Reductive Logic, Proof-search, and Coalgebra: A Perspective from Resource Semantics

Alexander Gheorghiu, Simon Docherty, David Pym
University College London
{alexander.gheorghiu.19 , s.docherty , d.pym} @ucl.ac.uk

September 18, 2022

Abstract

The reductive, as opposed to deductive, view of logic is the form of logic that is, perhaps, most widely employed in practical reasoning. In particular, it is the basis of logic programming. Here, building on the idea of uniform proof in reductive logic, we give a treatment of logic programming for BI, the logic of bunched implications, giving both operational and denotational semantics, together with soundness and completeness theorems, all couched in terms of the resource interpretation of BI's semantics. We use this set-up as a basis for exploring how coalgebraic semantics can, in contrast to the basic denotational semantics, be used to describe the concrete operational choices that are an essential part of proof-search. The overall aim, toward which this paper can be seen as an initial step, is to develop a uniform, generic, mathematical framework for understanding the relationship between the deductive structure of logics and the control structures of the corresponding reductive paradigm.

1 Introduction

While the traditional deductive approach to logic begins with premisses and in step-by-step fashion applies proof rules to derive conclusions, the complementary reductive approach instead begins with a putative conclusion and searches for premisses sufficient for a legitimate derivation to exist by systematically reducing the space of possible proofs. A first step in developing a mathematical theory of reductive logic, in the setting of intuitionistic and classical logic, has been given in [78].

Not only does this picture more closely resemble the way in which mathematicians actually prove theorems and, more generally, the way in which people solve problems using formal representations, it also encapsulates diverse applications of logic in computer science such as the programming paradigm known as *logic programming*, the proof-search problem at the heart of AI and automated theorem proving, precondition generation in program verification, and more [51]. It is also reflected at the level of truth-functional semantics — the perspective on logic utilized for the purpose of model checking and thus verifying the correctness of industrial systems — wherein the truth value of a formula is calculated according to the truth values of its constituent parts.

The definition of a system of logic may be given *proof-theoretically* as a collection of rules of inference that, when composed, determine proofs; that is, formal constructions of arguments that establish that a conclusion is a consequence of some assumptions:

$$\frac{\text{Premiss}_1 \quad \ldots \quad \text{Premiss}_k}{\text{Conclusion}} \Downarrow$$

This systematic use of symbolic and mathematical techniques to determine the forms of valid deductive argument defines *deductive logic*: conclusions are inferred from assumptions.

This is all very well as a way of defining what proofs are, but it relatively rarely reflects either the way in which logic is used in practical reasoning problems or the method by which proofs are actually found. Rather, proofs are more often constructed by starting with a desired, or putative, conclusion and applying the rules of inference 'backwards'.

In this usage, the rules are sometimes called *reduction operators*, read from conclusion to premisses, and denoted

$$\frac{\text{Sufficient Premiss}_1 \quad \ldots \quad \text{Sufficient Premiss}_k}{\text{Putative Conclusion}} \Uparrow$$

Constructions in a system of reduction operators are called *reductions* and this view of logic is termed 'reductive' [78]. The space of reductions of a (provable) putative conclusion is larger than its space of proofs, including also failed searches.

The background to these issues, in the context of capturing human reasoning, is discussed extensively in, for example, the work of Kowalski [50] and Bundy [16]. Reductive reasoning lies at the heart of widely deployed proof assistants such as LCF [28], HOL [27], Isabelle [65], Coq [1, 11], Twelf [2, 67], and more; most of which can be seen as building on Milner's theory of tactical proof [63] — that is, 'tactics' and 'tacticals' for proof-search — which gave rise to the programming language ML [66].

In deductive logic, if one has proved a set of premisses for a rule, then one immediately has a proof of the conclusion by an application of the rule; and, usually, the order in which the premisses are established is of little significance. In reductive logic, however, when moving from a putative conclusion to some choice of sufficient premisses, a number of other factors come into play. First, the order in which the premisses are attempted may be significant not only for the logical success of the reduction but also for the efficiency of the search. Second, the search may fail, simply because the putative conclusion is not provable. These factors are handled by a *control process*, which is therefore a first-class citizen in reductive logic.

Logic programming can be seen as a particular form of implementation of reductive logic. While this perspective is, perhaps, somewhat obscured by the usual presentation of Horn-clause logic programming with SLD-resolution (HcLP) — see, for example, [56] among many excellent sources — it is quite clearly seen in Miller's presentation of logic programming for intuitionistic logic (hHIL) in [58], which generalizes to the higher-order case [59].

Miller's presentation uses an operational semantics that is based on a restriction of Gentzen's sequent calculus LJ to hereditary Harrop formulas, given by the following mutually inductive definition:

$$\begin{array}{lcl} D & ::= & A \in \mathbb{A} \mid \forall x\, D \mid G \to A \mid D \wedge D \\ G & ::= & A \in \mathbb{A} \mid \exists x\, G \mid D \to G \mid G \wedge G \mid G \vee G \end{array}$$

where $\mathbb{A}$ is a denumerable set of propositional letters.

The basic idea is that a program is a list of 'definite' clauses D, and a query is given by a 'goal' formula G, so that a *configuration* in HhIL is given by a sequent

$$D_1, \ldots D_k \vdash G$$

We are assuming that all formulas in the sequent are closed, and therefore suppress the use of quantifiers.

The execution of a configuration $P \vdash G$ is defined by the search for a 'uniform' proof,

- Apply right rules of LJ reductively until G is reduced to an atom (here we are assuming some control strategy for selecting branches);
- At a sequent $P \vdash A$, choose (assuming some selection strategy) a definite clause of the form $G \to A$ in the program, and apply the $\to L$ rule of LJ reductively to yield the goal $P \vdash G'$. More precisely, due to the (suppressed) quantifiers, we use *unifiers* to match clauses and goals; that is, in general, we actually compute a substitution θ such that, for a clause $G' \to B$ in the program, we check $A\theta = B\theta$ and proceed to the goal $P \vdash G'\theta$. This step amounts to a 'resolution'.

If all branches terminate successfully, we have computed an overall, composite substitution θ such that $P \vdash G\theta$. This θ is the output of the computation.

The denotational semantics of logic programming amounts to a refinement of the term model construction for the underlying logic that, to some extent at least, follows the operational semantics of the programming language; that is, the control process for proof-search that delivers computation.

In Miller's setting, the denotational semantics is given in terms of the least fixed point of an operator T on the complete lattice of Herbrand interpretations. The operator is generated by the implicational clauses in a program, and its least fixed point characterizes the space of goals that is computable by the program in terms of the resolution step

in the operational semantics. Similar constructions are provided for a corresponding first-order dependent type theory in [71].

A general perspective may be offered by proof-theoretic semantics, where model-theoretic notions of validity are replaced by proof-theoretic ones [80, 81]. In the setting of term models, especially for reductive logic [74], it is reasonable to expect a close correspondence between these notions.

In this paper, we study logic programming for *the logic of Bunched Implications* (BI) [64] — which can be seen as the free combination of intuitionistic logic and multiplicative intuitionistic linear logic — in the spirit of Miller's analysis for intuitionistic logic, building on early ideas of Armelín [9, 10]. We emphasise the fundamental rôle played by the reductive logic perspective; and, we begin the development of a denotational semantics using coalgebra, that incorporates representations of the control processes that guide the construction of proof-search procedures. This is a natural choice if one considers that coalgebra can be seen as the appropriate algebraic treatment of systems with state [44, 48, 74]. The choice of BI serves to highlight some pertinent features of the interaction between reduction operators and control; and, results in an interesting, rational alternative to the existing widely studied substructural logic programming languages based on linear logic [40, 38, 17, 39, 57]. That is, we discuss how the coalgebraic model offers insight into control, generally speaking, and the context-management problem for multiplicative connectives, in particular, and how a general approach to handling the problem can be understood abstractly.

The motivations and application of BI as a logic are explained at some length in Section 2, which also sets up proof-search in the context of our view of reductive logic, in general, and the corresponding notions of hereditary Harrop formulas and uniform proof, in particular. The bunched structure of BI's sequent calculus renders these concepts quite delicate. We introduce BI from both a semantic, truth-functional perspective and a proof-theoretic one, through its sequent calculus. Before that, though, we begin with a motivation for BI as a logic of resources. While this interpretation is formally inessential, it provides a useful way to think the use of the multiplicative connectives and the computational challenges to which they give rise. Furthermore, the resource interpretation of BI's semantics stands in contrast with the proof-theoretic resource-interpretation of Linear Logic, and so helps to illustrate informally how BI and Linear Logic differ. In the sequel, we also give a precise, purely logical distinction between these two logics.

In Section 3, we establish the least fixed point semantics of logic programming with BI, again in the spirit of Miller's presentation.

While the least fixed point semantics is logically natural and appealing in its conceptual simplicity, its major weakness is that despite being constructed from an abstraction of behaviour, it lacks any explicit representation of the control processes that are central to the operational semantics. These include the selection operations alluded to above in our sketch of uniform proofs, and choices such as depth-first versus breadth-first navigation of the space of searches. In Section 4, we introduce the coalgebraic framework as a model of behaviour. The value of the choice of BI to illustrate these ideas — in addition to BI's value as logic — can now been seen: it has enough structure to challenge the approach (cf. work of Komendantskaya, Power, and Schmidt [48]) without becoming conceptually too complex to understand clearly.

In Section 5, we illustrate the ideas introduced in this paper with a simple example based on databases. Finally, in Section 6, we summarize our contribution and consider some directions for further research.

2 Logic Programming with Resource Semantics

We study proof-search in the logic of Bunched Implications (BI) [64] not because it is easy, but because it is not. As a logic BI has a well-behaved metatheory admitting familiar categorical, algebraic, and truth-functional semantics which have the expected dualities [77, 23, 73, 19, 75], as well as a clean proof theory including Hilbert, natural deduction, sequent calculi, tableaux systems, and display calculi [73, 23, 14, 19]. Moreover, BI has found significant application in program verification as the logical basis for Reynolds' Separation Logic, which is a specific theory of (Boolean) BI, and its many derivatives [43].

What makes proof-search in the logic both difficult and interesting is the interaction between its additive and multiplicative fragments.

2.1 The Logic of Bunched Implications

We begin with a brief summary of the semantics and proof theory of BI. One way to understand it is by contrast with Girard's Linear Logic (LL) [26], which is well-known for its computational interpretation developed by Abramsky [3] and others.

In LL, the structural rules of Weakening and Contraction are regulated by modalities, ! (related to $\Box$ in modal logic) and ? (related to $\Diamond$ in modal logic). To see how this works, consider the left and right (single-conclusioned) sequent calculus rules for the ! modality,

$$\frac{\Gamma, \phi \vdash \psi}{\Gamma, !\phi \vdash \psi} \quad \text{and} \quad \frac{!\Gamma \vdash \phi}{!\Gamma \vdash !\phi}$$

and note that Weakening and Contraction arise as

$$\frac{\Gamma \vdash \psi}{\Gamma, !\phi \vdash \psi}\, W \quad \text{and} \quad \frac{\Gamma, !\phi, !\phi \vdash \psi}{\Gamma, !\phi \vdash \psi}\, C,$$

respectively.

From our perspective, there are two key consequences of this set-up. First, proof-theoretically, the relationship between intuitionistic (additive) implication, $\rightarrow$, and linear (multiplicative) implication, $-\!\circ$, is given by Girard's translation; that is,

$$\phi \rightarrow \psi \equiv (!\phi) -\!\circ \psi$$

Second, more semantically, LL has a rudimentary interpretation as a logic of resource via the so-called *number-of-uses* reading, in which, in a sequent $\Gamma \vdash \psi$, the number of occurrences of a formula ϕ in Γ determines the number of times ϕ may be 'used' in ψ. The significance of the modality ! can now be seen: if $!\phi$ is in Γ, then ϕ may be used any number of times in ψ, including zero, and this reading is wholly consistent with the forms of Weakening and Contraction.

The relationship between logic and structure offered in the above reading has been characterized by Abramsky [4] as the *intrinsic* views. By comparison, BI is, perhaps, the prime example of the *descriptive* view of resource; that is, in contrast to the case for LL, in the resource interpretation of BI a proposition is not a resource itself, but a declaration about the state of some resources. Nonetheless, the constructive reading of BI's implications mean that one can read a sequent, in particular formulas in a context, as things available for the constructions of the formula. This tension is explored further in Section 3. Moreover, although our discussion of logic programming with BI will be guided by the resource semantics, it is in no way essential to the logic's metatheory.

The original presentation of BI emphasised it as an interesting system of formal logic. Technically, it is the combination of intuitionistic logic (IL) and multiplicative intuitionistic linear logic (MILL), and is conservative over both.

Definition 2.1 (Formulas, Additive and Multiplicative Connectives)**.** *Let $\mathbb{A}$ be a denumerable set of propositional letters. The formulas of* BI *are defined by the grammar:*

$$\phi, \psi ::= \top \mid \bot \mid \top^* \mid A \in \mathbb{A} \mid \phi \wedge \psi \mid \phi \vee \psi \mid \phi \rightarrow \psi \mid \phi * \psi \mid \phi -\!\!* \psi$$

Symbols in $\{\wedge, \vee, \rightarrow, \top, \bot\}$ are additive connectives, and symbols in $\{*, -\!\!*, \top^*\}$ are multiplicative connectives.

Following the resource discussion above we introduce BI from a semantic (truth-functional) perspective, before moving on to its proof theory.

Informally, a judgement $m \models \phi \wedge \psi$ is a declaration that the resource m satisfies ϕ and satisfies ψ, meanwhile the judgement $m \models \phi * \psi$ says that it can be *split* into two parts n and n' such that $n \models \phi$ and $n' \models \psi$.

Example 2.2. Consider the case where resources are gold coins and composition is summation. An atomic proposition is a brand of chocolate, and a judgement is a valuation of the cost.

Suppose chocolate bar A costs two gold coins, and chocolate bar B costs three, then we may write $3 \models A \wedge B$ to say that three gold coins suffice for both chocolates. Moreover, $7 \models A * B$ since seven gold coins may be split into two pile of three and four coins, and $3 \models A$ and $4 \models B$. Notice the persistence of the judgement; that is, since $2 < 3$ and two gold coins suffice for A, so do three. ■

$$
\begin{array}{lll}
m \models A & \text{iff} & m \in [\![A]\!] \\
m \models \phi \wedge \psi & \text{iff} & m \models \phi \text{ and } m \models \psi \\
m \models \phi * \psi & \text{iff} & \exists n, n' \in \mathbb{M} : \; m \sqsubseteq n \circ n' \text{ and } n \models \phi \text{ and } n' \models \psi \\
m \models \phi \vee \psi & \text{iff} & m \models \phi \text{ or } m \models \psi \\
m \models \phi \to \psi & \text{iff} & \forall n \sqsubseteq m : \; n \models \phi \implies n \models \psi \\
m \models \phi \mathrel{-\!\!*} \psi & \text{iff} & \forall n \in \mathbb{M} : \; n \models \phi \implies n \circ m \models \psi \\
m \models \top & \text{iff} & m \in \mathbb{M} \\
m \models \top^* & \text{iff} & m \sqsubseteq e
\end{array}
$$

Figure 1: Frame Satisfaction

While any set of objects can, in some sense, be considered an abstract notion of resource, some simple assumptions are both natural and useful. For example, from our experience of resources it makes sense to assume that elements can be *combined* and *compared*. This indicates an appropriate structure for the abstract treatment of them: ordered monoids,

$$\mathcal{M} = \langle \mathbb{M}, \sqsubseteq, \circ, e \rangle$$

in which $\sqsubseteq$ is a preorder on $\mathbb{M}$, $\circ$ is a monoid on $\mathbb{M}$, and e is a unit for $\circ$. That is, combination is given by $\circ$ and comparison given $\sqsubseteq$. Examples of such structures are given by the natural numbers, $\langle \mathbb{N}, \leq, +, 0 \rangle$ and, in Separation Logic, by concatenation and containment of blocks of computer memory.

Such monoids form a semantics of BI with satisfaction given as in Figure 1, but with Beth's treatment of disjunction, and with $\bot$ never satisfied — see [54, 23, 22]. However, for our semantics of logic programming with BI, we are able to work in a much simpler setting, without $\bot$, and in which occurrences of disjunction are restricted to the right-hand sides of sequents, meaning that the simpler Kripke clause for disjunction is adequate.

Definition 2.3 (Resource Frame, Resource Model)**.** *A resource frame is a structure $\langle \mathbb{M}, \sqsubseteq, \circ, e \rangle$ where $\sqsubseteq$ is a preorder; $\circ : \mathbb{M} \times \mathbb{M} \to \mathbb{M}$ is commutative, associative, and bifunctorial — that is, $m \sqsubseteq m' \;\&\; n \sqsubseteq n' \implies m \circ n \sqsubseteq m' \circ n'$, and e is a unit for $\circ$.*

A resource frame together with a monotonic valuation $[\![-]\!] : \mathbb{A} \to \mathcal{P}(\mathbb{M})$ becomes a resource model under the satisfaction relation in Figure 1.

Many different types of proof system — Hilbert-type, natural deduction, λ-calculus, tableaux, and display — are available for BI, but here we restrict attention to sequent calculi since they are more amenable to a concept of *computation as proof-search*. Here the presence of two quite different implications ($\to$ and $\mathrel{-\!\!*}$) means that contexts in the sequent calculus require two different context-formers: one that admits structural rules of Weakening and Contraction, and another that does not,

$$\frac{\Gamma; \phi \implies \psi}{\Gamma \implies \phi \to \psi} \qquad\qquad \frac{\Gamma, \phi \implies \psi}{\Gamma \implies \phi \mathrel{-\!\!*} \psi}$$

As a consequence, contexts in BI sequents are not a flat data structure, such as a sequence or a multiset, but instead are tree structures called *bunches*, a term that derives from the relevance logic literature (see, for example, [79]).

Definition 2.4 (Bunch)**.** *Bunches are constructed with the following grammar:*

$$\Gamma ::= \phi \mid \varnothing_+ \mid \varnothing_\times \mid \Gamma; \Gamma \mid \Gamma, \Gamma$$

where ϕ is a formula; $\varnothing_+$ and $\varnothing_\times$ are the additive and multiplicative units respectively, and, the symbols ; and , are the additive and multiplicative context-formers, respectively. The set of all bunches is denoted $\mathbb{B}$.

The two context-formers individually behave as the comma for IL and MILL, respectively, resulting in the following generalization of equivalence under permutation:

Definition 2.5 (Coherent Equivalence)**.** *Two bunches are coherent when $\Gamma \equiv \Gamma'$, where $\equiv$ is the least equivalence relation on bunches satisfying*

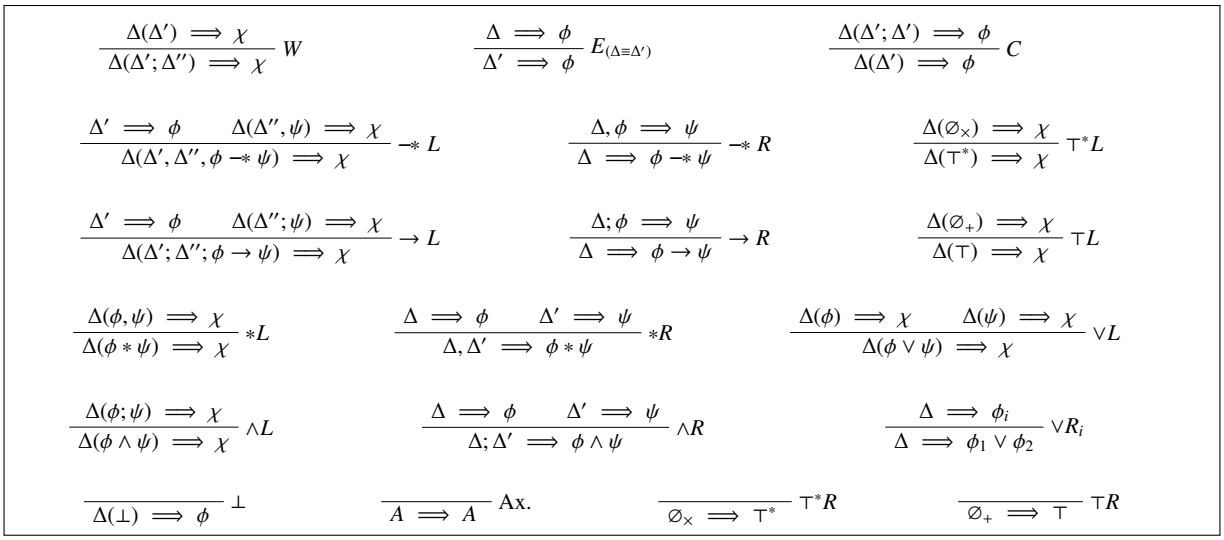


Figure 2: Sequent Calculus **LBI**

- *Commutative monoid equations for* $\varnothing_\times$,
- *Commutative monoid equations for* $\varnothing_+$,
- *Congruence:* $\Delta \equiv \Delta' \implies \Gamma(\Delta) \equiv \Gamma(\Delta')$.

Let $\Gamma(\Delta)$ denote that Δ is a sub-bunch of Γ, then let $\Gamma(\Delta)[\Delta \mapsto \Delta']$ — abbreviated to $\Gamma(\Delta')$ where no confusion arises — be the result of replacing the occurrence of Δ by Δ'.

Definition 2.6 (Sequent)**.** *A sequent is a pair of a bunch* Γ*, called the context, and a formula* ϕ*, and is denoted* $\Gamma \implies \phi$.

Note that a bunch Γ can be read as formula $\lfloor\Gamma\rfloor$ by replacing all additive and multiplicative context-formers with additive and multiplicative conjunctions respectively. Satisfaction extends to bunches by $\models \Gamma \iff \models \lfloor\Gamma\rfloor$.

Figure 2 presents the sequent calculus **LBI** which is sound and complete for BI with respect to the model theory presented above [73]. Note that the system has the subformula property, meaning that analytic proofs are complete, and it is therefore directly for the study of proof-search. Moreover, the system admits Cut [14, 25] (see [13] for the computational value of suitably formulated Cut-based approaches to proof-search),

$$\frac{\Delta \implies \phi \qquad \Gamma(\phi) \implies \psi}{\Gamma(\Delta) \implies \psi} \text{ Cut}$$

The completeness of **LBI** has been established both relative to the Beth semantics discussed above and to a Grothendieck topological semantics [77]. Relaxing composition of resources to be a partial function does yield a Kripke semantics, the completeness of which follows from a duality with the algebraic semantics [20, 19]. In fact, soundness and completeness theorems for BI have been considered in a number settings; see [64, 77, 73, 70, 23, 14, 15, 55, 19] for a range of discussions and explanations.

Finally, while, in LL, Girard's translation gives intuitionistic implication as derived from linear implication and the modality ! , no such translation is available in BI. This can be seen quite readily in the setting of BI's categorical semantics, which is given in terms of *bicartesian doubly closed categories* [64, 73]. That is, it can be shown that there cannot in general exist an endofunctor ! on such a category that satisfies Girard's translation.

2.2 Goal-directed Proof-search

Reduction operators in reductive logic are the set of operations available for computation, but this computation only becomes a procedure, specifically a *proof-search procedure*, in the presence of a *control* flow. It is this step which

begins the definition of an operational semantics of a *logic programming language* (LP). Instantiating Kowalski's maxim [50]:

Proof-search = Reductive Logic + Control

Here reductive logic provides the structure of the computation and control is the mechanism determining the individual steps. There are essentially two parts which must be decided: which rule to use, and which set of sufficient premisses to choose upon application. A mathematically appealing conceptional framework via the use of coalgebra for the former, and choice functions for the latter, is offered in Section 4, for now we offer a more traditional presentation.

The original logic programming language (HcLP [49]) is still widely used in the many Prolog variants that are available. It is based on the Horn clause fragment of first-order logic and uses Cut as the *only* rule, thereby solving the first control problem. The second problem is solved by showing that the use of fixed *selection functions* [52] result in a terminating procedure. This landmark result may perhaps be regarded as the second theorem of reductive logic, the first being Gentzen's completeness of analytic proofs [24].

A more general approach to proof-search is *goal-directed* proof-search which can be defined for a sequent $P \implies G$ as follows:

- If G is complex, apply right rules indefinitely until one has sub-goals $P_i \implies A_i$, where the A_i are atomic goals;
- If G is atomic, *resolve* it relative to the program by invoking a left rule.

Proofs constructed subject to this strategy are called *uniform* proofs [60, 76], and may be generalized even further, resulting in the *focusing principle* [7], which is now a common phenomenon in proof theory. Though BI's sequent calculus admits the focusing principle [25], we choose to work with the simpler case as it is sufficient for illustrating how proof-search arises from the perspective of reductive logic.

Definition 2.7 (Uniform Proof)**.** *A proof is said to be uniform when, in its upward reading, it is composed of phases where all available right rules are applied before left rules (including structural rules).*

Not all valid sequents in BI admit uniform proofs, but the existence can be guaranteed for the clausal *hereditary Harrop* fragment.

Definition 2.8 (Hereditary Harrop BI)**.** *Let D be a metavariable for a* definite clause*, G for a goal formula:*

$$\begin{array}{lcl} D & ::= & \top^* \mid \top \mid A \in \mathbb{A} \mid D \wedge D \mid D * D \mid G \to A \mid G \mathrel{-\!\!*} A \\ G & ::= & \top^* \mid \top \mid A \in \mathbb{A} \mid G \wedge G \mid G \vee G \mid G * G \mid D \to G \mid D \mathrel{-\!\!*} G \end{array}$$

Unfortunately, in BI, uniform proof-search is not sufficient for goal-directedness. The problem can be remedied by a judicious goal-directed instance of Cut which creates two branches that each are also independently goal-directed. As a simulation of reasoning, this transformation represents the appeal to a lemma specifically to prove a desired goal form a collection hypotheses; a perspective which has a mechanical implementation in HcLP by a device called *tabling* [61].

Definition 2.9 (Resolution Proof)**.** *A uniform proof is a resolution proof if the right sub-proof of an implication rule consists of a series of weakening followed by a Cut on the atom defined by the implication.*

Example 2.10. The following proof is uniform, but not goal-directed as the goal A is not principle in the first $\mathrel{-\!\!*}\, L$ inference:

$$\dfrac{B \implies B \qquad \dfrac{\dfrac{\dfrac{A \implies A}{\varnothing_\times, A \implies A}\, E}{\top^*, A \implies A}\, \top^* L}{B, B \mathrel{-\!\!*} \top^*, A \implies A}\, \mathrel{-\!\!*} L}$$

The problem can be remedied by the introduction of a cut,

$$\dfrac{\dfrac{B \implies B \qquad \dfrac{\varnothing_\times \implies \top^*}{\top^* \implies \top^*}\top^* L}{B, B \multimap \top^* \implies \top^*} \multimap L \qquad \dfrac{\dfrac{A \implies A}{\varnothing_\times, A \implies A} E}{\top^*, A \implies A}\top^* L}{B, B \multimap \top^*, A \implies A}\,\text{Cut}$$

More complicated cases include the possibility that A is the atom defined by the implication, in which case one can also make a judicious use of Cut to keep the proof goal-directed. ■

The control régime of resolution proofs can be enforced by augmenting $\rightarrow L, \multimap L$ and Ax to rules to encode the necessary uses of Cut such that resolution proofs are identified with uniform proofs in the system which has the replacement rules. For multiplicative implication, the use of Cut then allows porting the additional multiplicative content to the left sub-proof, resulting in a single thread of control.

Lemma 2.11 (Based on Armelín [9, 10])**.** *The following rules, where $\alpha \in \{\top, \top^*\} \cup \mathbb{A}$, are admissible, and replacing $\rightarrow L, \multimap L$ and Ax with them in **LBI** does not affect the completeness of the system, with respect to the hereditary Harrop fragment:*

$$\dfrac{P; G \rightarrow \alpha \implies G \qquad Q \implies \top^*}{Q, (P; G \rightarrow \alpha) \implies \alpha}\, Res_1 \qquad \dfrac{P \implies G}{P, G \multimap \alpha \implies \alpha}\, Res_2 \qquad \dfrac{\Gamma \implies \top^*}{\Gamma, \alpha \implies \alpha}\, Res_3$$

Moreover, uniform proofs in the resulting system are complete for the hereditary Harrop fragment of BI.

2.3 Operational Semantics

To distinguish the metatheory of BI from the logic programming language, we use $P \vdash G$ to denote that a goal G is being queried relative to a program P. To enforce the structure of uniform proof-search, we restrict the system modified according to Lemma 2.11 further resulting in the *resolution* system.

Definition 2.12 (Program, Goal, Configuration, State)**.** *A program P is any bunch in which formulas are definite clauses; and, a goal is a (goal) formula G. A configuration is either empty (denoted $\square$), or is pair of a program P and goal G denoted $P \vdash G$; and a state is a list of configurations. The set of all programs is denoted by $\mathbb{P}$.*

Milner's *tactics and tacticals* [63] provide a conceptual theory to support reductive logic proof-search and its mechanization. A tactic (τ) is a mapping taking a configuration to a list of next configurations, together with a deduction operator verifying the correctness of the reduction. A tactical is a composition of tactics.

In the following labelled transition system, the C_i are configurations, the S_i are states, $\sqcup$ is concatenation of lists, and δ is a deduction operator (i.e., a rule):

$$\dfrac{C_0 \longrightarrow S_0 \qquad ... \qquad C_n \longrightarrow S_n}{[C_0, ..., C_n] \longrightarrow S_0 \sqcup ... \sqcup S_n} \qquad \dfrac{\delta(P_0 \vdash G_0, ..., P_n \vdash G_n) = P \vdash G}{P \vdash G \longrightarrow [P_0 \vdash G_0, ..., P_n \vdash G_n]}\,\tau$$

In practice, we are given $P \vdash G$ and compute the list of sufficient premisses, and not the other way around, thus the condition of τ ought to be a reduction operator,

$$\dfrac{[P_0 \vdash G_0, ..., P_n \vdash G_n] \in \rho(P \vdash G)}{P \vdash G \longrightarrow [P_0 \vdash G_0, ..., P_n \vdash G_n]}\,\tau$$

Of course, deduction and reduction operators are dual meaning that it is only the use of τ that is unchanged. Note the non-determinism, which introduces the use of a *choice* (or, with some particular agenda, *selection*) function.

We restrict the application of the inference rules syntactically so that they may only be applied correctly with respect to resolution proofs. For example, a left rule can only be applied when the goal is an atom, in which case the the positive left rules ($\top L, \top^* L, \wedge L, *L$) can be applied eagerly, handled by clausal decomposition.

Definition 2.13 (Clausal Decomposition)**.** *Clausal decomposition of programs is as follows:*

$$\begin{array}{lll} [\top^*] := [\varnothing_\times] := \varnothing_\times & [\top] := [\varnothing_+] := \varnothing_+ & [A] := A \\ [P; Q] := [P]; [Q] & [D_1 \wedge D_2] := [D_1]; [D_2] & [G \rightarrow A] := G \rightarrow A \\ [P, Q] := [P], [Q] & [D_1 * D_2] := [D_1], [D_2] & [G \multimap A] := G \multimap A \end{array}$$

$P \vdash \top$	$\Leftarrow$	Always
$P \vdash \top^*$	$\Leftarrow$	$[P] \equiv \varnothing_\times; R$
$P \vdash A$	$\Leftarrow$	$[P] \leqq S, A$ and $S \vdash \top^*$
$P \vdash A$	$\Leftarrow$	$[P] \leqq Q, G \mathrel{-\!\!*} A$ and $Q \vdash G$
$P \vdash A$	$\Leftarrow$	$[P] \leqq S, (Q; G \to A)$ and $S \vdash \top^*$ and $Q \vdash G$
$P \vdash G_1 \vee G_2$	$\Leftarrow$	$P \vdash G_1$ or $P \vdash G_2$
$P \vdash G_1 \wedge G_2$	$\Leftarrow$	$P \vdash G_1$ and $P \vdash G_2$
$P \vdash G_1 * G_2$	$\Leftarrow$	$P \equiv R; (Q, R)$ and $Q \vdash G_1$ and $R \vdash G_2$
$P \vdash D \to G$	$\Leftarrow$	$P; [D] \vdash G$
$P \vdash D \mathrel{-\!\!*} G$	$\Leftarrow$	$P, [D] \vdash G$

Figure 3: Resolution System

It remains to apply either an implication left rule or an axiom, which is made goal directed by Lemma 2.11. Their completeness requires the use of weakening, so we introduce a weak coherence ordering ($\leqq$) as follows:

$$P \leqq Q \iff \dfrac{\dfrac{P \implies \bot}{\{\text{ Weakening and Exchange }\}}}{Q \implies \bot}$$

Though the use of this ordering seems to reintroduce a lot of non-determinism, the choice it offers is still captured by the use of a selection function. That is, once a clause has been *selected* in the program, the weakenings may be performed in a *goal-directed* way to bring the clause to the top of the bunch. For example, for the $\mathrel{-\!\!*}$ resolution, one may alternate phases of removing additive data in the neighbourhood of the clause and permuting data to makes the desired context-former principal.

Example 2.14. The following may illustrate the process:

$$\dfrac{\dfrac{\dfrac{(Q_0, (Q_1; Q_2)), G \mathrel{-\!\!*} \alpha \implies \bot}{Q_0, ((Q_1; Q_2), G \mathrel{-\!\!*} \alpha) \implies \bot}\,\mathrm{E}}{Q_0, ((Q_1; Q_2), (Q_3; (G \mathrel{-\!\!*} \alpha)) \implies \bot}\,\mathrm{W}}{Q_0, ((Q_1; Q_2), (Q_3; (Q_4; G \mathrel{-\!\!*} \alpha)) \implies \bot}\,\mathrm{W}$$

Anything additively combined with the clause is removed (using weakening), and when something is multiplicatively combined, the bunch is re-ordered so that the context-former of the clause is principal. In the case in which Q_0 has been additively combined, it would also have to be removed . ■

Definition 2.15 (Resolution System)**.** *The resolution system is composed of the rules in Figure 3. The first two rules are called the initial rules, the following three the resolution rules (Cut-resolution, $\mathrel{-\!\!*}$-resolution, and $\to$-resolution respectively), and the final five are the decomposition rules.*

Theorem 2.16 (Soundness and Completeness)**.** *Let P be a program and G a goal.*

- *Soundness. Any execution of $P \vdash G$ is a resolution proof of $P \implies G$.*
- *Completeness. Any resolution proof of $P \implies G$ is an execution of $P \vdash G$.*

Proof. This follows from Lemma 2.11. □

The traditional approach for applying a selection function follows from the application of a backtracking schedule. For example, taking contexts to be lists one may simply choose to always attempt the leftmost clause first when using a resolution rule; and, having made a choice, one then progresses until success or failure of the search, returning to an earlier stage in the computation in case of the latter. This forms one possibility which has been called *depth-first search with leftmost selection* [56, 51]. Another example of a schedule is *breadth-first search with leftmost selection*,

where after one step of reduction one immediately backtracks so that every possibility is tried as soon as possible. These two choices are the extremes of a range of possibilities which have different advantages and disadvantages including complexity [69]. The later is always safe, that is will always terminate when there is a proof, but the same is not true for the former as seen in Section 4.3. Whatever the choice, assuming a safe one is chosen, the result is the programming language hHBI.

The use of a selection function is sufficient to make proof-search effective, but is limited in that such functions are fundamentally nothing more than a prescribed pattern of *guessing*. To improve the situation one may study *control* as a phenomenon in itself; for example, one may use proof theory to restrict the choices as was done with the restriction to uniform proofs. In Section 4, we offer a framework in which multiple choices can be handled simultaneously, resulting in a partial determinization of the proof-search procedure. This method works in cases in which other control mechanisms do not.

3 Denotational Semantics

Traditionally, formal systems are given semantics of two different flavours: operational semantics, which specifies the local behaviour, and denotational semantics, which maps expressions into abstract mathematical entities. A basic requirement of the denotational semantics is that it is *faithful* — meaning that the mathematical structures respect the behaviour prescribed by the operational semantics — and that it is *adequate* — meaning that the dynamics of the denotations determine the operational semantics. The most exacting requirement is that equality in the denotational semantics characterizes equivalence in syntax, in which case the denotational semantics is said to be a *full abstraction*.

A standard construction of a denotational semantics for a programming language is the greatest fixed point of behaviour (see [84] for more details). Often, including for hHBI, all the constructs of the formal system are finite, so the semantic domain admits an inductive algebraic formulation: for hHBI, *the least Herbrand model*.

Below, we implement a strategy outlined by Apt [8], employing the method presented by Miller [58] for this construction. The correctness of the semantics is offered in Theorem 3.4, where we use the nomenclature of mathematical logic for the notions of faithfulness and adequacy: soundness and completeness.

3.1 Denotations and Interpretations

We develop a denotational semantics of hHBI within the setting of resource reading of BI by instantiating the resource model given in Definition 2.3. What is required is an *interpretation* of programs as some collection of resources. A simple choice is to interpret a given program as the set of atomic formulas which it satisfies, this being a subset of the Herbrand base ($\mathbb{H}$). To be precise, the models are given by a special interpretation $J : \mathbb{P} \to \mathcal{P}(\mathbb{H})$ such that a particular configuration executes if and only if the programs satisfies the goal,

$$P \vdash G \iff J(P) \models G$$

In the propositional case studied above, the Herbrand base $\mathbb{H}$ is the set of atomic propositions $\mathbb{A}$, though in Section 5 the language is generalized to a first-order setting and it's semantics can be analogous handled by choosing the set of ground formulas instead.

To give a resource frame, one must be able to combine and compare resource elements (here, programs). The first comes for free using the multiplicative context-former (or equivalently, multiplicative conjunction), but the second is more subtle. From a programming perspective, the ordering follows from the fact that a program may evolve during execution, and since execution is determined by provability, one may simply choose the *Cut* ordering:

$$P \preccurlyeq Q \iff P \vdash \lfloor Q \rfloor$$

The ordering makes sense because it is *internally monotone*; that is, if a program gets larger then the set of associated atoms does not decrease. The resulting frame is adequate for a resource model when interpretation satisfies a basic precondition,

$$A \in I(P) \implies P \vdash A$$

A primitive example of such an interpretation is $I_\perp : P \mapsto \emptyset$.

$$
\begin{array}{lcl}
I, P \Vdash \top & \iff & \text{Always} \\
I, P \Vdash \top^* & \iff & P \equiv \varnothing_\times; R \\
I, P \Vdash A & \iff & A \in I(P) \\
I, P \Vdash G_1 \vee G_2 & \iff & I, P \Vdash G_1 \text{ or } I, P \Vdash G_2 \\
I, P \Vdash G_1 \wedge G_2 & \iff & I, P \Vdash G_1 \text{ and } I, P \Vdash G_2 \\
I, P \Vdash G_1 * G_2 & \iff & \exists Q, R \in \mathbb{P} :\ P \preccurlyeq Q \circ R \text{ and } I, Q \Vdash G_1 \text{ and } I, R \Vdash G_2 \\
I, P \Vdash D \to G & \iff & I, (P; [D]) \Vdash G \\
I, P \Vdash D \mathrel{-\!\!*} G & \iff & I, (P, [D]) \Vdash G
\end{array}
$$

Figure 4: Program Satisfaction

Definition 3.1 (Monoid of Programs, Interpretation)**.** *The monoid of programs is the structure $\langle \mathbb{P}, \preccurlyeq, \circ, \varnothing_\times \rangle$, where $\circ$ is multiplicative composition. An interpretation of the monoid of programs is an order reversing mapping $I : \mathbb{P} \to \mathcal{P}(\mathbb{A})$.*

Unfortunately, the use of the Cut ordering reintroduces non-analyticity into the semantics. Some of this may be handled using the adjunction between the implications and their respective conjunctions, resulting in the simpler *program* satisfaction relation of Figure 4. One can verify by induction on goals that it is still internally monotone; that is, for any interpretation I, if $P \preccurlyeq Q$ then $I, Q \Vdash G$ implies $I, P \Vdash G$.

However, since the accessibility condition of the implication clauses in program satisfaction is a restriction of the corresponding clauses in the frame satisfaction relation — that is, program satisfaction replaces the accessibility condition by a particular program extension — it actually forms a new semantics altogether. The advantage is that this new semantics is more amenable to computation.

Lemma 3.2 (Adequacy)**.** *If $A \in I(P)$ implies $P \vdash A$, then $I, P \Vdash G$ implies $P \vdash G$*

Proof. By induction on the structure of the goal formula.

Base case. There are the sub-cases to consider for $G \in \{\top, \top^*\} \cup \mathbb{A}$, but in either case it follows by equivalence of the respective clauses between program satisfaction and the resolution system.

Inductive case.

- $G = G_1 \vee G_2$. By definition, $I, P \Vdash G_1$ or $I, P \Vdash G_2$, so by the induction hypothesis, $P \vdash G_1$ or $P \vdash G_2$, whence $P \vdash G_1 \vee G_2$.
- $G = G_1 \wedge G_2$. By definition, $I, P \Vdash G_1$ and $I, P \Vdash G_2$, so by the induction hypothesis, $P \vdash G_1$ and $P \vdash G_2$, whence $P \vdash G_1 \wedge G_2$.
- $G = G_1 * G_2$. By definition, $P \preccurlyeq Q \circ R$ with $I, Q \Vdash G_1$ and $I, R \Vdash G_1$. It follows from the induction hypothesis that $Q \vdash G_1$ and $R \vdash G_2$, so that $Q, R \vdash G_1 * G_2$, whence $P \vdash G_1 * G_2$.
- $G = D \to G'$. By definition, $P; [D] \Vdash G'$, so by the induction hypothesis, $P; [D] \vdash G'$, so $P \vdash D \to G'$.
- $G = D \mathrel{-\!\!*} G'$. By definition, $P, [D] \Vdash G'$, so by the induction hypothesis, $P, [D] \vdash G'$, so $P \vdash D \to G'$. □

The restriction to program satisfaction now increases the set of possible frames, since, for example, one no longer requires bifunctoriality. Therefore, another candidate for the ordering on frames is simply to make the clauses of program satisfaction match the clauses for resolution proofs, so one has the following extension ordering:

$$P \leq Q \iff \exists R \in \mathbb{P} :\ P \equiv Q; R$$

However, such a choice does *not* yield a resource frame (precisely because the relation is not bifunctorial), but does yield an adequate semantics with respect to program satisfaction. The extension ordering is strictly *coarser* than the provability ordering; that is, $P \leq Q \implies P \preccurlyeq Q$, but not the other way around as witnessed by the following:

$$\dfrac{\dfrac{\varnothing_\times \vdash \top^*}{\varnothing_\times; \top^* \to A; \varnothing_+ \vdash A}}{\varnothing_\times; \top^* \to A \vdash \top \to A}$$

One has $\varnothing_\times; \top^* \to A \preccurlyeq \top \to A$, but the former is not an extension of the latter. In fact, the two ordering both give full abstractions relative to different notions of equality which predictably are equality of programs and equality of declarations:

$$P = Q \mod \leq \iff P \equiv Q$$
$$P = Q \mod \preccurlyeq \iff \varnothing_\times \vdash \lfloor P \rfloor \mathrel{*\!\!-\!\!*} \lfloor Q \rfloor$$

3.2 Least Fixed Point

Reading the resources semantics as a possible-worlds interpretation of the logic, we use the symbols $w, u, v...$ for programs. The adequacy condition requires that a world is inhabited by nothing other than the propositional formulas that it satisfies, given by an interpretation $I(w)$.

Observe that since $\mathcal{P}(\mathbb{A})$ forms a complete lattice, so do interpretations (with $I_\perp$ as the bottom),

$$\begin{array}{lcl} I_1 \sqsubseteq I_2 & \iff & \forall w (I_1(w) \subseteq I_2(w)) \\ (I_1 \sqcup I_2)(w) & := & I_1(w_1) \cup I_2(w_2) \\ (I_1 \sqcap I_2)(w) & := & I_1(w_1) \cap I_2(w_2) \end{array}$$

The derived ordering essentially requiring that larger interpretations see more of the worlds than the smaller interpretations.

Combining the adequacy condition in Lemma 3.2 with the well-behaved structure of the resolution system suggests that one can inductively improve the vacuous interpretation $I_\perp$ by unfolding the resolution system. To this end, consider the T-operator on interpretations.

Definition 3.3 (T-Operator)**.** *The T operator is defined as follows:*

$$T(I)(w) := \{A \mid [w] \leqq u, A \text{ and } I, u \Vdash \top^*\} \cup \tag{1}$$
$$\{A \mid [w] \leqq u, G \mathrel{*\!\!-\!\!*} A \text{ and } I, u \Vdash G\} \cup \tag{2}$$
$$\{A \mid [w] \leqq u, (v; G \to A) \text{ and } I, u \Vdash \top^* \text{ and } I, (v; G \to A) \Vdash G\} \tag{3}$$

Observe that the three parts of the definition correspond exactly to the resolution clauses of execution, so T precisely incorporates *one* resolution step. Applying it indefinitely, therefore, corresponds to performing an arbitrary number of resolutions. This is handled mathematically by the use of a least fixed-point.

It follows from the Knaster-Tarski Theorem [45, 83, 8] that if T is monotone and continuous, then the following limit operator is well-defined:

$$T^\omega(I_\perp) := I_\perp \sqcup T(I_\perp) \sqcup T^2(I_\perp) \sqcup T^3(I_\perp) \sqcup \dots$$

These conditions are shown to hold in Lemma 3.5(3) and Lemma 3.5(4), respectively. The interpretation may thenceforth be suppressed, so that $P \Vdash G$ denotes $T^\omega(I_\perp), P \Vdash G$.

The completeness of the semantics follows immediately from the definition of T, since it simply observes the equivalence between resolution and application of the T-operator; that is, that T extends correctly. Soundness, on the other hand, requires showing that every path during execution is eventually considered during the unfolding.

Theorem 3.4 (Soundness and Completeness)**.** $P \vdash G \iff P \Vdash G$

Proof of Completeness ($\Longleftarrow$). From Lemma 3.2, it suffices to show that the adequacy condition holds:

$$A \in T^\omega I_\perp(P) \implies P \vdash A$$

It follows from Lemma 3.5(2) that the antecedent holds only if there exists $k \in \mathbb{N}$ such that $T^k I_\perp, P \Vdash A$. We proceed by induction on k, the base case $k = 0$ being vacuous as $I_\perp(P) = \emptyset$. For the inductive step, observe that $A \in T^k I_\perp(P)$ only if at least one of (1), (2), or (3) holds in Definition 3.3 for $I = T^{k-1} I_\perp$. In either case, the result follows from the induction hypothesis and the correspondence between (1), (2), and (3), with the resolution clauses of Figure 3. □

The completeness proof is a repackaging of Miller's familiar ordinal induction proof [58]. The difference is simply to pull out the sub-induction on the structure of the formula as an instance of Lemma 3.2.

Proof of Soundness ($\Longrightarrow$). By induction on the height N of executions.

Base case: $N = 1$**.** It must be that the proof of $P \vdash G$ follows from the application of an initial rule. Hence, either $G = \top$, or $G = \top^*$ and $P \equiv \varnothing_\times; R$. In either case, $P \Vdash G$ follows immediately.

Inductive case. We consider each of the cases for the last inference.

- Cut-resolution. By definition, $[P] \leqq Q, A$, where $Q \vdash \top^*$ with height $N' < N$. So, by the induction hypothesis, $Q \Vdash \top^*$, so that $A \in T(T^\omega(I_\bot))(P) = T^\omega(I_\bot)(P)$, whence $P \Vdash A$.
- $-\!\!*$-resolution. By definition, $[P] \leqq Q, G' -\!\!* A$, where $Q \vdash G'$ with height $N' < N$. So, by the induction hypothesis, $Q \Vdash G'$, so that $A \in T(T^\omega(I_\bot))(P) = T^\omega(I_\bot)(P)$, whence $P \Vdash A$.
- $\rightarrow$-resolution. By definition, $[P] \equiv Q, (R; G' \rightarrow A)$, where $Q \vdash \top^*$ and $R \vdash G'$, with heights $N', N'' < N$. So, by the induction hypothesis, $Q \Vdash \top^*$ and $R \Vdash G'$, hence $A \in T(T^\omega I_\bot)(P) = T^\omega I_\bot(P)$, whence $P \Vdash A$.
- $G = G_1 \vee G_2$. By definition, $P \vdash G_i$ for some $i \in \{1, 2\}$ with height $N' < N$. So, by the induction hypothesis, $P \Vdash G_i$ for some $i \in \{1, 2\}$, so $P \Vdash G_1 \vee G_2$.
- $G = G_1 \wedge G_2$. By definition, $P \vdash G_i$ for all $i \in \{1, 2\}$ with heights $N', N'' < N$. So, by the induction hypothesis, $P \Vdash G_i$ for all $i \in \{1, 2\}$, so $P \Vdash G_1 \wedge G_2$.
- $G = G_1 * G_2$. By definition, $P \equiv Q \circ R$ where $Q \vdash G_1$ and $R \vdash G_2$ with heights $N', N'' < N$. So, by the induction hypothesis, $Q \Vdash G_1$ and $R \Vdash G_2$, so from reflexivity of provability $P \Vdash G_1 * G_2$.
- $G = D \rightarrow G'$. By definition, $P; [D] \vdash G'$ with height $N' < N$. So, by the induction hypothesis, $P; [D] \Vdash G'$, so $P \Vdash D \rightarrow G'$.
- $G = D -\!\!* G'$. By definition, $P, [D] \vdash G'$ with height $N' < N$. So, by the induction hypothesis, $P, [D] \Vdash G'$, so $P \Vdash D -\!\!* G'$. □

The remainder of this subsection shows that $T^\omega(I_\bot)$ is well defined.

Lemma 3.5. *Let $I_0 \sqsubseteq I_1 \sqsubseteq \ldots$. be a collection of interpretations, let $w \in \mathbb{P}$, and let G be a goal, then,*

1. $I_1, w \Vdash G \implies I_2, w \Vdash G$. *(Persistence of Satisfaction)*
2. $\sqcup_{i=1}^{\infty} I_i, w \Vdash G \implies \exists k \in \mathbb{N} : I_k, w \Vdash G$. *(Compactness of Satisfaction)*
3. $T(I_0) \sqsubseteq T(I_1)$. *(Monotonicity of T)*
4. $T(\sqcup_{i=0}^{\infty} I_i) = \sqcup_{i=0}^{\infty} T(I_i)$. *(Continuity of T)*

Proof of Lemma 3.5(1). By induction on the structure of the goal formula. Throughout, assume $I_1, w \Vdash G$, otherwise the statement holds vacuously.

Base case. The cases $G \in \{\top, \top^*\}$ are immediate since satisfaction is independent of interpretation, meanwhile the case in which $G \in \mathbb{A}$ follows from the definition of the ordering on interpretations.

Inductive case. Assume, inductively, that the claim holds for any goal smaller than G independent of program.

- $D \rightarrow G$. By definition, $I_1, w; [D] \Vdash G$, so, by the induction hypothesis, it follows that $I_2, w; [D] \Vdash G$, thus $I_2, w \Vdash D \rightarrow G$.
- $D -\!\!* G$. By definition, $I_1, (w, [D]) \Vdash G$, so, by the induction hypothesis, it follows that $I_2, (w, [D]) \Vdash G$, thus $I_2, w \Vdash D -\!\!* G$.
- $G_1 * G_2$. By definition, $w \preccurlyeq u \circ v = u, v$ such that $I_1, u \Vdash G_1$ and $I_1, v \Vdash G_2$ and so, by the induction hypothesis, it follows that $I_2, u \Vdash G_1$ and $I_2, v \Vdash G_2$, so that $I_2, w \Vdash G_1 * G_2$.
- $G_1 \wedge G_2$. By definition, $I_1, w \Vdash G_1$ and $I_1, w \Vdash G_2$, so, by the induction hypothesis, $I_2, w \Vdash G_1$ and $I_2, w \Vdash G_2$, and so $I_2, w \Vdash G_1 \wedge G_2$.

- $G_1 \vee G_2$. By definition, $I_1, w \Vdash G_1 \vee G_2$, then $I_1, w \Vdash G_1$ or $I_1, w \Vdash G_2$, so, by the induction hypothesis, $I_2, w \Vdash G_1$ or $I_2, w \Vdash G_2$, and so $I_2, w \Vdash G_1 \vee G_2$. □

Proof of Lemma 3.5(2). By induction on the structure of the goal formula. Throughout, assume $\sqcup_{i=1}^{\infty} I_i, w \Vdash G$ otherwise the statement holds vacuously.

Base case. The cases $G \in \{\top, \top^*\}$ are immediate since satisfaction is independent of interpretation. For the case in which $G \in \mathbb{A}$, note that $\left(\sqcup_{i=0}^{\infty}\right)(w) = \bigcup_{i=0}^{\infty} I_i(w)$, so by the definition of satisfaction $G \in I_k(w)$, for some k, and so $I_k, w \Vdash G$.

Inductive case. Assume, inductively, that the claim holds for any goal smaller than G independent of program.

- $G_1 \vee G_2$. By definition, $\sqcup_{i=1}^{\infty} I_i, w \Vdash G_1$ or $\sqcup_{i=1}^{\infty} I_i, w \Vdash G_2$, so, by the induction hypothesis, $\exists k \in \mathbb{N}$ such that $I_k, w \Vdash G_1$ or $I_k, w \Vdash G_2$, whence $I_k, w \Vdash G_1 \vee G_2$.
- $G_1 \wedge G_2$. By definition, $\sqcup_{i=1}^{\infty} I_i, w \Vdash G_1$ and $\sqcup_{i=1}^{\infty} I_i, w \Vdash G_2$, so, by the induction hypothesis, $\exists m, n \in \mathbb{N}$ such that $I_m, w \Vdash G_1$ and $I_n, w \Vdash G_2$. Let $k = \max(m, n)$, from Lemma 3.5(1), it follows that $I_k, w \Vdash G_1$ and $I_k \Vdash G_2$. Therefore, $I_k, w \Vdash G_1 \wedge G_2$.
- $G_1 * G_2$. By definition, $w \preccurlyeq u, v$ such that $\sqcup_{i=1}^{\infty} I_i, u \Vdash G_1$ and $\sqcup_{i=1}^{\infty} I_i, v \Vdash G_2$. It follows from the induction hypothesis, that $\exists m, n \in \mathbb{N}$ such that $I_m, u \Vdash G_1$ and $I_n, u \Vdash G_1$. Let $k = \max(m, n)$, it follows from Lemma 3.5(1) that $I_k, u \vdash G_1$ and $I_k, v \Vdash G_2$. Therefore, $I_k, (u, v) \Vdash G_1 * G_2$, thus $I_k, w \Vdash G_1 * G_2$.
- $D \to G'$. By definition, $\sqcup_{i=1}^{\infty} I_i, (w; [D]) \Vdash G_1$, so, by the induction hypothesis, $\exists k \in \mathbb{N}$ such that $I_k, (w; [D]) \Vdash G'$. Thus, by definition, $I_k, w \Vdash D \to G'$.
- $D \mathrel{-\!\!*} G'$. By definition, $\sqcup_{i=1}^{\infty} I_i, (w, [D]) \Vdash G'$, so, by the induction hypothesis, $\exists k \in \mathbb{M}$ with $I_k, (w, [D]) \Vdash G'$. Thus, by definition, $I_k, w \Vdash D \mathrel{-\!\!*} G'$. □

Proof of Lemma 3.5(3). Let $w \in \mathbb{P}$ be arbitrary, and suppose $A \in TI_0(w)$, then we must show $A \in TI_1(w)$. There are three cases, corresponding to the definition of T.

(1) Suppose $[w] \leqq u, A$ such that $I_0, u \Vdash \top^*$. Then, from Lemma 3.5(1), it follows that $I_1, u \Vdash I$, so that $A \in TI_1(w)$.

(2) If $[w] \leqq u, G \mathrel{-\!\!*} A$ such that $I_0, u \Vdash G$, then, from Lemma 3.5(1), it follows that $I_1, u \Vdash G$, so that $A \in TI_1(w)$.

(3) If $[w] \leqq u, (v; G \to A)$ such that $I_0, u \Vdash \top^*$ and $I_0, v \vdash G$, then, from Lemma 3.5(1), it follows that $I_1, u \Vdash \top^*$ and $I_1, v \vdash G$, so that $A \in TI_1(w)$. □

Proof of Lemma 3.5(4). We consider each direction of the inclusion independently.

First, $\sqcup_{i=0}^{\infty} T(I_i) \sqsubseteq T(\sqcup_{i=0}^{\infty} I_i)$. Let $j \geq 0$, then $I_j \sqsubseteq \sqcup_{i=1}^{\infty} I_i$, so from Lemma 3.5(3) it follows that $T(I_j) \sqsubseteq T(\sqcup_{i=1}^{\infty} I_i)$. Since j was arbitrary, $\sqcup_{i=1}^{\infty} TI_i \sqsubseteq T(\sqcup_{i=1}^{\infty} I_i)$

Second, $T(\sqcup_{i=0}^{\infty} I_i) \sqsubseteq \sqcup_{i=0}^{\infty} T(I_i)$. Let $w \in \mathbb{P}$ be arbitrary, and suppose $A \in T(\sqcup_{i=1}^{\infty} I_i)(w)$, then it suffices to $\exists k \in \mathbb{N}$ such that $A \in T(I_k)$. Here are three cases, corresponding to the definition of T.

(1) If $[w] \leqq u, A$ and $\sqcup_{i=1}^{\infty} I_i, u \Vdash \top^*$, then it follows from Lemma 3.5(2) that $\exists k \in \mathbb{N}$ such that $I_k, u \Vdash \top^*$. Therefore, by definition, $A \in TI_k(w)$.

(2) If $[w] \leqq u, G \mathrel{-\!\!*} A$ and $\sqcup_{i=1}^{\infty} I_i, u \Vdash G$, then it follows from Lemma 3.5(2) that $\exists k \in \mathbb{N}$ such that $I_k, u \Vdash \top^*$. Therefore, by definition, $A \in TI_k(w)$.

(3) If $[w] \leqq u, (v; G \to A)$ and $\sqcup_{i=1}^{\infty} I_i, u \Vdash \top^*$ and $\sqcup_{i=1}^{\infty} I_i, v \Vdash G$, then it follows from Lemma 3.5(2) that $\exists m, n \in \mathbb{N}$ such that $I_m, u \Vdash \top^*$ and $I_m, v \Vdash G$. Let $k = \max(m, n)$, then from 3.5(1), it follows that $I_k, u \Vdash \top^*$ and $I_k, v \Vdash G$, so $A \in TI_k(w)$. □

4 Coalgebraic Semantics

The fixed point semantics for hHBI is appealing in its conceptional simplicity and relationship to the resource interpretation of BI, but leaves something to be desired as a semantics of proof-search as it lacks explicit witnessing of

the control structures. Therefore we look for a denotational semantics of the behaviour of proof-search itself; that is, an abstraction of the operational semantics. In the past, models for classical, intuitionistic, and linear logics have included dialogue games [78, 62] in the spirit of Abramsky's full abstraction for PCF [5]. However, in the case of logic programming, the uniform treatment of Turi and Plotkin [84] via coalgebraic methods, an essential feature of modern approaches to operational semantics in general, has been successfully implemented for HcLP [46, 47, 48, 12], and is the method demonstrated here. Though we give a coalgebraic model of proof-search specifically for the resolution system of the hereditary Harrop fragment of BI, which defines the behaviour of hHBI, we make every attempt to keep the approach general. The choice of coalgebra to model proof-search is both natural and sensible when one considers it an abstract mathematical framework for the study states and transitions.

The deductive view of logic is inherently algebraic; that is, rules in a proof system can be understood as functions, called deduction operators, which canonically determine algebras for a functor. The space of valid sequents is defined by the recursive application of these operators, which instantiates the inductive definition of proofs. In contrast, the reductive paradigm of logic is coalgebraic; that is, the reduction operators can be mathematically understood as coalgebras. The much larger space of sequents (or configurations) explored during proof-search is corecrusively generated, and reductions are coinductively defined. This framework immediately allows the modelling of an interpretation of hHBI with parallel execution, analogous to the treatment of HcLP discussed above [46, 47, 48, 12].

The parallel semantics and the familiar sequential one using a backtracking-schedule share a common problem: the size of the set of premisses upon application of a reduction operator. We propose a general approach to ameliorate the situation using algebraic constraints. The mechanism can be understood in the coalgebraic setting as a lifting of a single step of proof-search and works on the metalevel of reductive logic, so is independent of the particular execution semantics (sequential or parallel). In Section 5, we show that this method closer simulates intelligent reasoning than the immediate use of selection functions.

4.1 Background

Endofunctors on the category of sets and functions are a suitable mathematical framework for an abstract notion of structure, and throughout we will use the word *functor* exclusively for such mappings. We may suppress the composition symbol and simply write $\mathcal{GF}$ for the mapping which first applies $\mathcal{F}$ and then applies $\mathcal{G}$; similarly, we may write $\mathcal{F}X$ for the application of functor $\mathcal{F}$ on a set X.

There are numerous functors used throughout mathematics and computer science, for example elements of the *flat polynomial* functors,

$$\mathcal{F} ::= \mathcal{I} \mid \mathcal{K}_A \mid \mathcal{F} \times \mathcal{F} \mid \mathcal{F} + \mathcal{F}$$

Here $\mathcal{I}$ is the identity functor (i.e., the mapping fixing both objects and functions); $\mathcal{K}_A$ is the constant functor for a given set A, which is defined by mapping any set to the set A and any arrow to the identity function on A; $\mathcal{F} \times \mathcal{G}$ is the cartesian product of $\mathcal{F}$ and $\mathcal{G}$; and, $\mathcal{F} + \mathcal{G}$ is the disjoint union of $\mathcal{F}$ and $\mathcal{G}$.

Occasionally one can transform one functor into another uniformly. That is, one can make the transformation componentwise, so that the actions on sets and function cohere.

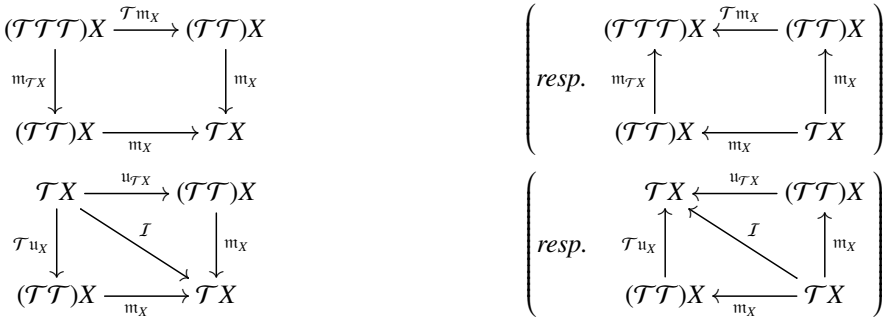
Definition 4.1 (Natural Transformation)**.** *A collection of functions indexed by sets* $\mathfrak{n} := (\mathfrak{n}_X)$ *is a natural transformation between functors* $\mathcal{F}$ *and* $\mathcal{G}$ *if and only if* $\mathfrak{n}_X : \mathcal{F}X \to \mathcal{G}X$ *and if* $f : X \to Y$ *then then the following diagram commutes:*

$$\begin{array}{ccc} \mathcal{F}X & \xrightarrow{\mathfrak{n}_X} & \mathcal{G}X \\ {\scriptstyle \mathcal{F}(f)}\downarrow & & \downarrow{\scriptstyle \mathcal{G}(f)} \\ \mathcal{F}(Y) & \xrightarrow[\mathfrak{n}_Y]{} & \mathcal{G}(Y) \end{array}$$

Every functor $\mathcal{F}$ admits at least one natural transformation called the identity: $\mathfrak{i}_X := I_{\mathcal{F}X}$, where $I_{\mathcal{F}X}$ is the identify function on $\mathcal{F}X$. As an abuse of notation, we use the notation of function types when speaking about natural transformation; that is, we may write $\mathfrak{n} : \mathcal{F} \to \mathcal{G}$ to denote that $\mathfrak{n}$ is a natural transformation between $\mathcal{F}$ and $\mathcal{G}$.

There are two particularly well-behaved classes of functors, called *monads* and *comonads*, that are useful abstractions of data-type and behaviour-type when modelling computation.

Definition 4.2 (Monad and Comonad)**.** *Let $\mathcal{T}$ be a functor. It is a (co)monad if there are natural transformations $\mathfrak{u} : \mathcal{I} \to \mathcal{T}$ and $\mathfrak{m} : \mathcal{T}^2 \to \mathcal{T}$ (resp. $\mathfrak{u} : \mathcal{T} \to \mathcal{I}$ and $\mathfrak{m} : \mathcal{T} \to \mathcal{T}^2$) satisfying the following commutative diagrams:*



The natural transformations $\mathfrak{u}$ and $\mathfrak{m}$ are often called the (co)unit and (co)multiplication of the (co)monad. There is an abundance of examples of monads; for example, the powerset functor $\mathcal{P}$, which takes sets to their powersets and functions to their direct image fuctions, is a monad whose unit is the singleton function and whose multiplication is the union operator. Simple examples of comonads are less common. However, since they are used to define behaviour-type, modelling operational semantics will involve defining one: the proof-search comonad.

Under relatively mild conditions, there is a canonical way to construct a (co)monad from a functor: the (co)free construction. Heuristically, this is the indefinite application of the functor structure until a fixed-point is reached. The cofree construction is analogous.

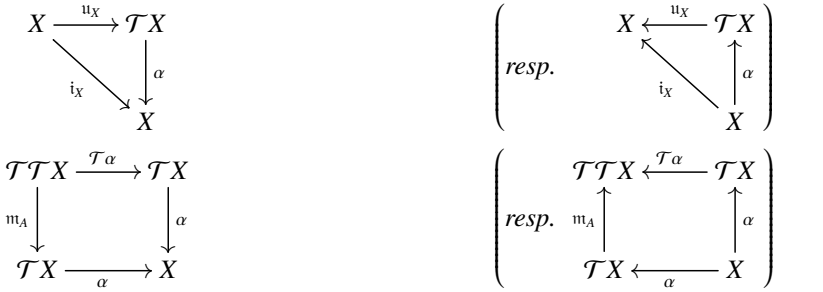
Example 4.3. Consider the functor $\mathcal{F}_A : X \mapsto \mathsf{nil} + A \times X$, where nil is the emptyset. One can generate the least fixed point $\mathcal{L}_A$ for $\mathcal{F}_A$ by the ω-chain in the following diagram:

$$\mathsf{nil} \longrightarrow \mathsf{nil} + A \times \mathsf{nil} \longrightarrow \mathsf{nil} + A \times (\mathsf{nil} + A \times \mathsf{nil}) \longrightarrow ...$$

The arrows are inductively defined by extending with the unique function out of the emptyset. The mapping $A \mapsto L_A$ defines the free functor $\mathcal{L}_A$, which can be understood as structuring elements of A as lists (identified with products). It is a monad whose unit is the single-element list constructor $a \mapsto a :: (\mathsf{nil} :: \mathsf{nil}...)$ and whose multiplication concatenation. ■

Given a space structured by a functor $\mathcal{F}$, one can define actions which respect the structure. There are two directions: either one wants the domain to be structured, in which case one has an $\mathcal{F}$-algebra, or the codomain, in which case one has a $\mathcal{F}$-coalgebra. When structure represents a data-type (resp. a behaviour-type) given by a (co)monad, one may add extra conditions on the (co)algebra that make it a co(module). These functions are used to give abstract models of data and behaviour.

Definition 4.4 (Algebra and Coalgebra)**.** *Let $\mathcal{T}$ be a functor, then a function $\alpha : \mathcal{T}X \to X$ is called an $\mathcal{T}$-algebra and any function $\beta : X \to \mathcal{T}X$ is a $\mathcal{T}$-coalgebra. If $\langle \mathcal{T}, \mathfrak{u}, \mathfrak{m} \rangle$ is a monad (resp. comonad), then α (resp. β) is a module (resp. comodule) when the following diagrams commute:*



The abstract modelling of operational semantics witnesses both the use of algebra and coalgebra: the former for specifying the constructs, and the latter for specifying the transitions. In the best case the two structures cohere, captured mathematically by the mediation of a natural transformation called a distributive law, and form a bialgebra.

Definition 4.5 (Distributive Law)**.** *A distributive law for a functor $\mathcal{G}$ over a functor $\mathcal{F}$ is a natural transformation $\partial : \mathcal{G}\mathcal{F} \to \mathcal{F}\mathcal{G}$.*

Definition 4.6 (Bialgebra)**.** *Let $\partial : \mathcal{G}\mathcal{F} \to \mathcal{F}\mathcal{G}$ be a distributive law, and let $\alpha : \mathcal{G}X \to X$ be an algebra and $\beta : X \to \mathcal{F}X$ be a coalgebra. The triple (X, α, β) is a ∂-bialgebra when the following diagram commutes:*

$$\begin{array}{ccccc} \mathcal{G}X & \xrightarrow{\alpha} & X & \xrightarrow{\beta} & \mathcal{F}X \\ {\scriptstyle \mathcal{G}\beta}\downarrow & & & & \uparrow{\scriptstyle \mathcal{F}\alpha} \\ \mathcal{G}\mathcal{F}X & & \xrightarrow[\partial_X]{} & & \mathcal{F}\mathcal{G}X \end{array}$$

There are additional coherence condition which may be applied for when one has a monad or a comonad structure.

In Turi's and Plotkin's bialgebraic models of operational semantics [84], the algebra supplies the structure of the syntax and the coalgebra supplies the behaviour of execution, and under relatively mild conditions (i.e. the coalgebra structure preserves weak pullbacks) forms are even a *full* abstractions (with respect to bisimulation).

4.2 Reduction Operators

To model proof-search coalgebraically, we first model the key components: reduction operators. These are the essential elements in reductive logic and proof-search, and they readily admit coalgebraic treatment by dualizing the algebraic perspective of deductive logic. We begin, however, by modelling the syntax of hHBI.

The construction of the list monad $\mathcal{L}$ in Example 4.3 is a model of the grammar of infinite lists,

$$\ell ::= x :: \ell$$

The variable x represents a data element from a set A, so is interpreted by the functor $\mathcal{K}_A$; the ℓ represents a list, the thing being modelled, so is interpreted by identity $\mathcal{I}$; and the :: constructor represents the pairing of the two components, so is interpreted as the product. In the BNF format, the | symbol represents a choice of productions, so it is modelled by the sum of functors, therefore the base for the *free* construction in Example 4.3 is $X \mapsto \mathsf{nil} + A \times X$.

Modelling configurations $P \vdash G$ as abstract data structures works analogously; that is, we use the productions in the context-free grammars for definite clauses and goals in Definition 2.8 to generate functors to which we apply the free construction. There are several pairs in these grammar(s) that ought to be distinguished, such as $\phi \wedge \psi$ and $\phi * \psi$, so rather than simply using a binary product, one signs the pairs, resulting in functors of the following shape:

$$\wedge : X \mapsto X \times \{\wedge\} \times X \qquad \text{and} \qquad * : X \mapsto X \times \{*\} \times X$$

The functors will be written using infix notation to simplify presentation. The X is a set of variables upon which the construction takes place, at the end of the construction the variables will be replaced by the constants $\mathbb{A}$.

Modelling the syntax of the hereditary Harrop fragment of BI is slightly more elaborate because of the mutual induction taking place over definite clauses and goals. Consequently, we first consider mappings of two variables delineating the structural difference of the two types of formulas:

$$\begin{array}{lcl} \mathcal{F}_G(X,Y) & := & X \wedge X \;+\; X * X \;+\; X \vee X \;+\; Y \to X \;+\; Y \mathrel{-\!\!*} X \;+\; \mathcal{K}_{\{\top^*\}} \;+\; \mathcal{K}_{\{\top\}} \\ \mathcal{F}_D(X,Y) & := & Y \wedge Y \;+\; Y * Y \;+\; X \to Y \;+\; X \mathrel{-\!\!*} Y \;+\; \mathcal{K}_{\{\top^*\}} \;+\; \mathcal{K}_{\{\top\}} \end{array}$$

To recover the free construction, we unfold the inductive definition simultaneously and use the first and second projection function π_1 and π_2 to put the right formulas in the right place. That is, we consider the following functor:

$$\mathcal{F}(Z) := \mathcal{F}_G(\pi_1 Z, \pi_2 Z) \times \mathcal{F}_D(\pi_1 Z, \pi_2 Z)$$

Here sets Z contain products. The functors defining $\mathcal{F}$ are sufficiently simple and the category sufficiently well-behaved that the free construction yields a limit,

$$\begin{array}{l} Z \longrightarrow Z + \mathcal{F}Z = \\ \qquad Z + \underbrace{\mathcal{F}_G(\pi_1(Z+\mathcal{F}Z),\, \pi_2(Z+\mathcal{F}Z))}_{\mathcal{F}_G(\pi_1 Z + \mathcal{F}_G Z,\, \pi_2 Z + \mathcal{F}_D Z)} \times \underbrace{\mathcal{F}_D(\pi_1(Z+\mathcal{F}Z),\, \pi_2(Z+\mathcal{F}Z))}_{\mathcal{F}_D(\pi_1 Z + \mathcal{F}_G(Z),\, \pi_2 Z + \mathcal{F}_D(Z))} \longrightarrow \ldots \end{array}$$

Each transition in the construction is the embedding of the previously constructed set within the next which contains it as a component of its disjoint union; for example, the first arrow embeds Z in $Z + \mathcal{F}(Z)$. This embedding is in fact a natural transformation $\mathcal{I} \to \mathcal{I} + \mathcal{F}$. Hence, as the construction continues more and more stages of the inductive definition of goals and definite clauses are captured, and the limit, therefore, contains all the possible goals and definite clauses at once.

Let $\hat{\mathcal{F}}$ denote the free functor for $\mathcal{F}$, then the goal formulas and definite clauses are recovered via the first and second projections,

$$\hat{\mathcal{F}}_G(X) := \pi_1 \hat{\mathcal{F}}(X, X) \qquad \hat{\mathcal{F}}_D(X) := \pi_2 \hat{\mathcal{F}}(X, X)$$

By fixing the set of atomic proposition $\mathbb{A}$ (i.e., the base of the inductive construction), the disjoint union present in the free construction means that all goal formulas and definite clauses are present in $\hat{\mathcal{F}}_G(\mathbb{A})$ and $\hat{\mathcal{F}}_D(\mathbb{A})$, respectively.

The model of bunches is comparatively simple since there is no mutual induction, so it simply requires the free monad $\hat{\mathcal{B}}$ for the functor

$$\mathcal{B} := \hat{\mathcal{D}} + \overset{\circ}{,} + \overset{\cdot}{,} + \mathcal{K}_{\{\varnothing_\times\}} + \mathcal{K}_{\{\varnothing_+\}}$$

Configurations are pairs of programs and goals, so their abstract data-structure is given by the functor $\mathcal{G}(X) := \hat{\mathcal{B}}(X) \times \hat{\mathcal{F}}(X)$. In particular, configurations are modelled by elements of $\mathcal{G}(\mathbb{A})$. Below, we use the abstract data structure and the formal grammar interchangeably.

Example 4.7. The configuration $(A \wedge \top) \mathrel{-\!\!*} A \vdash A$, where $A \in \mathbb{A}$, is modelled by the tuple $(((A, \wedge, \top), \mathrel{-\!\!*}, A), A)$, which is nothing more than the typical encoding of the syntax-tree as an ordered tree. ■

A rule R in a sequent calculus is typically given by a rule figure together with correctness conditions, and can be understood mathematically as relation $\mathbf{R}$ on sequents that holds only when the first sequent (the conclusion) is inferred from the remaining (the premisses) by the rule.

Example 4.8. The exchange rule E in **LBI** is comprised of the following figure, together with correctness condition $\Delta' \equiv \Delta$:

$$\frac{\Delta \implies \phi}{\Delta' \implies \phi}\,\mathrm{E}$$

Consider the following:

$$\begin{array}{l} (\varnothing_\times \implies A)\,\mathbf{E}\,((\varnothing_\times, \varnothing_\times) \implies A) \\ (\varnothing_\times \implies A)\,\mathbf{E}\,((\varnothing_\times; \varnothing_\times) \implies A) \end{array}$$

Both have the right shape, but only the first respects the correctness condition and so is the only one of the two that holds. ■

This understanding immediately lends itself to the modelling of a rule as a non-deterministic partial function δ, an example of the functions we called deduction operators in Section 2.3,

$$\delta : \mathcal{LG}(\mathbb{A}) \rightharpoonup \mathcal{PG}(\mathbb{A}) : \ell \mapsto \{(P, G) \in \mathcal{G}(\mathbb{A}) \mid (P, G)\,\mathbf{R}\,\ell\}$$

In the study of proof-search, we consider the inverse action, since we have a putative conclusion and look for sufficient premisses,

$$\delta^{-1} : \mathcal{G}(\mathbb{A}) \to \mathcal{PLG}(\mathbb{A}) : (P, G) \mapsto \{\ell \in \mathcal{LG}(\mathbb{A}) \mid \delta(\ell) = (P, G)\}$$

Example 4.9. Let R be the $\to$-resolution rule in the modified sequent calculus (Lemma 2.11), then the deduction operator and its inverse are the following:

$$\begin{array}{lcll} \delta & : & [(P; G \to A \vdash G), (Q \vdash A)] & \mapsto & (Q, (P; G \to A)) \vdash A \\ \delta^{-1} & : & (Q, (P; G \to A)) \vdash A & \mapsto & [(P; G \to A \vdash G), (Q \vdash A)] \end{array}$$

It is worth comparing this presentation to the one given in Lemma 2.11; where δ represents the downward reading, δ^{-1} represents the upward one. ■

$$
\begin{array}{rlcl}
\rho_{\top}: & P \vdash \top & \mapsto & \{\Box\} \\
\rho_{\top^*}: & P \vdash \top^* & \mapsto & \{\Box \mid [P] \equiv \varnothing_{\times}; R\} \\
\rho_{res_1}: & P \vdash A & \mapsto & \{\{Q \vdash A\} \mid [P] \leqq Q, A\} \\
\rho_{res_2}: & P \vdash A & \mapsto & \{\{Q \vdash G\} \mid [P] \leqq Q, G \multimap A\} \\
\rho_{res_3}: & P \vdash A & \mapsto & \{\{Q \vdash \top^*, R; G \to A \vdash G\} \mid [P] \leqq Q, (R; G \to A)\} \\
\rho_{\vee}: & P \vdash G_1 \vee G_2 & \mapsto & \{\{P \vdash G_1\}, \{P \vdash G_2\}\} \\
\rho_{\wedge}: & P \vdash G_1 \vee G_2 & \mapsto & \{\{P \vdash G_1, P \vdash G_2\}\} \\
\rho_{*}: & P \vdash G_1 * G_2 & \mapsto & \{\{Q \vdash G_1, R \vdash G_2\} \mid P \equiv (Q, R); S\} \\
\rho_{\to}: & P \vdash D \to G_2 & \mapsto & \{\{P; [D] \vdash G\}\} \\
\rho_{\multimap}: & P \vdash D \multimap G_2 & \mapsto & \{\{P, [D] \vdash G\}\}
\end{array}
$$

Figure 5: Reduction Operators for the Resolution System

We make now an important assumption for proof-search, as presented herein, to be effective: the proof-search space must be finitely branching. Rules whose upward reading renders an infinite set of sufficient premisses (e.g., E) cannot be handled with step-by-step reduction since the breadth-first approach to search may fail. At first this assumption seems to be hugely restrictive, since almost all common proof-systems seemingly fail this criterion; however, it is not to be understood as injunction against such rules, merely a requirement to *control* them. For example, the E rule is confined in the resolution system (Figure 3) to the weak coherence ordering in Section 2.3, and an algorithmic approach to its use is offered. Hence, we drop the $\mathcal{P}$ structure in deduction operators and their inverses, and replace it with the finite powerset monad $\mathcal{P}_f$.

Moreover, since there is no logical constraint on the order in which sufficient premisses are verified, we may simplify the structure of the states by replacing the list structure $\mathcal{L}$ with the finite powerset monad $\mathcal{P}_f$ too; indeed this simplification is tacit in the labelled transition system as all the possible configurations in a possible state are explored simultaneously. When the construction in Example 4.9 is performed for the resolution system in Figure 3, with the simplification on the structure of states and collections of premisses, one forms the coalgebras in Figure 5, which are formally the *reduction operators* for the resolution system.

Definition 4.10 (Reduction Operator)**.** *A reduction operator for a logic L with sequent structure $\mathcal{G}(\mathbb{A})$ is a coalgebra $\rho : \mathcal{G}(\mathbb{A}) \to \mathcal{P}_f\mathcal{P}_f\mathcal{G}(\mathbb{A})$ such that $\rho(C) = \{P_0, ..., P_n\}$ if and only if the following inference is L-sound:*

$$\frac{P_0 \qquad \dots \qquad P_n}{C}$$

The would-be atomic steps of proof-search, a reductive inference, can be decomposed into two steps in the coalgebraic model of rules above: the application of a reduction operator, and the selection of a set of sufficient premisses,

$$\mathcal{G}(\mathbb{A}) \xrightarrow{\rho} \mathcal{P}_f\mathcal{P}_f\mathcal{G}(\mathbb{A}) \xrightarrow{\sigma} \mathcal{P}_f\mathcal{G}(\mathbb{A})$$

The choices presented by these steps represent the control problems of proof-search, as discussed in 2.3; that is, the choice of rule, and the choice of instance.

Simply by this presentation one has insight into the proof-search behaviour. For example, the disjointedness of the defined portions of the reduction operators for the operational and initial rules means the choice of application is deterministic for a non-atomic goal. Therefore, one may coalesce the operators into a single goal-destructor:

$$\rho_{\text{op}} := \rho_{\vee} + \rho_{\wedge} + \rho_{*} + \rho_{\to} + \rho_{\multimap} + \rho_{\top} + \rho_{\top^*}$$

Moreover, since there is no *a priori* way to know which resolution rule to use, one in principle tests all of them, so uses a reduction operator of the shape:

$$\rho_{\text{res}} : P \vdash A \mapsto \rho_{res_1}(P \vdash A) \cup \rho_{res_2}(P \vdash A) \cup \rho_{res_3}(P \vdash A)$$

The presentation of ρ_{op} and ρ_{res} as operators is simple when working with coalgebras, but it is not at all clear how to present them as a single rule figures. Attempts toward such presentations are the *synthetic* rules derived from focused

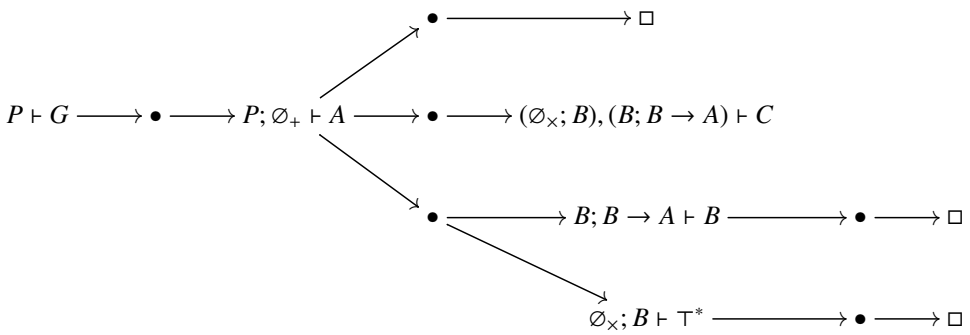
proof systems [18]. It is thus that the first control problem vanishes for hereditary Harrop fragment of BI (and in logic programming in general), since one may then say that *the* reduction operator for resolution proofs in the hereditary Harrop fragment of BI is $\rho := \rho_{\text{op}} + \rho_{\text{res}}$.

4.3 The Proof-search Space

The construction of reduction operators as above is an interpretation of a reduction step. We now turn to modelling reduction proper; that is, we construct a coalgebraic interpretation of the *proof-search space* — the structure explored during proof-search — itself. In practice, we simply formalize the heuristic exploratory approach of reductive inference as the corecursive application of reduction operators.

Computations, such as proof-search, can be understood as sequences of actions, and the possible traces can be collected into tree structures where different paths represent particular threads of execution. In logic programming, such trees appear in the literature as *coinductive derivation trees* (CD-trees) [46, 47, 48, 12], and an *action* is one step of reductive inference. Typically one distinguishes the two components, the reduction operator and the choice function, by using an intermediary node labelled with a •, sometimes called an *or*-node, as it represents the disjunction of sets of sufficient premisses.

Example 4.11. Let $P = ((\varnothing_\times; B), C \multimap A, (B; B \to A)); A$ and $G = \top \to A$, then the (P)CD-tree for $P \vdash G$ in the resolution system is the following:



At the first bifurcation point, the three • nodes represent from top to bottom the choice of the unique member of $\rho_{res_i}(P; \varnothing_\times \vdash A)$ for $i = 1, 2, 3$. In the case of ρ_{res_1} and ρ_{res_3}, the procedure continues and terminates successfully; meanwhile, for ρ_{res_2}, it fails. ■

The bullets • serve only as *punctuation* separating the possible choice functions, so the actual coinductive derivation tree is the tree without them. That is, the punctuation helps to define the proof-search space, but is not part of it.

Definition 4.12 ((Punctuated) Coinductive Derivation Tree)**.** *A punctuated coinductive derivation tree (PCD-tree) for a sequent S is a tree satisfying the following:*

- *The root of the tree is S*
- *The root has $|\rho(S)|$ children labelled •*
- *For each • there is a unique set $\{S_0, ..., S_n\} \in \rho(S)$ so that the the children are PCD-trees for the S_i.*

A saturated coinductive derivation tree for a sequent (CD-tree) is the tree constructed from the PCD-tree for the sequent constructed by connecting the parents of •-nodes directly to the children, removing the node itself.

The CD-trees model reduction only (as opposed to proof-search) since the representation of a control régime is lacking; nonetheless, we shall see in Section 4.4 that a model of proof-search is immediately available. A possible approach to at least represent the controls used in a particular search is to replace the bullets with integers that enumerate the order of preference in the possible choices.

The CD-structure on a set X of sequents is formally the cofree comonad $C(X)$ on the $\mathcal{P}_f\mathcal{P}_f$ functor, the behaviour-type of reduction, constructed inductively as follows:

$$\begin{cases} Y_0 & := X \\ Y_{\alpha+1} & := X \times \mathcal{P}_f\mathcal{P}_f Y_\alpha \end{cases}$$

Each stage of the construction yields a coalgebra $\rho_\alpha : X \to Y_\alpha$ defined inductively as follows, where I is the identify function:

$$\begin{cases} \rho_0 & := I \\ \rho_{\alpha+1} & := I \times (\mathcal{P}_f\mathcal{P}_f\rho_\alpha \circ \rho) \end{cases}$$

For some limit ordinal λ, the coalgebra $\rho_\lambda : \mathcal{G}(\mathbb{A}) \to C(\mathcal{G}(\mathbb{A}))$ precisely maps a configuration to its CD-tree. To show that this model of the proof-search space is faithful we must show that every step, represents a valid reduction; meanwhile, to show that it is adequate we must prove that every proof is present.

Definition 4.13 (Controlled Subtree of CD-tree)**.** *A subtree $\mathcal{R}$ of $\rho_\lambda(P, G)$ is controlled if and only if is a tree extracted from the PCD-tree for $P \vdash G$ by taking the root node and connecting it to the heads of the reduction trees of all the children of one $\bullet$-node. It is successful if and only if the leaves are all $\Box$.*

Observe that the application of a choice function, determined by a control régime, is precisely the choosing of a particular $\bullet$-node at each stage of the extraction.

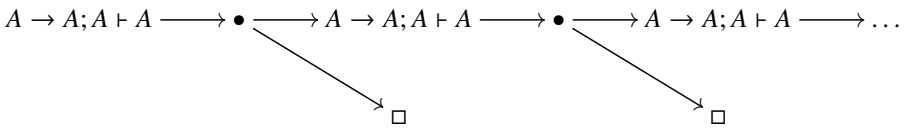
Example 4.14. The following is an example of a controlled sub-tree from the example (P)CD-tree above:

$$P \vdash G \longrightarrow P; \varnothing_\times \vdash A \longrightarrow \Box$$

The first choice of $\bullet$ is trivial (as there is only one) and the second choice is the upper path. ■

We do not claim that every controlled sub-tree in a CD-tree is finite; in fact, this is demonstrably not the case.

Example 4.15. Consider the PCD-tree for $A; A \to A \vdash A$:



Every finite execution of the configuration is successful; however, there is an infinite path which represents an attempt at proof-search that never terminates, but also never reaches a invalid configuration. This further demonstrates the care that is required when implementing controls because the *depth-first search with leftmost selection* régime here fails. ■

Theorem 4.16 (Soundness and Completeness)**.** *A tree labelled with sequents is a proof of a configuration $P \vdash G$ in the resolution system if and only if it is a successful controlled sub-tree of $\rho_\lambda(P, G)$.*

Proof. Immediate by induction on the height of proofs and the definition of reduction tree. □

4.4 Choice and Control

The difference between modelling the proof-search space and modelling proof-search is subtle, and comes down to the handling of the space of sufficient premisses. Throughout we have restricted attention to a traditional sequential paradigm for computation, where one chooses one particular collection of sufficient premisses, we now show that an alternative parallel model is immediately available from the constructions above. A notion of control is, nonetheless, ever present, and so we also show how the coalgebra offers a unifying perspective for a particular approach to it that is independent of the particular execution semantics.

It is the τ rule in the operational semantics of Section 2.3 that invokes the reduction operators. It expresses precisely the two-step understanding of reductive inference: apply a reduction operator, choose a collection of sufficient premisses. The usual way to interpret it is to perform these action in sequence as read, yielding the backtracking-schedule approach to computation studied so far; however, it may be interpreted simply *extracting* a correct reduction from the proof-search space. In this latter reading the traces of the proof-search space can be understood as being in a superposition, forming a parallel semantics of computation. This reading has been thoroughly studied for HcLP [31, 32, 30] and this idea of parallelism can be captured proof-theoretically by hypersequent calculi [33, 53].

The coalgebraic model of the proof-search space immediately offers a coalgebraic model of parallel proof-search; that is, the controlled sub-tree extraction from $\rho_\lambda : \mathcal{G}(\mathbb{A}) \to C(\mathcal{G}(\mathbb{A}))$. Indeed, this is precisely analogous to the parallel model of HcLP [46, 47, 48]. The coalgebraic approach has the advantage over more traditional algebraic models in that it allows for infinite searches, thereby extending the power of logic programming to include features such as corecursion [30, 82, 48].

Indeed, the parallel semantics is amenable to a more accurate model by unpacking the algebraic structure of the state-space, yielding a bialgebraic semantics. Observe then that in the structure $\mathcal{P}_f\mathcal{P}_f$ for reduction operators, the external functor structures the set of choices, and the internal one structures the states themselves (the collections of sufficient premisses). The outer one is disjunctive, captured diagrammatically by the *or*-nodes represented by $\bullet$ in the PCD-trees; meanwhile, the inner one is conjunctive since every premiss needs to be verified. There is no distributive law of $\mathcal{P}_f$ over $\mathcal{P}_f$, but there is a distributive law of $\mathcal{L}$ over $\mathcal{P}_f$ which coheres with this analysis:

$$[X_1, ..., X_N] \mapsto \{[x_1, ..., x_n] \mid x_i \in X_i\}$$

This suggests a bialgebraic model for the parallel reading of the operational semantics given in Section 2.3, obtained by performing the same cofree comonad construction for the behaviour, but with $\mathcal{P}_f\mathcal{L}$ instead.

These models are studied partly to let one reason about computation, and perhaps use knowledge to improve behaviour. For example, in the sequential semantics a programmer may purposefully tailor the program to the selection function to have better behaviour during execution, meanwhile in the parallel approach the burden is shifted to the machine (or, rather, theorem prover) which may give more time to branches that are more promising.

Example 4.17. While generating the (P)CD-tree in Example 4.11 a theorem prover can ignore the branch choosing the $C \multimap\!\!* A$ in the program since it is clear that it will never be able to justify C as the atom appears nowhere else in the context. ■

The problem being handled in either case is how best to explore the space of possible reductions. The two approaches, parallel and sequential, both suffer from the *amount* of non-determinism in the reduction operator ρ. In fact, this problem is exponentially increasing with each inference made as each collection of premisses represents another branch in the CD-tree. Moreover, in practice, with any additional features in the logic the problem compounds so that such reasoning becomes increasingly intractable.

As a case study, consider the plethora of research on *resource management* in linear logic programming [40, 38, 17, 39, 57] (which has sequential execution), to deal with the problem of context decomposition in operators such as ρ_*. These solution work for linear logic, but the *input-output* method, which underpins many of these mechanisms, forces a commitment to depth-first search, which is not always ideal. Moreover, the solution simply do not work hHBI because of the presence of another context-former.

One approach to improve the situation, given a particular reduction system, is to attempt to make the operators deterministic. Occasionally, it may be possible to quotient different paths or choices in the CD-tree by enriching the calculus with algebraic labels that can be used to differentiate the threads at a later point. That is, suppose we wish to perform a search in a system $\mathbf{L}'$, then we may simplify the problem by appealing to a more well-behaved $\mathbf{L}$,

$$\text{Proof-search in } \mathbf{L}' = \text{Reduction in } \mathbf{L} + \text{Controls } \mathcal{A}$$

For the resource management problem this is captured by the *resource distribution via boolean constraints* [35, 34] mechanism, which is uniformly applicable to LL and BI.

Modifying the rules of the sequent calculus so that sequents carry labels from a set $\mathcal{X}$ of boolean variables, one has inferences of the form:

$$\frac{\phi_0[e_0 f_0], ..., \phi_n[e_n f_n] \Rightarrow \psi_0[e] \qquad \phi_0[e_0 \bar{f}_0], ..., \phi_{G+1}[e_n \bar{f}_n] \Rightarrow \psi_1[e]}{\phi_0[e_0], ..., \phi_n[e_n] \implies \psi_0 * \psi_1[e]}\, e = 1$$

The possible choices for decomposing the context can be understood as assignments $A : \mathcal{X} \to \{0, 1\}$ subject to the constraints collected during proof-search, such as $A(e) = 1$. Let $\mathcal{B}$ denote boolean algebra, then the boolean-constaints approach to resource distribution instantiates the equation,

$$\mathbf{BI} = \mathbf{LK} + \mathcal{B}$$

Example 4.18. Consider the BI sequent $(A, (B; C), B \mathrel{-\!\!*} D); E \implies D * A$, and consider the following reduction tree with boolean labels:

$$\frac{\dfrac{\dfrac{x_0 = 0 \qquad x_1 = 1}{A[x_0], B[x_1] \Rightarrow B[x_3]}}{A[x_0], (B[x_1]; C[x_1]) \Rightarrow B[x_3]} \qquad \mathcal{D}}{(A[x_0], (B[x_1]; C[x_2]), (B \mathrel{-\!\!*} D)[x_3]); E[x_4] \Rightarrow D * A[x_5]}\, x_3, x_5 = 1, x_1 = x_2$$

where $\mathcal{D}$ is

$$\frac{\dfrac{\bar{x}_0 y_0 = 0 \qquad \bar{x}_1 y_1 \bar{x}_2 y_1 = 0 \qquad x_3 y_3 = 1}{A[\bar{x}_0 y_0], (B[\bar{x}_1 y_1]; C[\bar{x}_2 y_1]), D[x_3 y_3] \Rightarrow D[x_5]} \qquad \frac{\bar{x}_0 \bar{y}_0 = 1 \qquad \bar{x}_1 \bar{y}_1 \bar{x}_1 \bar{y}_1 = 0 \qquad x_3 \bar{y}_3 = 0}{A[\bar{x}_0 \bar{y}_0], (B[\bar{x}_1 \bar{y}_1]; C[\bar{x}_1 \bar{y}_1]), D[x_3 \bar{y}_3 = 0] \Rightarrow A[x_5]}}{(A[\bar{x}_0], (B[\bar{x}_1]; C[\bar{x}_2]), D[x_3]) \Rightarrow D * A[x_5]}}{(A[\bar{x}_0], (B[\bar{x}_1]; C[\bar{x}_2]), D[x_3]); E[x_4] \Rightarrow D * A[x_5]}\, x_5 = \bar{x}_0 + \bar{x}_1 + \bar{x}_2 + x_3$$

Each reductive inference replaces the distribution condition of the corresponding rule in **LBI** with a side-condition encoding it, thereby making it deterministic, without loss of expressivity. An assignment satisfying all the constraints is,

$$x_0 = 0, x_1 = 1, x_2 = 1, x_3 = 1, x_4 = 1, x_5 = 1, y_0 = 0, y_1 = 1, y_3 = 1$$

which instantiates to the following **LBI**-proof,

$$\frac{\dfrac{B \Rightarrow B}{(B; C) \Rightarrow B}\, W \qquad \dfrac{\dfrac{D \Rightarrow D \qquad A \Rightarrow A}{A, D \Rightarrow D * A}\, {*R}}{(A, D); E \Rightarrow D * A}\, W}{(A, (B; C), (B \mathrel{-\!\!*} D)); E \Rightarrow D * A}\, {\mathrel{-\!\!*} L}$$

Indeed, not only does the boolean constraints system have the same expressive power as **LBI**, but generally each reduction offers more information. For example, another assignment available for the above is the same but with $x_4 = 0$, meaning that we have at the same time found a proof of $A, (B; C), (B \mathrel{-\!\!*} D) \Rightarrow D * A$. ∎

This approach to control can be understood abstractly in the coalgebraic setting as a lifting of a proof-search step. Let $(\mathcal{G}.\mathcal{A})(\mathbb{A})$ be sequents labelled with variables from an $\mathcal{A}$-algebra together with $\mathcal{A}$-algebra equations, let ρ be a reduction operator for $\mathbf{L}'$, and let π be a deterministic reduction operator for $\mathbf{L}$, then the mechanism is captured by the following diagram:

$$\begin{array}{ccccc} (\mathcal{G}.\mathcal{A})(\mathbb{A}) & & \xrightarrow{\quad\pi\quad} & & \mathcal{L}(\mathcal{G}.\mathcal{A})(\mathbb{A}) \\ \uparrow{\scriptstyle \alpha} & & & & \downarrow{\scriptstyle \omega} \\ \mathcal{G}(\mathbb{A}) & \xrightarrow[\rho]{} & \mathcal{P}_f\mathcal{L}\mathcal{G}(\mathbb{A}) & \xrightarrow[\sigma]{} & \mathcal{L}\mathcal{G}(\mathbb{A}) \end{array}$$

The function α is the application of labels, and the function ω is a *valuation* determined by the choice function σ. Using the constructing of reduction operators from Section 2.3, with a formal meta-logic for the relations representing rules, one can give suitable conditions under which such a lifting is possible.

Moreover, though the best case is a fully deterministic π, this is not required since the top-path may still contain two facets and be an improvement as long as the reduction operator is, in general, more deterministic.

$$
\begin{array}{lcl}
P \vdash \exists x G & \Leftarrow & P \vdash G\theta \\
P \vdash A & \Leftarrow & P \equiv R; (Q, \forall x B) \text{ and } Q \vdash \top^* \text{ and } \exists \theta :\ B\theta = A\theta \\
P \vdash A & \Leftarrow & P \equiv R; (Q, \forall x (G \mathrel{-\!\!*} B)) \text{ and } \exists \theta :\ Q \vdash G\theta \text{ and } A\theta = B\theta \\
P \vdash A & \Leftarrow & \begin{cases} P \equiv R; (S, (Q; \forall x (G \to B))) \text{ and } S \vdash \top^* \text{ and} \\ \exists \theta :\ Q; \forall x (G \to B) \vdash G\theta \text{ and } A\theta = B\theta \end{cases}
\end{array}
$$

Figure 6: Unification Rules

5 Computation

5.1 Predicate BI

As a programming language hHBI should compute some *thing* which is determined by the goal, with constraints given by the program. Therefore, following the discussion in the introduction, we increase the expressivity of the language by extending the logic with predicates and quantifiers.

The standard approach for turning a propositional logic into predicate logic begins with the introduction of a set of terms $\mathbb{T}$, typically given by a context-free grammar which has three disjoint types of symbols: variables, constants, and functions. The propositional letters are then partitioned $\mathbb{A} := \bigcup_{i<\omega} \mathbb{A}_i$ into classes of predicates/relations of different arities, such that the set of atomic formulas is given by elements $A(t_0, ..., t_i)$, where $t_0, ..., t_i$ are terms and $A \in \mathbb{A}_i$. In the model theory, the Herbrand universe is the set of all ground terms $\overline{\mathbb{T}}$ (terms not containing free variables), and the Hebrand base is the set of all atomic formulas (instead of atomic propositions).

The extra expressivity of predicate logic comes from the pressence of two quantifiers: the universal quantifier $\forall$ and the existential quantifier $\exists$, which for the hereditary Harrop fragment gives the following grammar for formulas:

$$
\begin{array}{lcl}
D & ::= & ... \mid \forall x\, A \mid \forall x (G \to A) \mid \forall x (G \mathrel{-\!\!*} A) \\
G & ::= & ... \mid \exists x\, G
\end{array}
$$

Formally, programs and goals are not constructed out of arbitrary formulas, but only out of *sentences*: formulas containing no free-variables. However, since in this fragment the quantifiers are restricted to different types of formulas (and the sets of variables and constants are disjoint) they may be suppressed without ambiguity. For example, the formulas $A(x)$ regarded as a goal is unambiguously existentially quantified, whereas when regarded as a definite clause it is unambiguously universally quantified.

Rules for the quantifiers require the use of a mappings from $\theta : \mathbb{T} \to \overline{\mathbb{T}}$ that are fixed on $\overline{\mathbb{T}} \subseteq \mathbb{T}$, which are uniquely determined by their assignment of variables. Such a function becomes a *substitution* under the following action:

$$
\phi\theta := \begin{cases} A(\theta(t_0), ..., \theta(t_n)) & \text{if } \phi = A(t_0, ..., t_n) \\ \psi_0\theta \circ \psi_1\theta & \text{if } \phi := \psi_0 \circ \psi_1 \text{ for any } \circ \in \{\wedge, \vee, \to, *, \mathrel{-\!\!*}\} \\ \phi & \text{if } \phi \in \{\top, \top^*\} \end{cases}
$$

The resolution system (Figure 3) is thus extended with the operators in Figure 6 which incorporate the quantifier rules. Observe that substitution is used to match a definite clause with the goal, and for this reason is traditionally called a *unifier*. Since execution is about finding some term (some element of the Herbrand universe) which satisfies the goal, one may regard the thing being computed as the combined effect of the substitutions witnessed along the way, often called the *most general unifier*.

The presentation of the quantifier rules seems to have a redundancy since in each unification rule the goal A is supposed to already be ground, and thus by definition $A\theta = A$. However, in practice one may postpone commitment for existential substitution until one of the unification rules is used, yielding much better behaviour. This is already implicitly an example of the *control via algebraic constraints* phenomenon from Section 4.2.

The introduction of quantifiers into the logic programming language offered here is minimal, and much more development is possible [10]. An attempt at a full predicate BI is found in [70, 72], but its metatheory is not currently adequate. The intuition follows from the intimate relationship between implication and quantification in intuitionistic

Column 1	Column 2	Column 3
Al(gebra)	Lo(gic)	Da(tabases)
Pr(obability)	Ca(tegories)	Co(mpilers)
Gr(aphs)	Au(tomata)	AI

Figure 7: Columns of Electives

logics such as IL and MILL. The intended reading of an implication $A \to B$ in BI is a constructive claim of the existence of a procedure which turns a proof A into a proof of B. Therefore, a proof of an existential claim $\exists x A(x)$ involves generating (or showing how to generate) an object t for which one can prove $A(t)$; similarly, a proof of a universal claim $\forall x A(x)$ is an procedure which takes any object t and yields a proof of $A(t)$ [21]. The presence of both additive ($\to$) and multiplicative ($-\!\!*$) implications in BI results in the possibility of both additive (resp. $\{\exists, \forall\}$) and multiplicative (resp. $\{\Im, \not\mathscr{L}\}$) quantifiers. Intuitively, the difference between $\forall$ and $\not\mathscr{L}$ is that for $\not\mathscr{L} x\phi$ makes a claim about objects x *separate* from any other term appearing in ϕ, thus it may be read *for all new*; similarly for the relationship between the $\exists$ and $\Im$ quantifiers. This behaviour is similar to the freshness quantifier ($\mathcal{N}$) from nominal logic [68], which is the familiar universal quantifier ($\forall$) together with an exclusivity condition, and has a well understood metatheory.

5.2 Example: Databases

Abramsky and Väänänen [6] have shown that BI is the natural propositional logic carried by the Hodge's compositional semantics [41, 42] for the logics of informational dependence and independence by Hintikka, Sandu, and Väänänen [37, 36, 85], and in doing so introduced several connectives not previously studied in this setting. We develop here an example illustrating how BI's connectives indeed offer a useful, and natural, way to represent the combination of pieces of data.

Logic programming has had a profound effect on databases both theoretically, providing a logical foundation, and practically, by extending the power to incorporate reasoning capabilities [29]. Standard relational database systems are the fundamental information storage solution in data management, but have no reasoning abilities meaning information is either stored explicitly or is not stored at all. One may combine a database with a LP language resulting in a *deductive* database, which extends the capabilities of such relational databases to included features such as multiple file handling, concurrency, security, and inference.

A deductive database combines two components. The *extensional* part contains the atomic facts (ground atoms), and is the type of data that can exist in a relational database; meanwhile the *intensional* part contains inference rules and represents primitive reasoning abilities relative to a knowledgebase. In the case of hHLP, if there are no recursive rules in the intensional database then it corresponds to views in a relational database. However, even without recursion the two connectives of hHBI offers extra abilities as demonstrated in the following.

Suppose Unseen University offers a computer science course. To have a well-rounded education, students must select one module from each of three columns in Figure 7, with the additional constraint that to complete the course students must belong to a particular stream, A or B. Stream A contains $Al, Gr, Lo, Ca, Au, Co, AI$, and students must pick one from each column, stream B contains the complement. This compatibility information for modules may be stored as an extensional database provided by the bunch ED = (Col1,Col2,Col3) with each Col$i$ bunch defined as follows:

Col1 := A(Al) ; A(Gr) ; B(Pr) ; B(Gr)
Col2 := A(Lo) ; A(Ca) ; A(Au) ; B(Ca) ; B(Au)
Col3 := A(Co) ; A(AI) ; B(Da) ; B(Co) ; B(AI)

Let x be a list of subjects, then the logic determining CS courses for the $Astr$(eams) and $Bstr$(eams) respectively is captured by an intensional database ID given by the following bunch, where π_i is the ith projection function:

$$Astr(\pi_0(x), \pi_1(x), \pi_2(x)) \to str(x) \quad ; \quad Bstr(\pi_0(x), \pi_1(x), \pi_2(x)) \to str(x) \quad ;$$
$$A(x) * A(y) * A(z) -\!\!* Astr(x, y, z) \quad ; \quad B(x) * B(y) * B(z) -\!\!* Bstr(x, y, z)$$

The equivalent implementation in hHLP would require a tagging system to show compatibility of the columns; meanwhile the computation can be handled easily, and more importantly logically, in hHBI. Conversely, because hHBI is an extension of hHLP, any program in the latter is automatically executable in the former.

To find the possible combinations of subjects one performs the query $ED \;;\; ID \vdash str(x)$ — note that there are implicit quantifiers throughout. One possible execution is the following:

$$\dfrac{\dfrac{\dfrac{Col1 \vdash A(Al) \qquad Col2 \vdash A(Lo)}{Col1, Col2 \vdash A(Al) * A(Lo)} \qquad Col3 \vdash A(AI)}{Col1, Col2, Col3 \vdash A(Al) * A(Lo) * A(AI)}\,(1) \qquad \begin{array}{l}\theta := [\pi_0(x) \mapsto Al \\ \qquad\; \pi_1(x) \mapsto Lo \\ \qquad\; \pi_2(x) \mapsto AI]\end{array}}{\dfrac{ED; ID \vdash Astr([Al, Lo, AI])}{ED; ID \vdash str([Al, Lo, AI])}\,(2)}$$
$$\dfrac{}{ED; ID \vdash str(x)}$$

The example includes the use of two particular selection functions, on lines (1) and (2), which merit discussion. The first is the choice of decomposition of the context, for which there are six possibilities; the second is the choice of unifier θ, for which there are twenty-seven possible for each choice of definite clause. These two choices require control régimes. The problem at (1) is the context-management problem found in linear logic programming, and can be handled by encoding the *resource distribution via boolean constraints* mechanism [72]. Meanwhile, the problem at (2) is common to all logic programming languages: the choice of definite clause. Assuming a preferential selection and backtracking régime, what remains is the choice of unifier, which at this point in the execution is no better than *guessing*. Indeed, this problem is already found in the instantiation of x as the term $[Al, Lo, AI]$!

A student at Unseen University might reason slightly differently about their course, to avoid needless backtracking. For example, they may initially simply determine from the intensional database that to belong to the A stream they need to pick three modules, thus they need to find a list x with three components $y_0 = \pi_0(x)$, $y_1 = \pi_1(x)$, $y_2 = \pi_2(x)$. After this, they realize that the each y_i must belong to a seperate column, and look *simultaneously* across the possible choices, keeping track of the constraints as they accumulate. Such reasoning is presented by a reduction with algebraic side-conditions:

$$\dfrac{\dfrac{\dfrac{\vdots \qquad \vdots}{Col1, Col2, Col3 \vdash A(y_0) * A(y_1) * A(y_2)}\; y_0 = \pi_0(x), y_1 = \pi_1(x), y_2 = \pi_2(x)}{ED; ID \vdash Astr(y)}\; y = [\pi_0(x), \pi_1(x), \pi_2(x)]}{ED; ID \vdash str(x)}$$

This form of reasoning is precisely instantiating the control via algebraic constraints mechanism of Section 4. The hidden part of the computation contains the boolean constraints mechanism for resource distribution, and what is shown is the same kind of lifting used to control quantifier elimination instead. The study of hHBI shows that the meta-theory of control in proof-search is limited and fragmented, but that algebraic constraints offer not only a general approach, but a meaningful one supported by an established mathematical framework for computation.

6 Conclusion

We have developed a logic programming language HhBI based on goal-directed proof-search in a fragment of the logic of Bunched Implications. The analysis uses the now traditional approach of uniform proofs and selection functions. However, we begin from the general point of view of reductive logic which guides the entire construction; the object logic BI serves as an extended example where certain complication around control, a defining feature of proof-search, can be explored in more interesting detail. The resulting programming language is nonetheless interesting in itself, being particular suitable for programming systems that have constituent parts which partition the whole. Future work on the language itself includes implementation, and to extend it with further quantifiers as outlined in Section 5.1.

Though reductive reasoning is pervasive in both theory and practice of formal logic, it has no unified foundation. We have show that coalgebra is a robust framework for providing a uniform study via formally defined reduction operators; and, that such an approach coheres with contemporary mathematical approaches, such as bialgebra, to operational semantics. Developing the mathematical theory is a substantial ongoing area of research, for which the above may be regarded as a first step toward generalizing the perspective. Future work includes providing a systematic approach the meta-theory based on some general specification of an object logic, as opposed to developing the theory

on a case by case basis.

Proof-search follows from reductive logic by the application of a control regime. However, the abstract study of control is currently limited and fragmented lacking even a uniform language of discourse. Once more the coalgebraic framework appears to be effective since it resolves the problem into two facets (i.e., choice of rule, and choice of instance) each of which may be understood simply as the application of a coalgebra. This allows for an abstract perspective of traditional approaches; for example, for the choice-of-rule problem it show that the right rules can be condensed into a single reduction operator without additional non-determinism allowing uniform proofs to form a foundation for logic programming.

The second control problem, choice-of-instance, is more subtle. The incremental addition of features in the base logic for an logic programming language results in drastically more complicated handling of selection functions; consider, for example, the study of context management in linear logic programming as discussed in Section 4.4. An alternate approach from the use of a backtracking schedule is to make the reduction operators deterministic by lifting to a better behaved reduction system in which the possible choices are encoded as simple algebraic equations. The coalgebraic framework gives a clear account of this works. Moreover, in Section 5.2 we show that the algebraic constraints approach mimics the action of reasoning about a problem before executing it.

Future work includes developing the meta-theory of control, with an initial step being a precise mathematical development of this lifting. More generally, we would seek to develop a bialgebraic framework able to model generically — in the sense of a 'logical framework' — not only logical structure, but also control structure, in an integrated way.

Acknowledgements

We thank Pablo Armelín for the initiating the study on logic programming with the logic of Bunched Implications, and Fabio Zanasi and Sonia Marin for many valuable discussions concerning the algebraic and coalgebraic approaches we have described. This work has been partially supported by the UK's EPSRC through research grant EP/S013008/1.

References

[1] The Coq Proof Assistant. Internet. `https://coq.inria.fr/`.

[2] The Twelf Project. Internet. `http://twelf.org/wiki/Main_Page`.

[3] Samson Abramsky. Computational Interpretations of Linear Logic. *Theoretical Computer Science*, 111(1-2):3–57, 1993.

[4] Samson Abramsky. Information, Processes and Games. *Philosophy of Information*, 8:483–549, 2008.

[5] Samson Abramsky, Radha Jagadeesan, and Pasquale Malacaria. Full Abstraction for PCF. *Information and Computation*, 163(2):409–470, 2000.

[6] Samson Abramsky and Jouko Väänänen. From IF to BI. *Synthese*, 167(2):207–230, 2009.

[7] Jean-Marc Andreoli. Logic Programming with Focusing Proofs in Linear Logic. *Journal of logic and Computation*, 2(3):297–347, 1992.

[8] Krzysztof Apt and Maarten Emden. Contributions to the Theory of Logic Programming. *Journal of the ACM*, 29:841–862, 07 1982.

[9] Pablo Armelín and David J Pym. Bunched logic programming. In *International Joint Conference on Automated Reasoning*, pages 289–304, 2001.

[10] Pablo Armelín. *Programming with Bunched Implications*. PhD thesis, University College London, 2002.

[11] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development*. Springer-Verlag Berlin Heidelberg, 2004.

[12] Filippo Bonchi and Fabio Zanasi. Bialgebraic Semantics for Logic Programming. *Logical Methods in Computer Science*, 11(1), 2015.

[13] George Boolos. Don't Eliminate Cut! *Journal of Philosophical Logic*, 13(4):373–378, 1984.

[14] James Brotherston. Bunched Logics Displayed. *Studia Logica*, 100(6):1223–1254, 2012.

[15] James Brotherston and Jules Villard. Sub-classical Boolean Bunched Logics and the Meaning of Par. In *24th EACSL Annual Conference on Computer Science Logic*, pages 325–342, 2015.

[16] Alan Bundy. *The Computer Modelling of Mathematical Reasoning*. Academic Press Professional, Inc., 1985.

[17] Iliano Cervesato, Joshua S. Hodas, and Frank Pfenning. Efficient Resource Management for Linear Logic Proof Search. *Theoretical Computer Science*, 232(1-2):133–163, 2000.

[18] Kaustuv Chaudhuri, Sonia Marin, and Lutz Straßburger. Focused and synthetic nested sequents. In *Proceedings of 19th International Conference on Foundations of Software Science and Computation Structures*, pages 390–407, 2016.

[19] Simon Docherty. *Bunched Logics: A Uniform Approach*. PhD thesis, University College London, 2019.

[20] Simon Docherty and David Pym. A Stone-type Duality Theorem for Separation Logic via its Underlying Bunched Logics. *Electronic Notes in Theoretical Computer Science*, 336:101–118, 2018.

[21] Michael Dummett. *Elements of Intuitionism*. Logic Guides. Clarendon Press, 1977.

[22] Didier Galmiche, Michel Marti, and Daniel Méry. Relating Labelled and Label-Free Bunched Calculi in BI Logic. In *Proceedings of 28th International Conference on Automated Reasoning with Anamytic Tableaux and Related Methods*, pages 130–146, 2019.

[23] Didier Galmiche, Daniel Méry, and David Pym. The Semantics of BI and Resource Tableaux. *Mathematical Structures in Computer Science*, 15(6):1033, 2005.

[24] Gerhard Gentzen. *The Collected Papers of Gerhard Gentzen*. North-Holland Publishing Company, 1969.

[25] Alexander Gheorghiu and Sonia Marin. Focused Proof-search in the Logic of Bunched Implications. In *24th International Conference on Foundations of Software Science and Computation Structures*, 2021.

[26] Jean-Yves Girard. Linear Logic. *Theoretical Computer Science*, 50(1):1–101, 1987.

[27] Michael J. Gordon and Tom F. Melham. *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press, 1993.

[28] Michael J. Gordon, Christopher P. Wadsworth, and Robin Milner. *Edinburgh LCF. A mechanised logic of computation*. Springer-Verlag Berlin Heidelberg, 1979.

[29] John Grant and Jack Minker. The Impact of Logic Programming on Databases. *Communications of the ACM*, 35(3):66–81, 1992.

[30] Gopal Gupta, Ajay Bansal, Richard Min, Luke Simon, and Ajay Mallya. Coinductive Logic Programming and its Applications. In *Logic Programming*, volume 27, pages 27–44, 2007.

[31] Gopal Gupta and Vítor Santos Costa. Optimal Implementation of and-or Parallel Prolog. *Future Generation Computer Systems*, 10(1):71 – 92, 1994.

[32] Gopal Gupta, Enrico Pontelli, Khayri A.M. Ali, Mats Carlsson, and Manuel V. Hermenegildo. Parallel Execution of Prolog Programs: A Survey. *ACM Trans. Program. Lang. Syst.*, 23(4):472–602, July 2001.

[33] James Harland. An Algebraic Approach to Proof-search in Sequent Calculi. Presented at the International Joint Conference on Automated Reasoning, July 2001.

[34] James Harland and David J. Pym. Resource-distribution via Boolean Constraints. In *Automated Deduction—CADE-14*, pages 222–236, 1997.

[35] James Harland and David J. Pym. Resource-distribution via Boolean Constraints. *ACM Transactions on Computational Logic*, 4(1):56–90, 2003.

[36] Jaakko Hintikka. Hyperclassical logic (a.k.a. IF Logic) and its Implications for Logical Theory. *Bulletin of Symbolic Logic*, 8(3):404–423, 2002.

[37] Jaakko Hintikka and Gabriel Sandu. Informational Independence as a Semantical Phenomenon. In Jens Erik Fenstad, Ivan T. Frolov, and Risto Hilpinen, editors, *Logic, Methodology and Philosophy of Science VIII*, volume 126 of *Studies in Logic and the Foundations of Mathematics*, pages 571 – 589. Elsevier, 1989.

[38] Joshua S. Hodas. *Logic Programming in Intutionistic Linear Logic: Theory, Design and Implementation*. PhD thesis, University of Pennsylvania, 1994.

[39] Joshua S. Hodas, Pablo López, Jeffrey Polakow, Lubomira Stoilova, and Ernesto Pimentel. A Tag-frame System of Resource Management for Proof Search in Linear-logic Programming. In *International Workshop on Computer Science Logic*, pages 167–182, 2002.

[40] Joshua S. Hodas and Dale Miller. Logic Programming in a Fragment of Intuitionistic Linear Logic. *Information and Computation*, 110(2):327–365, 1994.

[41] Wilfrid Hodges. Compositional Semantics for a Language of Imperfect Information. *Logic Journal of the IGPL*, 5(4):539–563, 1997.

[42] Wilfrid Hodges. Some Strange Quantifiers. In *Structures in Logic and Computer Science: A Selection of Essays in Honor of A. Ehrenfeucht*, pages 51–65. Springer, Berlin, Heidelberg, 1997.

[43] Samin S. Ishtiaq and Peter W. O'Hearn. BI as an Assertion Language for Mutable Data Structures. *ACM SIGPLAN Notices*, 36(3):14–26, January 2001.

[44] Bart Jacobs. *Introduction to Coalgebra: Towards Mathematics of States and Observation*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2016.

[45] Bronisław Knaster and Alfred Tarksi. Un Théorème sur les Fonctions d'Ensembles. *Annales de la Societe Polonaise de Mathematique*, 6:133–134, 1928.

[46] Ekaterina Komendantskaya, Guy McCusker, and John Power. Coalgebraic Semantics for Parallel Derivation Strategies in Logic Programming. In *International Conference on Algebraic Methodology and Software Technology*, volume 13, pages 111–127, 2011.

[47] Ekaterina Komendantskaya and John Power. Coalgebraic Semantics for Derivations in Logic Programming. In *International Conference on Algebra and Coalgebra in Computer Science*, pages 268–282. Springer, 2011.

[48] Ekaterina Komendantskaya, John Power, and Martin Schmidt. Coalgebraic Logic Programming: from Semantics to Implementation. *Journal of Logic and Computation*, 26(2):745 – 783, April 2016.

[49] Robert Kowalski. Predicate Logic as Programming Language. In *IFIP Congress*, volume 74, pages 569–544, 1974.

[50] Robert Kowalski. Algorithm = Logic + Control. *Communications of the ACM*, 22(7):424–436, 1979.

[51] Robert Kowalski. *Logic for Problem-Solving*. North-Holland Publishing Co., 1986.

[52] Robert Kowalski and Donald Kuehner. Linear Resolution with Selection Function. *Artificial Intelligence*, 2:227–260, 12 1971.

[53] Hidenori Kurokawa. Hypersequent Calculi for Intuitionistic Logic with Classical Atoms. *Annals of Pure and Applied Logic*, 161(3):427–446, 2009.

[54] Joachim Lambek and Phillip Scott. *Introduction to Higher Order Categorical Logic*. Cambridge University Press, 1986.

[55] Dominique Larchey-Wendling. The Formal Strong Completeness of Partial Monoidal Boolean BI. *Journal of Logic and Computation*, 26(2):605–640, 2016.

[56] John W. Lloyd. *Foundations of Logic Programming*. Symbolic Computation. Springer-Verlag, 1984.

[57] Pablo López and Jeff Polakow. Implementing Efficient Resource Management for Linear Logic Programming. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, volume 11, pages 528–543, 2005.

[58] Dale Miller. A Logical Analysis of Modules in Logic Programming. *Journal of Logic Programming*, 6(1-2):79–108, 1989.

[59] Dale Miller and Gopalan Nadathur. *Programming with higher-order logic*. Cambridge University Press, 2012.

[60] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform Proofs as a Foundation for Logic Programming. *Annals of Pure and Applied Logic*, 51(1):125 – 157, 1991.

[61] Dale Miller and Vivek Nigam. Incorporating tables into proofs. In *International Workshop on Computer Science Logic*, pages 466–480, 2007.

[62] Dale Miller and Alexis Saurin. A Game Semantics for Proof Search: Preliminary Results. *Electronic Notes in Theoretical Computer Science*, 155:543–563, 2006.

[63] Robin Milner. The Use of Machines to Assist in Rigorous Proof. *Philosophical Transactions of the Royal Society of London. Series A, Mathematical and Physical Sciences*, 312(1522):411–422, 1984.

[64] Peter W. O'Hearn and David J. Pym. The Logic of Bunched Implications. *The Bulletin of Symbolic Logic*, 5(2):215–244, 1999.

[65] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*. Springer-Verlag Berlin Heidelberg, 1994.

[66] Lawrence C Paulson. *ML for the Working Programmer*. Cambridge University Press, 1996.

[67] Frank Pfenning and Carsten Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In *Automated Deduction — CADE-16*, pages 202–206, 1999.

[68] Andrew M. Pitts. Nominal Logic, a First Order Theory of Names and Binding. *Information and Computation*, 186(2):165–193, 2003.

[69] David A. Plaisted and Yunshan Zhu. *The Efficiency of Theorem Proving Strategies*. Springer, 1997.

[70] David J. Pym. Errata and Remarks for 'The Semantics and Proof Theory of the Logic of Bunched Implications'. `http://www.cantab.net/users/david.pym/BI-monograph-errata.pdf`.

[71] David J. Pym. *Proofs, Search and Computation in General Logic*. PhD thesis, University of Edinburgh, 1990.

[72] David J. Pym. On Bunched Predicate Logic. In *Symposium on Logic in Computer Science*, volume 14, pages 183–192, 1999.

[73] David J. Pym. *The Semantics and Proof Theory of the Logic of Bunched Implications*. Springer Netherlands, Dordrecht, 2002.

[74] David J. Pym. Reductive Logic and Proof-theoretic Semantics. In *Proceedings of the Third Tübingen Conference on Proof-Theoretic Semantics: Assessment and Future Perspectives*, 2019. `https://publikationen.uni-tuebingen.de/xmlui/handle/10900/93935`.

[75] David J. Pym. Resource Semantics: Logic as a Modelling Technology. *ACM SIGLOG News*, 6(2):5–41, April 2019.

[76] David J. Pym and James A Harland. A Uniform Proof-theoretic Investigation of Linear Logic Programming. *Journal of logic and Computation*, 4(2):175–207, 1994.

[77] David J. Pym, Peter W. O'Hearn, and Hongseok Yang. Possible Worlds and Resources: the Semantics of BI. *Theoretical Computer Science*, 315(1):257 – 305, 2004.

[78] David J. Pym and Eike Ritter. *Reductive Logic and Proof-search: Proof Theory, Semantics, and Control*. Oxford Logic Guides. Clarendon Press, 2004.

[79] Stephen Read. *Relevant Logic*. Basil Blackwell, 1988.

[80] Peter Schroeder-Heister. Proof-Theoretic versus Model-Theoretic Consequence. In *The Logica Yearbook 2007*, pages 187–200. College Publiations, 2008.

[81] Peter Schroeder-Heister. The Definitional View of Atomic Systems in Proof-Theoretic Semantics. In *The Logica Yearbook 2016*, pages 185–200. College Publiations, 2017.

[82] Luke Simon, Ajay Bansal, Ajay Mallya, and Gopal Gupta. Co-Logic Programming: Extending Logic Programming with Coinduction. In Lars Arge, Christian Cachin, Tomasz Jurdziński, and Andrzej Tarlecki, editors, *Automata, Languages and Programming*, pages 472–483, 2007.

[83] Alfred Tarski. A Lattice-theoretical Fixpoint Theorem and Its Applications. *Pacific Journal of Mathematics*, 5(2):285–309, 1955.

[84] Daniele Turi and Gordon Plotkin. Towards a Mathematical Operational Semantics. In *Proceedings of Twelfth Annual IEEE Symposium on Logic in Computer Science*, pages 280–291, 1997.

[85] Jouko Väänänen. *Dependence logic: A New Approach to Independence Friendly Logic*. Cambridge University Press, 2007.