

Keywords Ludics, Game Semantics, Logic Programming, Proof Search, Interaction, Proof Normalization.

1. Introduction.

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 issues. 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 to functional programming languages through the use of the well-known Curry-Howard isomorphism relating simply 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 formal analysis tools thanks to the study of cut-elimination procedures in proof systems such as natural deduction or sequent calculus and the development of type theory and its relationships with functional programming languages, logic programming has been built on the paradigm of proof search where the computation of a program 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 limit: the introduction of Uniform Proofs and Abstract Logic Programming Languages [16] and the discovery of the Focalization Property [2] in Linear Logic [11] allowed to extend the proof search paradigm to larger fragments (Hereditary Harrop formulas for instance) and to richer logics (higher-order logics, linear logic, ...) and to benefit from the geometrical properties of sequent calculus. In the uniform proofs model for instance, computation will be modelled as a search for a proof of a sequent $\mathcal{P} \vdash G$ where $\mathcal{P}$ represents the logic program and G the computation goal. The computation then proceeds as a search for a proof directed by the goal G, the logic program $\mathcal{P}$ being used through backchaining when the goal is an atomic formula.

In addition to the interest from the fundamental point of view, this approach 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 (backtracking, intelligent backtracking, `cut` predicate). As a consequence, some parts of the languages do not have a very well established semantics 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" [15], one can notice that with the development of real implementations of logic programming languages, the correspondence turned out to become much closer to: "Algorithm = Logic + Control + I/O + Data Abstraction + Modules + ..." since those useful programming primitives are needed. 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 there.

One of the long-standing research directions on proof search is to treat the extra-logical primitive in a logical way in order to get closer to the "ideal" correspondence: "Algorithm = Logic". 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 [13] or thanks to extensions of λ-calculus such as $\lambda\mu$-calculus [17].

Why is it so difficult to have a satisfactory logical treatment of control? There is certainly a question of point of view which stands there. For instance, we can notice that there is a mismatch between sequent calculus proof theory and logic programming: while in sequent calculus, the objects we manipulate are proofs (and the theorems which are proved 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. 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.

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 proof rules which are applied to sequents: the sequent should rather be considered as a constraint on the action that can be performed to progress towards completing the search.

Ludics and Interaction. We noticed earlier the strong relationship 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. Such impacts can be expected in particular when these new results shed light on concepts that had not been well understood before. We think that the recent work by Girard on Ludics [12] should be considered in this way.

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 since they are an abstraction of linear sequent calculus proofs and a concrete version of strategies in a game semantics model. Ludics is founded on many concepts of proof theory, especially of linear logic sequent calculus [11] and especially the fundamental result of Focalization [2] that allows for synthetic connectives to be defined. Ludics has lots of connections with game semantics [14, 1] as well since the interaction process in Ludics can be seen as a play [9]. Many concepts have been inspired by proof search. We shall introduce the main definitions of Ludics in section 3 after we provided some more intuitions on Ludics concepts and technics.

Computation as Interaction. In recent years, interaction has become a crucial concept of the theory of computation. The introduction of linear logic [11] 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, 6, 4]. Ludics certainly goes further in this direction.

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 [7].

This paper is the first of a series of works [19, 20, 21] in which we investigate the use of Ludics as a foundation for proof search and logic programming by means of a model of interactive proof search. The present paper is concerned with the very milestone of this project, that is defining what would be an interactive proof search procedure in Ludics. In order to do so, we first draw the general picture of this paradigm of "computation as interaction" (or more precisely "computation as interactive proof search"), give an informal account of Ludics together with intuitions and motivations and illustrate it with examples from MALL sequent calculus in Section 2. To set the stage to the definitions of Interactive Proof Search in section 4, we then introduce the reader to the basic definition of Ludics in Section 3.

The heart of the paper is Section 4 where we present an algorithm for IPS through the definition of an abstract machine, the SLAM, that is obtained by modifying Faggian's Loci Abstract Machine [8]. After running the interactive search process on an example, we define the SLAM firstly in a restricted case and later in the general setting and we establish several properties of the IPS related with correctness of this algorithm and then we illustrate our method on a toy program for paths in a graph. We finally conclude in Section 5 by outlining future works and relating our results with other works.

2. Logic programming, interactivity and Ludics

The aim of this section is to draw the general picture for Interactive Proof Search and to give an informal description of Ludics.

2.1 Computation as Interaction

We described Computation as Proof Search in the previous section as the search for a proof of $\mathcal{P} \vdash G$ in sequent calculus. The proof is required to satisfy certain conditions.

While the whole dynamics of proof search is concerned with partial proofs, sequent calculus proof theory is a theory of complete proofs. Thus it is very difficult to speak about failures, backtrack or changes in the search strategy (such as what is done by the `cut` predicate) in this setting.

We propose to consider another approach which considers proof search interactively.

Searching for proofs interactively The sequent $\mathcal{P} \vdash G$ is the 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. In some sense, the computation environment cannot really be dealt with explicitly.

This is sad because some important programming primitives precisely deal with these constraints, adding some of them, strengthening others, etc...

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 roughly correspond to the following (the terms will be made clear later on):

- We are willing to search for a proof $\mathcal{D}$ of $\vdash A$.
- Formula A is actually given as a set of tests: the tests that shall be successfully passed by the proof we are constructing: $\mathcal{E}_1, \ldots, \mathcal{E}_n$.
- The proof construction shall proceed by consensus with the tests: $\mathcal{D}$ can be extended with rule R only if the extended object interacts well with the tests.
- After some interaction, we may have an object that cannot be extended any more. Either the construction is terminated

because $\mathcal{D}$ cannot pass every test or because all the test are satisfied and no more constraint applies to $\mathcal{D}$ 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 win.

- The interesting point with this interactive approach lies in the fact that the setting if symmetrical: $\mathcal{D}$ is tested by $\mathcal{E}_i$ but it is also a test for the $\mathcal{E}_i$s. In particular, even failures are interesting and useful objects.
- if a failure $\mathcal{D}^{✠}$ has been reached, we may be willing to try another search. Indeed, maybe at some point we chose a wrong way to extend $\mathcal{D}$ and that caused the failure. There is a standard way to backtrack, that is erase some part of $\mathcal{D}^{✠}$ and try some other construction. But since we are in an interactive setting, there is another option: we can try to use $\mathcal{D}^{✠}$ in order to provide new tests: $\mathcal{E}_i^{\mathcal{D}^{✠}}$ that will constrain the search to look for a proof that shall be different from the failure $\mathcal{D}^{✠}$.
- the use of previous computations in order to enrich the computational behaviour has actually no reason to be restricted to failures...

Basically, this is the research program we are willing to investigate. The present paper will only be concerned with defining the interactive search. The treatment of past computations will be postpone to a future paper.

2.2 Motivations and intuitions for Ludics

Ludics has recently been introduced by Girard [12] as an interactive theory that aims at overcoming the traditional distinction between syntax and semantics by saying that neither syntax nor semantics should come first as a foundational stone for logic: interaction should come first and logic shall be reconstructed from this interactive approach.

The whole theory of Ludics is built on the notion of interaction of designs which are intermediate objects between syntax and semantics (they can be viewed as an abstraction of MALL sequent proofs or as a concrete presentation of game semantics strategies).

Thus in Ludics, things are not built from syntactic objects to which is assigned a semantic interpretation, they do not come from a semantic space for which we need an adequate language. There are objects interacting with each other and their properties are defined interactively. Even Ludics' behaviours which are the analogous of types or formulas will be defined interactively.

Of course, Ludics in not built by forgetting every thing that exists. On the contrary one can argue that Ludics comes from a careful analysis of logic and from a clever synthesis in order to obtain the right notion of interaction. In particular, Ludics has been inspired by many fundamental properties from proof theory that we are going to present in order to provide the reader with intuition before exposing the formal definitions of Ludics.

We now introduce informally and discuss some key notions of Ludics in a way that we hope will emphasize connections towards logic programming and proof search.

Focalization. Andréoli showed a fundamental property of linear logic proofs which has great impact on proof search. Focalization is also very important because it is the root of a polarized approach to logic and polarization is the first step towards a game theoretic interpretation of proofs. Indeed polarization tells you whose turn it is to play.

Probably the most important outcome of Focalization is the possibility of defining synthetic connectives and synthetic rules in MALL and a hypersequentialized calculus (that is, a calculus using the synthetic rules). MALL connectives can be decomposed in two sets: the positive connectives ($\otimes, \oplus, \mathbf{1}, \mathbf{0}$) and the negative connectives ($\parr, \&, \bot, \top$). When searching for a proof, one alternates between two phases, a synchronous phase and an asynchronous phase. During the asynchronous phase, we are certain not to lose provability, while during the synchronous phase we can make the wrong choice and end up not finding a proof even if the sequent we started with was provable. Thus there is clearly an active phase (positive, synchronous) and a passive phase (negative, asynchronous) and the two phases alternate. This is the first step towards a game theoretic interpretation of sequent proofs:

- the negative phase is the opponent's turns to play (and the asynchronous rule gather all information that is needed to react to this move)
- the positive phase is the player's turn: after a move of the opponent, the player decides what she will play following what her strategy (that is her proof rules) 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.

Proof Normalization. The cut elimination process reflects this game interpretation: a conversion step corresponds to the selection, by the positive rule, of a continuation for the normalization (for the play): think of the selection of a $\&$-premise by a $\oplus$-rule.

But there is still a problem for an interactive interpretation: there are not enough proofs! If the system is consistent, we cannot ever find both a proof for A and a proof for $A^\perp$. 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 $A^\perp$. The only thing 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 for the point where the failure was encountered (here normalization is undefined... for the moment).

A failed (or interrupted) attempt to prove A is a proof tree where some branches are still open. Let us add a new rule to mark the fact that the search for a proof has been stopped, that we gave up: $\dfrac{}{\vdash \Gamma}\ ✠$. What is this sequent $\vdash \Gamma$ where we stopped? It would be unfair to stop if $\vdash \Gamma$ contains a negative formula since decomposing this formula costs nothing (remember: it is asynchronous). Thus we restrict the application of ✠ to sequents that are made only of positive formulas (positive sequents). We thus have paraproofs for any sequents, even for the empty one $\dfrac{}{\vdash}\ ✠$.

Winning and loosing. The normalization between two paraproofs is clearly a process through which they test each other. The one that is caught using ✠ is considered as the loser of the play and since he lost, the play ends there. Notice that this normalization process is an exploration of the two paraproofs: the cut visits some parts of the paraproofs. In the case the normalization ends with ✠ the paraproofs are said to be orthogonal.

Locations. Whereas in functional programming it is important to know if the types of a function and its argument are identical, it is not relevant for proof search to know the complete structure of the proof from the beginning. We only need to know enough to choose a rule to apply. This idea is reflected in Ludics by the use of addresses or locations (or loci). A formula is only manipulated through its address ξ. When we apply a rule on ξ we come to know *where* its subformulas are (not *what* they are...): $\xi i, \xi j, \ldots$ the subaddresses of ξ.

Let us say a word about proof normalization: what happens if we cut $A \,\&\, B$ with $A^\perp \oplus C^\perp$? Clearly, things may go wrong if the $\oplus\, R$ rule is applied to $A^\perp \oplus C^\perp$. But on the other hand if $\oplus\, L$ is applied, the normalization goes perfectly well... the problem we

noticed in case of the $\oplus$ R should be compared to the case of a player that did not anticipated the fact that its opponent could play some move and thus has nothing ready to play in response (except if she can give up with a ✠!).

Behaviours. A provable formula can be considered as the set of its proofs... since any formula is paraprovable in Ludics, it is perfect! Actually things are even more drastic in Ludics: the formula is defined interactively depending on the way the tests go. They will be defined by a standard technique of biorthogonality closure.

Incarnation. Given a paraproof Π in A and a paraproof Π' in $A^\perp$, a part of Π can be explored by Π' thanks to normalization. Under very special circumstances, it may happen that Π is entirely visited by Π'. The usual situation is actually that there are parts of Π that cannot be explored, whatever $\Pi' \in A^\perp$ you choose. However, a class of para-proofs which is highly interesting from our interactive perspective is the class of paraproofs that can be completely visited during normalization against elements of $A^\perp$. They are said to be incarnated or material. They are the most interesting elements in A since they can be completely characterized interactively.

2.3 Searching for proof interactively? Interactive proof search in MALL.

In this section, we give some concrete logical example where the construction of a proof of a MALL sequent is led by a context made of "counter-proofs" (they are proofs with weakening and daimon – ✠).

2.3.1 Adding more proofs

If we want to search for proofs by interaction, we need to have object to interact with, but we never have a proof for A and $A^\perp$ at the same time, that means we need to extend a little bit the logic... and introduce the daimon for instance as described in the previous section.

2.3.2 An example of IPS in MALL✠

Consider the two following paraproofs of sequent $\mathbf{1}_0^\perp \oplus (\mathbf{1}_1^\perp \oplus \mathbf{1}_2^\perp)^\perp \vdash$:

$$\mathcal{D}_i = \dfrac{\dfrac{}{\vdash \mathbf{1}_0}\ \text{✠} \quad \dfrac{\dfrac{\dfrac{}{\vdash}\ \text{✠}}{\mathbf{1}_i \vdash}\ \mathbf{1}\vdash}{\vdash \mathbf{1}_1^\perp \oplus \mathbf{1}_2^\perp}\ \vdash\oplus_i}{\mathbf{1}_0^\perp \oplus (\mathbf{1}_1^\perp \oplus \mathbf{1}_2^\perp)^\perp \vdash}\ \oplus\vdash$$

i is 1 or 2.

We can look for a paraproof $\mathcal{D}$ of sequent $\vdash \mathbf{1}_0^\perp \oplus (\mathbf{1}_1^\perp \oplus \mathbf{1}_2^\perp)^\perp$ such that the paraproof Π_i ($i \in \{1,2\}$) built by relating $\mathcal{D}$ to any of the $\mathcal{D}_i$s using a cut can normalize, that is reduce the cut up to finding simply the empty sequent proved thanks to the ✠ rule which is the only rule that can prove any positive sequent.

$$\dfrac{\dfrac{\mathcal{D}_i}{\mathbf{1}_0^\perp \oplus (\mathbf{1}_1^\perp \oplus \mathbf{1}_2^\perp)^\perp \vdash} \quad \dfrac{\mathcal{D}}{\vdash \mathbf{1}_0^\perp \oplus (\mathbf{1}_1^\perp \oplus \mathbf{1}_2^\perp)^\perp}}{\vdash}\ cut$$

Performing the cut reduction will impose constraints on $\mathcal{D}$ and we can use this as a guide to search for a paraproof on $\vdash \mathbf{1}_0^\perp \oplus (\mathbf{1}_1^\perp \oplus \mathbf{1}_2^\perp)^\perp$

Finally, we end up with:

$$\mathcal{D} = \dfrac{\dfrac{\dfrac{\star_1}{\vdash \mathbf{1}_1} \vdash \mathbf{1} \quad \dfrac{\star_2}{\vdash \mathbf{1}_2} \vdash \mathbf{1}}{\mathbf{1}_1^\perp \oplus \mathbf{1}_2^\perp \vdash}\ \oplus\vdash}{\vdash \mathbf{1}_0^\perp \oplus (\mathbf{1}_1^\perp \oplus \mathbf{1}_2^\perp)^\perp}\ \vdash\oplus_2$$

The branch ending at $\star_1$ has been built by interacting with $\mathcal{D}_1$ while the branch ending at $\star_2$ has been built by interacting with $\mathcal{D}_2$

When normalizing $\mathcal{D}$ with $\mathcal{D}_i$, the ✠ used by $\mathcal{D}_i$ that means that the paraproof $\mathcal{D}$ was able to find an error in $\mathcal{D}_i$, without letting $\mathcal{D}_i$ discover its potential logical leakage.

One could have added to the $\mathcal{D}_i$s a third paraproof:

$$\mathcal{D}_3 = \dfrac{\dfrac{}{\vdash \mathbf{1}_0}\ \text{✠} \quad \dfrac{}{\vdash \mathbf{1}_1^\perp \oplus \mathbf{1}_2^\perp}\ \text{✠}}{\mathbf{1}_0^\perp \oplus (\mathbf{1}_1^\perp \oplus \mathbf{1}_2^\perp)^\perp \vdash}\ \oplus\vdash$$

And in this case, one could have interactively built $\mathcal{D}'$:

$$\mathcal{D}' = \dfrac{\dfrac{\dfrac{}{\vdash}\ \text{✠}}{\mathbf{1}_0 \vdash}\ \oplus\vdash}{\vdash \mathbf{1}_0^\perp \oplus (\mathbf{1}_1^\perp \oplus \mathbf{1}_2^\perp)^\perp}\ \vdash\oplus_1$$

During the normalization between $\mathcal{D}$ and $\mathcal{D}_3$, $\mathcal{D}_3$ launches the ✠, but during the normalization between $\mathcal{D}$ and any $\mathcal{D}_i$ the ✠ would have been caused by $\mathcal{D}$: $\mathcal{D}$ loses the dispute with $\mathcal{D}_i$, $i \in \{1,2\}$, but wins the dispute with $\mathcal{D}_3$.

Finally, one could even have imagined adding the following (a bit peculiar) paraproof:

$$\mathcal{D}_4 = \dfrac{\dfrac{}{\vdash \mathbf{1}_0}\ \text{✠}}{\mathbf{1}_0^\perp \oplus (\mathbf{1}_1^\perp \oplus \mathbf{1}_2^\perp)^\perp \vdash}\ \oplus\vdash|_1$$

If $\mathcal{D}_4$ is in the normalization environment then $\mathcal{D}$ **is forced** to use $\vdash\oplus_1$ as a first rule.

On the contrary, it would have been possible to add:

$$\mathcal{D}_5 = \dfrac{\dfrac{}{\vdash \mathbf{1}_1^\perp \oplus \mathbf{1}_2^\perp}\ \text{✠}}{\mathbf{1}_0^\perp \oplus (\mathbf{1}_1^\perp \oplus \mathbf{1}_2^\perp)^\perp \vdash}\ \oplus\vdash|_2$$

and the result would have been to forbid the search that leads to a failure by forbidding the first rule $\vdash\oplus_1$.

2.3.3 Beyond MALL✠

This brief study shows that there are lots of possibilities to guide (or constraint) a proof search interactively. However, it is needed to relax some of the logical principles that usually ensure the safety of logic and are precisely the reasons why it is desirable to use logic. For instance, it is needed to add the daimon ✠ which allows to prove any sequent, but it is also important to admit "partial" logical rules (see $\mathcal{D}_4$ and $\mathcal{D}_4$) and other principles of (linear) logic shall be reconsidered (the weakening for instance).

This is one of the reasons why we go to Ludics which is a clean theory with a good theory of interaction. Moreover, the level of abstraction of Ludics is also interesting to develop this kind of model, even though we may go back to something more concrete in a later phase of this program.

3. Introduction to Ludics.

In this section, we provide the definitions for most Ludics objects that we need for interactive proof search. Most of the definitions in this section are adapted from Girard [12] or from Faggian [8].

3.1 Actions and Designs

DEFINITION 1 (Bias, Locus and Ramification). *A* **bias** *is a natural number (written $i, j, k, \ldots$). A* **locus** *(or address) is a finite sequence of biases (written $\xi, \sigma, \tau, \ldots$). σ is a* **sublocus** *of τ if τ is a prefix of σ (written $\tau \sqsubseteq \sigma$). σ and τ are* **disjoint** *if none of them is sublocus of the other. The empty sequence is a locus and is written $\langle\rangle$ or ϵ. Concatenation is written $\xi \star \sigma$ or $\xi\sigma$ when it is not ambiguous and $\xi \star \langle i \rangle$ is written ξi and is said to be the* **immediate sublocus** *of ξ. A* **ramification** *is a finite set of biases (written*

$I, J, K, \ldots$). *Given a ramification I and a locus ξ, we write ξI for the set of loci ξi for all $i \in I$.*

DEFINITION 2 (Base and Prebase). *A* **prebase** *is a set of polarized loci (ξ^+ or ξ^-). A* **base** *is a finite prebase of pairwise disjoint loci with at most one negative locus. A base is said to be* **negative** *if it contains exactly one negative locus, it is* **positive** *otherwise. We write $\xi \vdash \Lambda$ for negative bases, $\vdash \Lambda$ for positive bases and sometimes $\xi_1^{\epsilon_1}, \ldots, \xi_n^{\epsilon_n}$ for arbitrary bases. When a base is a singleton it is said to be* **atomic** *and is written $\xi \vdash$ or $\vdash \xi$. We simply write $\vdash$ for the* **empty base**.

DEFINITION 3 (Proper Action). *A* **proper action** *is a pair of a locus and a ramification together with a polarity: positive proper actions are written $(+, \xi, I)$ or $(\xi, I)^+$ while negative proper actions are written $(-, \xi, I)$ or $(\xi, I)^-$. We say that (ϵ, ξ, I) has* **focus** *ξ and that it creates subloci ξi for $i \in I$.*

We shall sometimes consider pairs of a locus and a ramification without a polarity, say (ξ, I), that we will refer to as **neutral actions**. *Given κ a proper action, we write κ_ν for the neutral action associated to κ. and κ^+ and κ^- for the positive and negative actions with same focus and ramification as κ.*

DEFINITION 4 (Action). *An* **action** *is either a proper action or the* **daimon** *written ✠. ✠ has positive polarity; it has no focus and creates no sublocus.*

We say that action κ **justifies** *action κ' if:*

- *they have opposite polarity;*
- *the focus of κ' is one of the loci created by κ.*

Notice that only proper actions can justify an action and that the daimon cannot be justified (it has no focus).

Designs (the ludics counter-part of proofs or strategies) will be defined as trees of actions satisfying certain properties. We introduce these conditions by defining chronicles first.

DEFINITION 5 (Chronicle). *A* **chronicle** *χ on a base $\beta = \xi_1^{\epsilon_1}, \ldots, \xi_k^{\epsilon_k}$ is a sequence of actions $(\kappa_0, \ldots, \kappa_n)$ such that:*

- **Polarity.** *The polarities of the actions alternate and the first action has the same polarity as the base;*
- **Daimon.** *For $i < n$, κ_i is a proper action;*
- **Justification.** *For $0 \leq i \leq n$ then either (i) κ_i is ✠ or (ii) it is (ϵ, ξ, I) and $\xi^\epsilon \in \beta$ or (iii) it is justified by an action κ_j occurring earlier in χ ($j \leq i$). Moreover, if κ_{i+1} is negative, it shall be justified by κ_i;*
- **Linearity.** *Each focus only appears once in χ.*

The **polarity** *of a chronicle is the polarity of its last action, κ_n.*

All actions except the daimon and the actions using a focus of the base shall be justified. The daimon can only appear as the last action in χ. The first action of the chronicle is either ✠ (in that case $n = 0$) or its focus is an element of the base. The loci cannot be reused in the chronicle.

When writing chronicles (and later designs and slices) we adopt the Faggian's drawing convention: the positive actions are circled with the ramification written outside the circle while the negative actions are not circled. We give in figure 1 three examples of chronicles, χ_1, χ_2, χ_3, which are respectively on bases $\vdash \xi$, $\xi \vdash \sigma$ and $\xi \vdash$.

DEFINITION 6 (Design). *A* **design** *on a base $\beta = \xi_1^{\epsilon_1}, \ldots, \xi_k^{\epsilon_k}$ is a (possibly infinite) forest of actions such that:*

- **Chronicles.** *The branches of the design are chronicles on β;*
- **Positivity.** *The leaves are positive actions;*

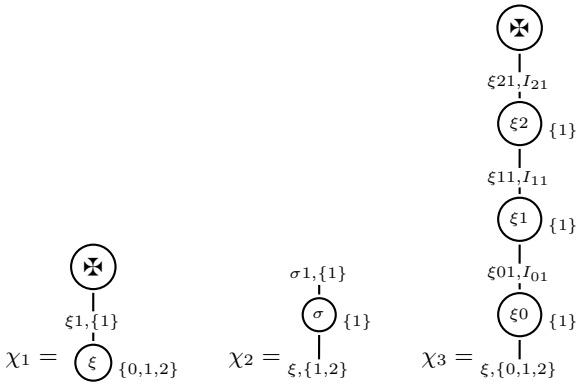


Figure 1. Examples of chronicles

- **Positive branching.** *The tree only branches on positive actions: two incomparable chronicles first differ on negative actions;*
- **Additive sharing.** *If κ_0 and κ_1 are two different actions with the same focus then the chronicles leading to κ_0 and κ_1 first differ on negative actions κ_0' and κ_1' that have the same focus: $\kappa_0 = (-, \xi, I)$ and $\kappa_1 = (-, \xi, J)$;*
- **Totality.** *If the base is positive, then the design is non-empty.*

A design is said to be positive or negative according to his base.

In figure 1, χ_1 and χ_3 are sequences of actions that satisfy the conditions for designs: they are chronicles which are also designs (actually, any non-empty positive chronicle is a design). On the contrary, χ_2 is not a design.

An essential design is the daimon, which is the positive design reduced to an action ✠. There is a daimon for any positive base. Other designs are shown in figure 2 and 3.

It can sometimes be useful to see a design as a set of chronicles with adequate conditions (prefixed-closed, maximal chronicles are positive, etc...).

We now introduce slices which are a particularly important class of designs:

DEFINITION 7 (Slice). *A* **slice** *is a design where the focus of any action is distinct of all other focus, that is in a slice every locus appears at most once.*

A slice of a design $\mathfrak{D}$ is any slice included in $\mathfrak{D}$ when considered as sets of chronicles.

In particular, a negative slice is a tree.

A design can be seen as the superimposition of slices in the same way as done with additive proof nets. We give in figure 2 two examples of slices on $\vdash \xi$ and an example of a design on $\vdash \xi$. Notice that $\mathfrak{D}$ is the superimposition of $\mathfrak{S}_1$ and $\mathfrak{S}_2$.

3.2 Normalization and Interaction

Interaction of designs is built with cut-nets normalization which reflects linear logic cut-normalization. In Ludics, there is no cut-rule in designs: a cut is the coincidence of a locus with opposite polarity in the base of two designs.

DEFINITION 8 (Cut-Net). *A* **cut-net** *$\mathfrak{R} = (\mathfrak{D}_i)_{i \in I}$ is a non-empty finite set of designs on bases $(\beta_i)_{i \in I}$ such that:*

1. *the loci in the bases are either equal or disjoint;*
2. *a locus ξ appears in at most two bases (in that case, it occurs once with positive polarity, once with negative polarity and it is called a* **cut** *in $\mathfrak{R}$);*

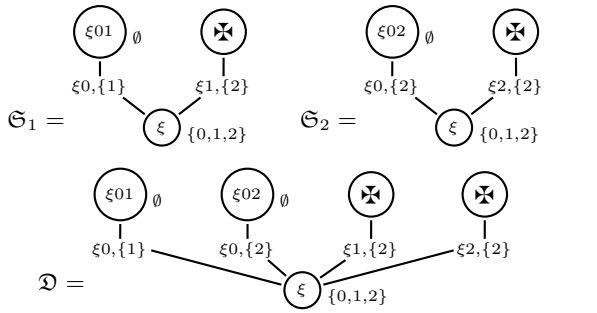


Figure 2. Two slices and a design

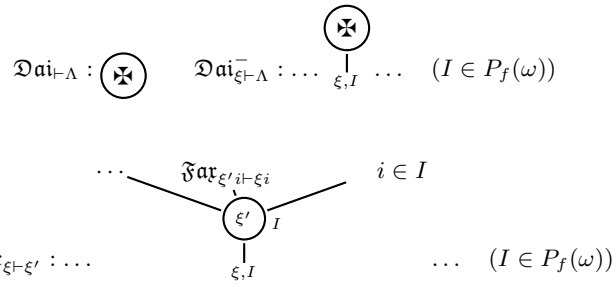


Figure 3. Some important designs

3. *the cuts define a binary relation over the designs which shall be connected and acyclic.*

The **base** *of $\mathfrak{R}$ is the set of polarized loci of the $(\beta_i)_{i\in I}$ which are not cuts. A net is said to be* **closed** *when its base is empty. An action in $\mathcal{R}$ is* **visible** *if it is $\maltese$ or if its focus is sublocus of a locus of the base of the net, it is hidden otherwise. In any cut-net, there is a* **main design** *which is the only positive design of the net if such a design exists or the only negative design of base $\xi \vdash \Lambda$ such that ξ is not a cut.*

Whereas the linearity condition ensures that in a slice all the actions are distinct, a design may contain several times the same action. Thus, describing an action by providing only its focus and ramification is ambiguous: we need an additional information on the *position* of the action in the design. The chronicle leading to the action we consider is surely a good description since it shows the path in the designs of a cut-net leading to this action. The following definition for views will provide a way to find the chronicle for an action provided we know a certain path in the designs of a cut-net leading to this action.

DEFINITION 9 (Positive and Negative Views). *Let χ be a sequence of neutral actions. We define the positive and negative views for χ by induction:*

- $\ulcorner \epsilon \urcorner^+ = \epsilon$;
- $\ulcorner \epsilon \urcorner^- = \epsilon$;
- $\ulcorner s \cdot (\xi, I) \urcorner^+ = \ulcorner s \urcorner^- \cdot (+, \xi, I)$;
- $\ulcorner s \cdot (\xi, I) \urcorner^- = \ulcorner t \urcorner^+ \cdot (-, \xi, I)$ *if $s = tu$ and u is the longest suffix of s such that no (neutral) action in u creates ξ (see definition 3).*

Notice that in the last case of the definition, either t is empty or its last neutral action is (σ, J) with $\xi = \sigma j$ for some $j \in J$.

Moreover, one can trivially extend positive views to sequences ending with the $\maltese$ (even though $\maltese$ is not a neutral action).

Given any χ, both $\ulcorner \chi \urcorner^+$ and $\ulcorner \chi \urcorner^-$ are sequences of actions with alternating polarity and such that the negative actions are either the first action of the sequence or are justified by the previous action in the sequence. That is not enough to ensure $\ulcorner \chi \urcorner^+$ and $\ulcorner \chi \urcorner^-$ are chronicles.

However, when the path will be the path of an interaction, then the view will be a chronicle and will actually be a chronicle in the interacting designs. Moreover, the next definition and proposition will tell more: from any visit path one can reconstruct the chronicle of an action.

DEFINITION 10 (Visit path). *Let $\mathfrak{S}$ be a slice. A path p in $\mathfrak{S}$ (that is a sequence of actions in $\mathfrak{S}$) is a visit path if it is:*

- *of alternating polarities;*
- *made only of proper actions;*
- *downward closed: given a prefix p' of p with last action κ, all the actions below κ in $\mathfrak{S}$ are in p'.*

The polarity of p is the polarity of its last action.

Given a path p, we write p_ν for the sequence of neutral actions canonically associated with p.

Notice that a visit path cannot necessarily be realized by interaction.

PROPOSITION 1. *If $\mathfrak{S}$ is a* slice *on an atomic base β and p is a visit path in $\mathfrak{S}$ of polarity ϵ then $\ulcorner p_\nu \urcorner^\epsilon$ is the chronicle of $\mathfrak{S}$ leading to the last action of p. If this last action is κ we note the chronicle $Ch(\kappa)$*

Proof Let us first notice a property of the visit paths: when a visit path p in a slice $\mathfrak{S}$ ends with a negative action then there is at most one positive action in $\mathfrak{S}$ that can extend p in a path which is still a visit path (if $p\kappa\kappa'$ is prefix of a visit path on $\mathfrak{S}$ with κ a negative action then κ' is immediately above κ in $\mathfrak{S}$).

We prove the proposition by induction on the length n of p and by case on the polarity of p in $\mathfrak{S}$.

- if p is the empty path, then the result is trivial.
- if p is of length 1, then the result is also immediate.
- if $p = p'\kappa$ with p' non-empty. All strict prefixes p'' of p of polarity ϵ are paths satisfying the conditions of the proposition and thus $\ulcorner p''_\nu \urcorner^\epsilon$ is a chronicle in $\mathfrak{S}$.
  - negative path: we have $\ulcorner p_\nu \urcorner^- = \ulcorner t \urcorner^+ \cdot \kappa^-$ with $p = tu$ and u is the longest suffix of s such that no (neutral) action in u creates the focus of κ. Since the base is atomic and $\mathfrak{S}$ is a slice, then $\ulcorner t \urcorner^+$ is not the empty sequence, induction hypothesis ensures it is a chronicle. Let κ_0 be the last action of $\ulcorner t \urcorner^+$. by definition of $\ulcorner t \urcorner^+$, κ_0 justifies κ. Since κ is a negative action in $\mathfrak{S}$ it is placed right above its justification, that is the chronicle for κ is $Ch(\kappa_0) \cdot \kappa = \ulcorner t \urcorner^+ \cdot \kappa^- = \ulcorner p_\nu \urcorner^-$
  - positive path: we have $\ulcorner p \urcorner^+ = \ulcorner p' \urcorner^- \cdot \kappa^+ = s \cdot \kappa^+$ with s a negative chronicle. By induction hypothesis, we know that $\ulcorner p' \urcorner^-$ is a chronicle in $\mathfrak{S}$. Moreover, since κ is positive, the previous action in p (that is, the last action in p') is the negative action which is immediately below κ in $\mathfrak{S}$. Let κ_1 be this action. By induction hypothesis, $\ulcorner p' \urcorner^- = Ch(\kappa_1)$ and finally $Ch(\kappa) = Ch(\kappa_1) \cdot \kappa = \ulcorner p' \urcorner^- \cdot \kappa = \ulcorner p \urcorner^+$

□

In the following definition, we introduce the LAM [8], an abstract machine that computes the interaction of a cut-net $\mathfrak{R}$. The interaction is described as tokens travelling on the cut-net.

DEFINITION 11 (Loci Abstract Machine and Normalization). *Let $\mathfrak{R}$ be a cut-net on a base β. A* **token** *is a pair (s, κ) of a neutral sequence of actions s and an action κ. It represents the position of the token in $\mathfrak{R}$: s records the path that the token has followed from the initial state up to κ.*

Let $T_{\mathfrak{R}}$ be the set of of all positions reached by the tokens during normalization.

- **Initialization.** *If κ is at the root of the main design in $\mathfrak{R}$ the $(\epsilon, \kappa) \in T_{\mathfrak{R}}$;*
- **Transitions.** *Let $(s, \kappa) \in T_{\mathfrak{R}}$. There are three cases:*
  - **Visible.** *If κ is a visible action of polarity ϵ, then for each κ' such that $\ulcorner s\kappa \urcorner^{\epsilon} \kappa' \in \mathfrak{R}$, $(s\kappa, \kappa') \in T_{\mathfrak{R}}$ (notice $\ulcorner s\kappa \urcorner^{\epsilon}$ is the chronicle leading to κ)*
  - **Up.** *If κ is a hidden negative action, then let κ' be the successor of the extremal action of $\ulcorner s\kappa \urcorner^{-}$, we have $(s\kappa, \kappa') \in T_{\mathfrak{R}}$*
  - **Jump.** *If κ is a hidden positive action, then let κ' be the same action as κ but for polarity. If $\ulcorner s\kappa \urcorner^{-} \in \mathfrak{R}$ then $(s, \kappa') \in T_{\mathfrak{R}}$. If $\ulcorner s\kappa \urcorner^{-} \notin \mathfrak{R}$ then normalization fails.*

DEFINITION 12 (Normal form of a cut-net). *Let $\mathfrak{R}$ be a cut-net and let $T_{\mathfrak{R}}$ be the positions reached by the tokens during normalization. A* **normalization path** *is the sequence of actions which are visited during the normalization of $\mathfrak{R}$: $Path(\mathfrak{R})$ is defined to be the set $\{s \cdot \kappa_\nu / (s, \kappa) \in T_{\mathfrak{R}}$ such that s is maximal$\}$. We also define $hide(p)$ to be the sequence resulting from removing all hidden actions in p and* **Hide($\mathfrak{R}$)** *to be the set $\{hide(p), p \in Path(\mathfrak{R})\}$.*

Moreover, given a sequence of neutral actions s, we define s^{ϵ} to be:

- $\epsilon^+ = \epsilon^- = \epsilon$;
- $(s \cdot \kappa)^+ = s^- \cdot \kappa^+$;
- $(s \cdot \kappa)^- = s^+ \cdot \kappa^-$

The **normal form** *of a cut-net $\mathfrak{R}$ is the design defined to be: $[\![\mathfrak{R}]\!] = \{\chi / \chi$ is a prefix of p^+ with $p \in Hide(\mathfrak{R})\}$.*

DEFINITION 13 (Dispute). *If $\mathfrak{R}$ is a closed cut-net, we call* **dispute** *the normalization path of $\mathfrak{R}$.*

If the net is $\{\mathfrak{D}, \mathfrak{E}\}$, we write $[\mathfrak{D} \rightleftharpoons \mathfrak{E}]$ for the dispute.

Property:

- Given $\{\mathfrak{D}, \mathfrak{E}\}$ a closed cut-net, $[\mathfrak{D} \rightleftharpoons \mathfrak{E}]$ is always contained in a single slice of $\mathfrak{D}$ and $\mathfrak{E}$;
- If $\mathfrak{R}$ is a closed cut-net, its base is $\vdash$. The only design on this base is the daimon: $\mathfrak{D}\mathfrak{ai}_{\vdash}$. If normalization of a closed cut-net terminates, its output can only be $\maltese$, otherwise normalization fails.

3.3 Orthogonality and Behaviours

Orthogonality describes those normalizations that were successful.

DEFINITION 14 (Orthogonality). *Two designs $\mathfrak{D}, \mathfrak{E}$ are* **orthogonal** *if they form a cut-net and $[\![\mathfrak{D}, \mathfrak{E}]\!] = \maltese$.*

If $\mathfrak{D}$ and $\mathfrak{E}$ are orthogonal, we write: $\mathfrak{D} \perp \mathfrak{E}$

In general if $\mathfrak{D}$ is a design of base $\xi_1^{\epsilon_1}, \ldots, \xi_n^{\epsilon_n}$ and $(\mathfrak{E}_{\xi_i})_{1 \leq i \leq n}$ are designs on atomic base $\xi_i^{-\epsilon_i}$ (polarity inverse from the one in β), $(\mathfrak{D}, \mathfrak{E}_{\xi_1}, \ldots, \mathfrak{E}_{\xi_n})$ forms a closed cut-net.

If $[\![\mathfrak{D}, \mathfrak{E}_{\xi_1}, \ldots, \mathfrak{E}_{\xi_n}]\!] = \maltese$ we write $\mathfrak{D} \perp (\mathfrak{E}_{\xi_i})_{1 \leq i \leq n}$

DEFINITION 15 (Orthogonal of a design, of a set of designs). *The* **orthogonal** *of a design $\mathfrak{D}$ with an atomic base is:*

$$\mathfrak{D}^{\perp} = \{\mathfrak{E} / \mathfrak{D} \perp \mathfrak{E}\}$$

If $\mathbf{E}$ is a set of designs on the same atomic base (an **ethic***), its orthogonal is:*

$$\mathbf{E}^{\perp} = \{\mathfrak{D} / \forall \mathfrak{E} \in \mathbf{E}, \mathfrak{D} \perp \mathfrak{E}\}$$

Property: Given two ethics $\mathbf{E}$ and $\mathbf{E}'$, the following properties hold:

- if $\mathbf{E} \subseteq \mathbf{E}'$ then $\mathbf{E}'^{\perp} \subseteq \mathbf{E}^{\perp}$;
- $\mathbf{E} \subseteq \mathbf{E}^{\perp\perp}$;
- $\mathbf{E}^{\perp} = \mathbf{E}^{\perp\perp\perp}$.

DEFINITION 16 ($\prec$). *Given $\mathfrak{D}$ and $\mathfrak{D}'$, we define $\mathfrak{D} \prec \mathfrak{D}'$ if $\mathfrak{D}^{\perp} \subseteq \mathfrak{D}'^{\perp}$.*

The preorder $\prec$ relation is actually a partial order.

DEFINITION 17 (Behaviours). *A* **behaviour** $\mathbf{G}$ *is an ethic which is equal to its bi-orthogonal: $\mathbf{G} = \mathbf{G}^{\perp\perp}$.*

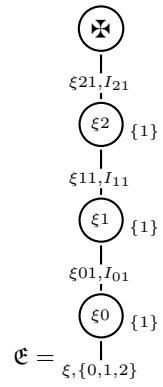
Property: The orthogonal of an ethic is a behaviour.

DEFINITION 18 (Principal behaviour). *Let $\mathfrak{D}$ be a design. The* **principal behaviour** *of $\mathfrak{D}$ is $\{\mathfrak{D}\}^{\perp\perp}$. It is a smallest behaviour containing $\mathfrak{D}$.*

DEFINITION 19 (Incarnation). *If $\mathfrak{E} \in \mathbf{G}$, there exists a smallest design $\mathfrak{D} \subset \mathfrak{E}$ such that $\mathfrak{D} \in \mathbf{G}$ it is the* **incarnation** *of $\mathfrak{E}$ in $\mathbf{G}$ written $|\mathfrak{E}|_{\mathbf{G}}$*

4. Interactive proof search algorithm.

In this section, we present an algorithm for Interactive Proof Search. We give a machine based on Faggian's Loci 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.



4.1 The Idea of the algorithm

Before going to the formal definitions of IPS procedure, we sketch 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 we already considered in figure 1. 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 no path by normalization: $Path_0 = \epsilon$;
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 $\mathfrak{D}$:
$$Path_1 = \langle (\xi, \{0,1,2\}) \rangle = \langle \kappa_{1\nu} \rangle \qquad \mathfrak{D}_1 = \kappa_1^+$$
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:

$$Path_2 = \langle \kappa_{1\nu}, \kappa_{2\nu} \rangle \qquad \mathfrak{D}_2 = \begin{array}{c} \kappa_2^- \\ | \\ \kappa_1^+ \end{array}$$

3. In $\mathfrak{E}$, κ_2^+ is followed by the actions in set $\{(\xi 01, I_{01})^-\}$, we choose an action in this set, say κ_3^-, and we extend $Path_2$ with $\kappa_{3\nu}$ and extend $\mathfrak{D}$ with κ_3^+:

$$Path_3 = \langle \kappa_{1\nu}, \kappa_{2\nu}, \kappa_{3\nu} \rangle \qquad \mathfrak{D}_3 = \begin{array}{c} \kappa_3^+ \\ | \\ \kappa_2^- \\ | \\ \kappa_1^+ \end{array}$$

4. In $\mathfrak{E}$, κ_3^- is followed by the only action $\kappa_4^+ = (\xi 1, \{1\})^+$ and thus, $Path_3$ shall be extend 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 chronicle 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:

$$Path_4 = \langle \kappa_{1\nu}, \kappa_{2\nu}, \kappa_{3\nu}, \kappa_{4\nu} \rangle \qquad \mathfrak{D}_4 = \begin{array}{ccc} \kappa_3^+ & & \\ | & & \\ \kappa_2^- & & \kappa_4^- \\ & \searrow \quad \swarrow & \\ & \kappa_1^+ & \end{array}$$

We then skip several steps to go directly to the last two steps of the IPS:

7. In $\mathfrak{E}$, $Succ(\kappa_6^+) = \{(\xi 21, I_{21})^-\}$. We choose one of these actions, say κ_7^-, to extend the dispute: $Path_7 = \langle \kappa_{1\nu}, \ldots, \kappa_{7\nu} \rangle$ and we add κ_7^+ where the view $\ulcorner Path_7 \urcorner^+$ of the dispute requires it to be placed:

$$Path_7 = \langle \kappa_{1\nu}, \ldots, \kappa_{7\nu} \rangle \qquad \mathfrak{D}_7 = \begin{array}{ccccc} \kappa_3^+ & & \kappa_5^+ & & \kappa_7^+ \\ | & & | & & | \\ \kappa_2^- & & \kappa_4^- & & \kappa_6^- \\ & \searrow & | & \swarrow & \\ & & \kappa_1^+ & & \end{array}$$

8. In $\mathfrak{E}$, κ_7^- is followed by a unique action, $\kappa_8^+ = \maltese$. Since the dispute leads to this action, the normalization ends with $\mathfrak{E}$ using a $\maltese$ and the final dispute is $[\mathfrak{D} \rightleftharpoons \mathfrak{E}] = \langle \kappa_{1\nu}, \ldots, \kappa_{7\nu}, \maltese \rangle$

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

This example was intended to illustrate the basic mechanisms that we shall encounter while doing IPS. We now introduce formal definitions that are needed to give a precise definition of the IPS process.

4.2 Definitions

DEFINITION 20 (IPS Design). *An* **IPS design** *is given by a design $\mathfrak{D}$ together with a set $\mathfrak{IP}$ of visit paths on $\mathfrak{D}$, the* **interaction paths**.

If $p \in \mathfrak{IP}$ leads to a leaf of $\mathfrak{D}$, then this leaf is said to be **open**. *We will sometimes refer to an open action κ as κ^{op}.*

DEFINITION 21 (bipartite Cut-net). *A* **bipartite Cut-net** *is a pair of two sets of designs $\mathbf{E}$ and $\mathbf{F}$ such that:*

- *$\mathbf{E} \cup \mathbf{F}$ is a cut net;*
- *the cut relation induces a* bipartite *graph between the elements of $\mathbf{E}$ and the elements of $\mathbf{F}$. That is, if $\mathfrak{D}_1$ and $\mathfrak{D}_2$ have base that share a cut, then one shall be in $\mathbf{E}$ and the other in $\mathbf{F}$.*

Note that since the cut relation is acyclic, one can always present a cut-net as a bipartite cut-net.

DEFINITION 22 (IPS Cut-net). *An* **IPS Cut-net** *is a closed bipartite cut-net $(\mathbf{E}_{IPS}, \mathbf{E}_{ENV})$ where $\mathbf{E}_{IPS}$ is a set of IPS designs and such that all the visit paths given by the IPS designs can be realized by the normalization in $\mathbf{E}_{IPS} \cup \mathbf{E}_{ENV}$.*

The designs in $\mathbf{E}_{IPS}$ are the designs to be built interactively while the elements of $\mathbf{E}_{ENV}$ are the designs of the environment that will guide the interactive search. The constraint on bipartite graphs ensures that designs in $\mathbf{E}_{ENV}$ only interact with designs in $\mathbf{E}_{ENV}$.

DEFINITION 23 (initial IPS Cut-net). *An* **IPS Cut-net** *is said to be initial when the first component $\mathbf{E}_{IPS}$ is such that a positive design in $\mathbf{E}_{IPS}$ shall be the daimon and a negative design shall be empty.*

4.3 SLAM-1

Let $\mathfrak{R} = (\mathbf{E}_{IPS}, \mathbf{E}_{ENV})$ be an initial IPS Cut net and let $\mathbf{E}_{IPS} = \{\mathfrak{D}_1, \ldots, \mathfrak{D}_n\}$ and $\mathbf{E}_{ENV} = \{\mathfrak{E}_1, \ldots, \mathfrak{E}_m\}$. In the algorithm, the sets $\mathbf{E}_{PIS}$ and $\mathbf{E}_{ENV}$ are viewed as sets of chronicles.

Initialization.

- if the main design of $\mathfrak{R}$ is in $\mathbf{E}_{ENV}$ with let its first action be κ^+ then $P_0 = (\epsilon, \kappa^+)$;
- if the main design $\mathfrak{D}$ of $\mathfrak{R}$ is in $\mathbf{E}_{IPS}$, then consider all designs of $\mathbf{E}_{ENV}$ that share a cut with $\mathfrak{D}$: $(\mathfrak{E}_j)_{j \in J}$. Those designs $\mathfrak{E}_j$ are negative. let $Init = \cup_{j \in J} Init(\mathfrak{E}_j)$ with $Init(\mathfrak{E}_j)$ the set of initial actions for $\mathfrak{E}_j$. Choose $\kappa^- \in Init$ and set: $P_0 = (\epsilon, \kappa^+)$;

Progression.

1. If $(s, \sigma^+) \in P_n$ with $\ulcorner s\sigma \urcorner^+ \in \mathbf{E}_{ENV}$, then add $\ulcorner s\sigma \urcorner^-$ in $\mathbf{E}_{IPS}$ and extend the corresponding interaction path with $s\sigma$: $(s, \sigma^-) \in P_{n+1}$;
2. If $(s, \sigma^-) \in P_n$ with $\ulcorner s\sigma \urcorner^- \in \mathbf{E}_{ENV}$, then let κ^+ be above σ^- in $\mathbf{E}_{ENV}$ (such an action exists and is unique), then $(s\sigma, \kappa^+) \in P_{n+1}$;
3. If $(s, \sigma^+) \in P_n$ with $\ulcorner s\sigma \urcorner^+ \in \mathbf{E}_{IPS}$, then $(s, \sigma^-) \in P_{n+1}$;
4. If $(s, \sigma^-) \in P_n$ with $\ulcorner s\sigma \urcorner^- \in \mathbf{E}_{IPS}$, then let $Succ(\ulcorner s\sigma \urcorner^+) = \{\kappa^- / \ulcorner s\sigma \urcorner^+ \kappa^- \in \mathbf{E}_{ENV}\}$. We choose a $\kappa^- \in Succ(\ulcorner s\sigma \urcorner^+)$ and we add $\ulcorner s\sigma^- \kappa^+ \urcorner^+$ in $\mathbf{E}_{IPS}$ and $(s\sigma, \kappa^+) \in P_{n+1}$.
   If $Succ(\ulcorner s\sigma \urcorner^+) = \emptyset$ then add $\ulcorner s\sigma^- \urcorner^- \maltese$ in $\mathbf{E}_{IPS}$

4.3.1 Properties of the SLAM-1

We state some essential properties of SLAM-1 when executed with an atomic environment on base ξ^ϵ.

The algorithm is non-deterministic but one can easily build a deterministic version of the algorithm. We postpone this to a future paper which will be dedicated to the study of backtrack and the treatment of control in Ludics Programming.

At any steps of the computation, we can consider the chronicles which are elements of the object under construction. If there is a maximal chronicle which is negative, we can extend it with a $\maltese$. By doing so, we get a design $\mathfrak{D}_n^{\maltese}$ at step n.

PROPOSITION 2. *If we execute SLAM-1 with an environment made of only one atomic design $\mathfrak{E}$, then at every step $\mathfrak{D}_n^{\maltese} \perp \mathfrak{E}$.*

PROPOSITION 3. *Any $\mathfrak{D}^{\Phi}$ that is built during the SLAM-1 execution is material in $\{\mathfrak{E}\}^{\perp\perp}$.*

PROPOSITION 4. *If $i \leq j$ then $\mathfrak{D}_j^{\Phi} \prec \mathfrak{D}_i^{\Phi}$.*

We have actually: $\{\mathfrak{E}\}^{\perp\perp} \subseteq \{\mathfrak{D}_j^{\Phi}\}^{\perp} \subseteq \{\mathfrak{D}_i^{\Phi}\}^{\perp}$ for $i \leq j$.
We can also phrase is as: $\{\mathfrak{D}_i^{\Phi}\}^{\perp\perp} \subseteq \{\mathfrak{D}_j^{\Phi}\}^{\perp\perp} \subseteq \{\mathfrak{E}\}^{\perp}$ for $i \leq j$.

Thus the $\mathfrak{D}_i^{\Phi}$ are more and more precise.

4.4 SLAM-n

The IPS procedure described by the SLAM-1 only produces slices. We extend SLAM-1 to n-Environments and n-IPS Cut nets in order to do interactive proof search with more than on set of designs and thus to enrich the search behaviour.

4.4.1 Definitions for SLAM-n

DEFINITION 24 (Base orthogonality). *Given bases $\{\beta_1, \ldots, \beta_n\}$, $\{\beta'_1, \ldots, \beta'_m\}$ is said to be* **orthogonal** *to $\{\beta_1, \ldots, \beta_n\}$ if there is a bipartite cut-net made of designs on $\{\beta_1, \ldots, \beta_n\}$ on the one hand and of designs on $\{\beta'_1, \ldots, \beta'_m\}$ on the other hand.*

DEFINITION 25 (n-Environment). *Given designs $\{\mathfrak{D}_1, \ldots, \mathfrak{D}_n\}$ on bases $\{\beta_1, \ldots, \beta_n\}$, a* **n-Environment** *is a family of sets of designs $(\mathbf{E}nv_i)_{i\in I} = (\{\mathfrak{E}_1^i, \ldots, \mathfrak{E}_{m_i}^i\})_{i\in I}$ such that for all $i \in I$, the bases of $\mathbf{E}nv_i$ are orthogonal to the bases of $\{\mathfrak{D}_1, \ldots, \mathfrak{D}_n\}$.*

DEFINITION 26 (n-IPS Cut net). *A* **n-IPS Cut net** *is given by:*

- *a set of designs $\{\mathfrak{D}_1, \ldots, \mathfrak{D}_n\}$;*
- *a family $(\mathfrak{IP}_i)_{i\in I}$ of paths such that the $\mathfrak{D}_i$ are IPS-designs;*
- *a n-environment $(\mathbf{E}nv_i)_{i\in I}$ for $\{\mathfrak{D}_1, \ldots, \mathfrak{D}_n\}$*

such that for all $i \in I$, $\{\mathfrak{D}_1, \ldots, \mathfrak{D}_n\}$, $\mathfrak{IP}_i$ together with $\mathbf{E}nv_i$ form an IPS Cut net.

An n-IPS Cut net is initial if for all $i \in I$, $\{\mathfrak{D}_1, \ldots, \mathfrak{D}_n\}$, $\mathfrak{IP}_i$ $\mathbf{E}nv_i$ form an initial IPS Cut net.

4.4.2 SLAM-n

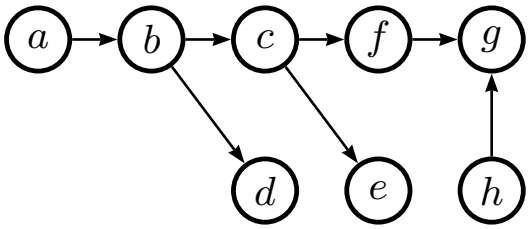
Let $\mathfrak{R}$ an initial n-IPS Cut net and let $(\mathfrak{R}^i)_{i\in I}$ the IPS Cut nets associated (the IPS part is shared).

Initialization. The main designs are either in $\mathbf{E}_{IPS}^i$ for all $i \in I$ or in $\mathbf{E}_{ENV}^i$ for all $i \in I$.

- if the main designs of $\mathfrak{R}$ are ENV, let $(\kappa^{+i})_{i\in I}$ be the first action of the main designs. then $(\epsilon, \kappa^{+i}, i) \in P_0$ for all $i \in I$.
- if the main designs of $\mathfrak{R}$ are IPS, then it is the same design for all the cut-nets. Given $i \in I$, let $Init_i = \cup_{j\in J_i} Init(\mathfrak{E}_j^i)$ for $\mathfrak{E}_j^i$ interacting with $\mathfrak{D}$. Let $Init = \cap_{i\in I} Init_i$.
  - If $Init = \emptyset$, then the IPS is finished and $\mathfrak{D} = \Phi$;
  - If $Init \neq \emptyset$, let $\kappa^- \in Init$ and $\forall i \in I, (\epsilon, \kappa^+, i) \in P_0$

Progression.

1. If $(s, \sigma^+, i) \in P_n$ with $\ulcorner s\sigma \urcorner^+ \in \mathbf{E}_{ENV}^i$, then
   - if $\ulcorner s\sigma \urcorner^- \in \mathbf{E}_{IPS}$ then $(s, \sigma^-, i) \in P_{n+1}$;
   - if $\ulcorner s\sigma \urcorner^- \notin \mathbf{E}_{IPS}$ then add $\ulcorner s\sigma \urcorner^-$ in $\mathbf{E}_{IPS}$ and extend the corresponding interaction path with $s\sigma$: $(s, \sigma^-) \in P_{n+1}$;
2. If $(s, \sigma^-, i) \in P_n$ with $\ulcorner s\sigma \urcorner^- \in \mathbf{E}_{ENV}^i$, then let κ^+ be above σ^- in $\mathbf{E}_{ENV}^i$, then $(s\sigma, \kappa^+, i) \in P_{n+1}$;
3. If $(s, \sigma^+, i) \in P_n$ with $\ulcorner s\sigma \urcorner^+ \in \mathbf{E}_{IPS}$, then
   - if $\ulcorner s\sigma \urcorner^- \in \mathbf{E}_{ENV}^i$ then $(s, \sigma^-, i) \in P_{n+1}$;
   - if $\ulcorner s\sigma \urcorner^- \notin \mathbf{E}_{ENV}^i$ then FAIL!




```
arc(a).
arc(b).   adj(a, b).
arc(c).   adj(b, c).
arc(d).   adj(b, d).
arc(e).   adj(c, e).
arc(f).   adj(c, f).
arc(g).   adj(f, g).
arc(h).   adj(h, g).

p(X,Y) :-  adj(X,Z), adj(Z,Y).
```

Figure 4. Graph

4. If $(s, \sigma^-, i) \in P_n$ with $\ulcorner s\sigma \urcorner^- \in \mathbf{E}_{IPS}$, then
   - if $\exists \kappa^+$ such that $\ulcorner s\sigma\kappa \urcorner^+ \in \mathbf{E}_{IPS}$ then $(s\sigma, \kappa^+, i) \in P_{n+1}$.
   - if $\nexists \kappa^+$ such that $\ulcorner s\sigma\kappa \urcorner^+ \in \mathbf{E}_{IPS}$ then let $Succ(\ulcorner s\sigma \urcorner^+) = \{\kappa^-/\ulcorner s\sigma \urcorner^+ \kappa^- \in \mathbf{E}_{ENV}^i\}$. We choose a $\kappa^- \in Succ(\ulcorner s\sigma \urcorner^+)$ and we add $\ulcorner s\sigma^- \kappa^+ \urcorner^+$ in $\mathbf{E}_{IPS}$ and $(s\sigma, \kappa^+, i) \in P_{n+1}$. If $Succ(\ulcorner s\sigma \urcorner^+) = \emptyset$ then add $\ulcorner s\sigma^- \urcorner^- \Phi$ in $\mathbf{E}_{IPS}$

4.5 A concrete example: paths in a graph

Let $\mathcal{G}$ be the graph represented in figure 4.

We want to implement the search for paths of length 2 in this graph thanks to interactive proof search. This corresponds to the predicates shown in figure 4.

For instance, $p(c, g)$ could be represented as the MALL formula:

$$\begin{aligned}
&(adj(c,a) \,\&\, adj(a,g)) \\
\oplus\ &(adj(c,c) \,\&\, adj(c,g)) \\
\oplus\ &(adj(c,d) \,\&\, adj(d,g)) \\
\oplus\ &(adj(c,e) \,\&\, adj(e,g)) \\
\oplus\ &(adj(c,f) \,\&\, adj(f,g)) \\
\oplus\ &(adj(c,g) \,\&\, adj(g,g)) \\
\oplus\ &(adj(c,h) \,\&\, adj(h,g))
\end{aligned}$$

One wants to search for a design $\mathfrak{D}$ by interacting with a set of counter-design specifying the graph and the path relation p.

Let suppose $a, b, \ldots, g, h$ are integer codes representing nodes of the graph in the obvious way.

The counter-design environnement would be made of two designs $\mathcal{E}_1$ and $\mathcal{E}_2$:

We are in a case of a 2-environment.

We have 8 choices for the first action in constructing design $\mathcal{D}$, but this leads then to the following designs (8 possible computations) depending on the choice of the first action (only one being a win):

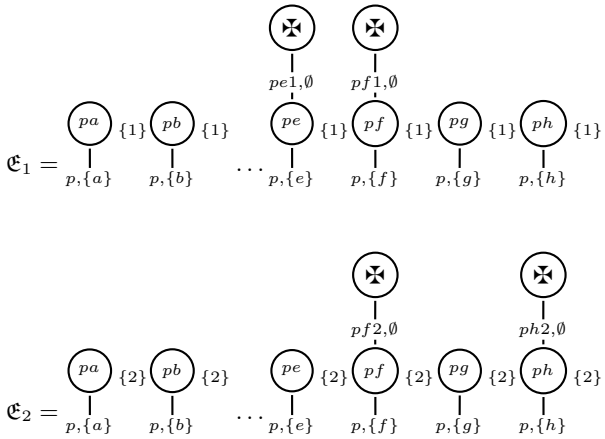
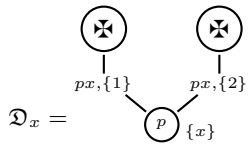
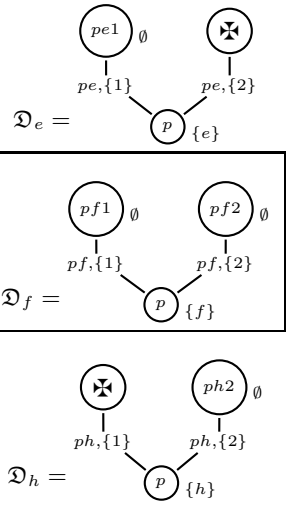


Figure 5. Designs $\mathfrak{E}_1$ and $\mathfrak{E}_2$.



If $x \in \{a, b, c, d, g\}$.



Short explanation of the dialog:

- action $(+, p, \{x\})$ corresponds to asserting that there is a path from c to g via x;
- the bunch of actions $\{(-, p, \{x\})\}_{x\in\{a,b,c,\ldots,g,h\}}$ corresponds to the possible arguments the designs $\mathcal{E}_1$ and $\mathcal{E}_2$ are ready to answer;
- action $(+, px, \{1\})$ (or $(+, px, \{2\})$) corresponds to a challenge by the $\mathcal{E}_i$s to $\mathcal{D}$ meaning: is x adjacent to c (resp. g);
- action $(-, px, \{1\})$ (or $(-, px, \{2\})$) corresponds to the possible attacks by the $\mathcal{E}_i$s the design $\mathcal{D}$ is able to answer;
- action $(+, px1, \emptyset)$ means: "yes! x is adjacent to c" (resp. with index 2 and g). The ✠ appearing instead of the previous positive action means that the error in the argument of $\mathcal{D}$ is there.
- action $(-, px1, \emptyset)$ is the preparation by $\mathcal{E}_1$ to get this argument by $\mathcal{D}$... and the following ✠ means that in this case, design $\mathcal{E}_1$ just gives up (but maybe another design in the environment will continue asking informations/evidences to $\mathcal{D}$ causing the design to be more precise.

5. Conclusion.

The aim of this paper which is the first of a series in which we investigate Ludics as a foundation for Logic programming was to motivate our approach and explain its general picture, to define the Ludics objects that are necessary and to define an Iinteractive Proof Search procedure in Ludics.

The point of IPS, or Ludics Programming, is to consider that the search for a proof in not guided by a sequent as in standard proof search, but that the search is constrained by a context, an environment, which is made of object of the same kind that the one we are building during the computation.

We introduced IPS thanks to an abstract machine, the SLAM, which is adapted from Faggian's Loci Abstract Machine [8]. We illustrated our approach on three kinds of examples: informal IPS for MALL✠ derivations in Section 2, SLAM execution on a simple design in the introduction of Section 4 and finally a more concrete example related with the Prolog program for finding paths in a graph.

We presented the first elemnts for analyzing properties of the SLAM but most has still to be done, in particulier with respect to backtracking and enlarging the environment based on previous computations. This is left for future works.

Related works In [8], Faggian introduced the Loci Abstract Machine and studied some properties which are related with some aspects of the LAM. In particular, from a counter-design and an interaction path, she can reconstruct a design realizing this path.

Pym and Ritter [18] give a semantics for proof search which is related with game semantics. We shall investigate the connections with our work.

Future Works Most is still to be done in order to have a computation model based on interactive proof search:

- first, we have to introduce the backtracking and the logic within our mechanism for IPS [20];
- then, we shall try to extend Ludics in two directions: first-order and recursive definitions. Ludics is a teory without first order

and that may be painful when one wants to program using Ludics, however Fleury and Quatrini [10] studied first-order in Lducis. On the other hand, Ludics is based on MALL and there is no exponentials in Ludics. To enlarge the setting one may be interesting in considering at least recursive definitions or fixpoints. We are currently working on such an extension of Ludics with definitions or fixpoints [21].

Acknowledgments

The author thanks Dale Miller for his advice and directions, Jean-Yves Girard for his comments on this project as well as Claudia Faggian for helpful discussions regarding the material in this paper.

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] Pierre-Louis Curien. Abstract böhm trees. *Mathematical Structures in Computer Science*, 8(6):559–591, 1998.

[5] Pierre-Louis Curien. Abstract machines, control, and sequents. In *Proceedings of APPSEM Summer School*, pages 123–136. Springer, September 2000.

[6] Pierre-Louis Curien. Symmetry and interactivity in programming. *Bulletin of Symbolic Logic*, 9(2):169–180, 2003.

[7] Pierre-Louis Curien and Hugo Herbelin. The duality of computation. In *ICFP*, pages 233–243, 2000.

[8] Claudia Faggian. Travelling on designs. In *Proceedings of CSL'02*, pages 427–441, 2002.

[9] Claudia Faggian and Martin Hyland. Designs, disputes and strategies. In *CSL*, pages 442–457, 2002.

[10] Marie-Renee Fleury and Myriam Quatrini. First order in ludics. *Mathematical Structures in Computer Science*, 14(2):189–213, 2004.

[11] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[12] Jean-Yves Girard. Locus solum. *Mathematical Structures in Computer Science*, 11(3):301–506, June 2001.

[13] Timothy Griffin. A formulae-as-types notion of control. In *POPL'90*, pages 47–58, 1990.

[14] 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.

[15] R. Kowalski. Algorithm = logic + control. *Communications of the Association for Computing Machinery*, 22:424–436, 1979.

[16] 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.

[17] 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.

[18] David J. Pym and Eike Ritter. A games semantics for reductive logic and proof-search. In *GALOP*, pages 107–123, 2005.

[19] Alexis Saurin. Programmation logique, ludique et contrôle: vers une programmation ludique. Unpublished draft, june 2004.

[20] Alexis Saurin. Ludics programming II: backtrack and control. Unpublished draft, july 2007.

[21] Alexis Saurin. Ludics programming III: definition and fixpoints in ludics. Unpublished draft, july 2007.