Proof-search in Type-theoretic Languages: An Introduction

Didier Galmiche
LORIA UMR 7503
Université Henri Poincaré
Nancy, France
Didier.Galmiche@loria.fr

David J. Pym
Queen Mary & Westfield College
University of London
England, U.K.
David.Pym@dcs.qmw.ac.uk

Abstract

We introduce the main concepts and problems in the theory of proof-search in type-theoretic languages and survey some specific, connected topics. We do not claim to cover all of the theoretical and implementation issues in the study of proof-search in type-theoretic languages; rather, we present some key ideas and problems, starting from well-motivated points of departure such as a definition of a type-theoretic language or the relationship between languages and proof-objects. The strong connections between different proof-search methods in logics, type theories and logical frameworks, together with their impact on programming and implementation issues, are central in this context.

Keywords: logics, proof-search, type theory, semantics, logical frameworks, type-theoretic languages, proof-objects.

1 Introduction

Algorithmic proof-search is a fundamental enabling technology throughout computing science. There is a long history of work in proof-search in a variety of systems of logic, including classical, intuitionistic, relevant, linear and modal systems, at the propositional, first- and higher-order levels. Such work has ranged from the most abstract to the most practical and has employed the full spectrum of logical techniques, from proof theory, model theory and recursion theory.

Recently, there has been a great deal of work on proof-search in type-theoretic languages. Such languages can be thought of as logical frameworks to represent proofs and to formalize connections between proofs and programs. Here again, the scope of languages studied and techniques employed has been wide, stretching to include algebraic and categorical methods.

From the computational point of view, the type-theoretic component of logical languages, which may involve propositional, first-order, higher-order or polymorphic assignment regimes, introduces significant challenges for both theoreticians and implementors.

This introductory article is focussed on the following ideas:

- The notion of a type-theoretic language, in § 2, including the following topics:
  - Consequence relations;
  - Proof-theoretic approaches;
  - Model-theoretic approaches;
  - Logical Frameworks;
- The view, in § 3, of reasoning as *proof-search* or *reduction*, as opposed to *inference* or *deduction*, together with a discussion of the specifically type-theoretic issues that arise;
- The rôle of proof-search in programming, § 4. We consider the state-of-the-art in proof- and model-theoretic approaches to the semantics of logic programming with type-theoretic languages. We also consider the current state of applications of proof-search in functional programming. In particular, we consider the issues in program synthesis, program extraction, verification and transformation which are analysed and supported by theoretical and practical work in proof-search.

We conclude the article, in § 5, with a brief discussion of the need and prospects for more semantic approaches to the theory of proof-search.

2 Type-theoretic Languages

In this section, we aim to fix, starting from the notion of *consequence relation*, what a type-theoretic language is. For that, we give the proof-theoretic but also the model-theoretic views and then detail the main characteristics of type theories and logical frameworks, emphasizing the notion of *proof-object*.

2.1 Consequence Relations

Logic begins with language. Formally, we begin with a *formal language*, L, with several syntactic categories, typically including a category of *individuals* and, most importantly, a category of *well-formed formulae* or *propositions*. Given such a language, we can describe a *logic* $\mathcal{L}$ over L as a *consequence relation*, $\vdash_{\mathcal{L}}$, between finite sequences of propositions which satisfies the following, in which ϕ denotes an arbitrary proposition, the Γs and Δs denote arbitrary finite (possibly empty) sequences of propositions and "," denotes monoidal combination of sequences:

- *Reflexivity*: $\phi \vdash_{\mathcal{L}} \phi$;
- *Transitivity*: if $\Gamma_1 \vdash_{\mathcal{L}} \Delta_1, \phi$ and $\phi, \Gamma_2 \vdash_{\mathcal{L}} \Delta_2$, then $\Gamma_1, \Gamma_2 \vdash_{\mathcal{L}} \Delta_1, \Delta_2$.

A consequence relation may also satisfy the following additional properties:

- *Commutativity*: if $\Gamma \vdash_{\mathcal{L}} \Delta$, then $\Gamma' \vdash_{\mathcal{L}} \Delta'$, where Γ' and Δ' are permutations of Γ and Δ, respectively;
- *Weakening*: if $\Gamma_1 \vdash_{\mathcal{L}} \Delta_1$, then $\Gamma_1, \Gamma_2 \vdash_{\mathcal{L}} \Delta_1, \Delta_2$;
- *Contraction*: if $\Gamma, \Gamma_1, \Gamma_1 \vdash_{\mathcal{L}} \Delta$, then $\Gamma, \Gamma_1 \vdash_{\mathcal{L}} \Delta$; if $\Gamma \vdash_{\mathcal{L}} \Delta, \Delta_1, \Delta_1$, then $\Gamma \vdash_{\mathcal{L}} \Delta, \Delta_1$.

In the absence of commutativity other, distinct, formulations of weakening and contraction are possible.[1]

Example 2.1 *Consider a propositional language $\mathcal{L}$ with a constant $\top$ and two binary operators $\circ$ and $\rightarrow$. A consequence relation $\vdash_{\mathcal{L}}$ can be generated by a calculus of sequents including the rules*

$$\frac{}{\phi \vdash \phi}\, Id \qquad \frac{\Delta_1, \phi, \Delta_2 \vdash_{\mathcal{L}} \psi \quad \Gamma \vdash_{\mathcal{L}} \phi}{\Delta_1, \Gamma, \Delta_2 \vdash_{\mathcal{L}} \psi}\, Cut,$$

together with the additional rules

$$\frac{\Gamma, \phi, \psi, \Delta \vdash_{\mathcal{L}} \chi}{\Gamma, \psi, \phi, \Delta \vdash_{\mathcal{L}} \chi}\, S_1 \qquad \frac{\Gamma, \phi, \Delta \vdash_{\mathcal{L}} \chi}{\Gamma, \phi, \phi, \Delta \vdash_{\mathcal{L}} \chi}\, S_2$$

$$\frac{\Gamma, \phi, \phi, \Delta \vdash_{\mathcal{L}} \chi}{\Gamma, \phi, \Delta \vdash_{\mathcal{L}} \chi}\, S_3 \qquad \frac{\Gamma \vdash_{\mathcal{L}} \chi}{\Gamma, \phi \vdash_{\mathcal{L}} \chi}\, S_4$$

We refer to this collection of conditions as structural rules*: such rules are so called because they are defined in terms of the structure of sequents and do not refer to the propositional operators. Let the symbol $\circ$ denote any binary operator which satisfies the conditions that $\Gamma, \phi, \psi, \Delta \vdash \chi$ if and only if $\Gamma, \phi \circ \psi, \Delta \vdash_{\mathcal{L}} \chi$ and $\Gamma, \Delta \vdash_{\mathcal{L}} \phi \circ \psi$ if and only if $\Gamma \vdash_{\mathcal{L}} \phi$ and $\Delta \vdash_{\mathcal{L}} \psi$. Similarly, let the symbol $\rightarrow$ denote any binary operator which satisfies the conditions that $\Gamma, \phi \vdash \psi$ if and only if $\Gamma \vdash_{\mathcal{L}} \phi \rightarrow \psi$ and $\Gamma, \phi \rightarrow \psi, \Delta \vdash_{\mathcal{L}} \chi$ if and only if $\Gamma \vdash_{\mathcal{L}} \phi$ and $\Delta, \psi \vdash_{\mathcal{L}} \chi$. Let the symbol $\top$ represent the empty sequence in the left-hand side of a sequent so that $\Gamma, \Delta \vdash_{\mathcal{L}} \phi$ if and only if $\Gamma, \top, \Delta \vdash_{\mathcal{L}} \phi$. Then the smallest consequence relation containing the operators $\top$, $\circ$ and $\rightarrow$ and closed under the structural rules corresponds to the $\{\top, \circ, \rightarrow\}$-fragment of intuitionistic propositional logic.*

Given a definition of a logic $\mathcal{L}$ as a consequence relation $\vdash_{\mathcal{L}}$, we can try to characterize $\mathcal{L}$ either *model-theoretically* or *proof-theoretically*.

(1) Model-theoretically, we must first establish a class of models and a notion of satisfaction which together carry the structure of the logical connectives in $\mathcal{L}$. If $\mathcal{M}$ is such a model and $\models$ is such a satisfaction relation, we write

$$\mathcal{M} \models_{\mathcal{L}} \phi$$

[1] Strictly speaking, what we have described so far are *simple* consequence relations. In [9], there is the additional requirement of *uniformity*. Uniformity concerns the internal structure of propositions and requires "closure under substitution". The formal definition of this idea requires some technical care and so, for brevity, it is elided here. It is not necessary for our conceptual development.

to denote that the proposition ϕ holds in $\mathcal{M}$ according to $\models$. Often, such a definition depends on the internal structure of $\mathcal{M}$, such as that provided by the *worlds* of Kripke models, so that we write

$$\mathcal{M}\,,\, m \models_{\mathcal{L}} \phi$$

to denote that the proposition ϕ holds *at world* m in $\mathcal{M}$ according to $\models$.

A satisfaction relation can be generated inductively, according to the structure of the propositions in $\mathcal{L}$. For example, the following clauses can be used to generate the satisfaction relation for the $\{\top, \circ, \to\}$-fragment of intuitionistic logic over a preorder $\mathcal{W} = (W, \sqsubseteq)$:

- $\mathcal{W}\,,\, w \models_{\mathcal{L}} \top$ for all $w \in W$;
- $\mathcal{W}\,,\, w \models_{\mathcal{L}} \phi \circ \psi$ iff $\mathcal{W}\,,\, w \models_{\mathcal{L}} \phi$ and $\mathcal{W}\,,\, w \models_{\mathcal{L}} \psi$;
- $\mathcal{W}\,,\, w \models_{\mathcal{L}} \phi \to \psi$ iff, for every $w' \in W$ such that $w \sqsubseteq w'$, $\mathcal{W}\,,\, w' \models_{\mathcal{L}} \phi$ implies $\mathcal{W}\,,\, w' \models_{\mathcal{L}} \psi$.

This satisfaction relation is sound and complete with respect to the consequence relation described in Example 2.1.

(2) Proof-theoretically, we fix a system **S** of axioms and rules which allow us to derive judgements of provability of the form

$$\mathbf{S} \quad \text{proves} \quad \Gamma \vdash_{\mathcal{L}} \phi.$$

Just as satisfaction relations can be defined by induction on the structure of propositions, so judgements of provability can be defined by induction on the structure of propositions, provided structural rules are also taken. Example 2.1 shows us how to do this, using a sequent calculus [58], for the $\{\top, \circ, \to\}$-fragment of intuitionistic propositional logic.

2.2 The Proof-theoretic View of Type-theoretic Languages

The definition of a consequence relation that we have considered so far can be analysed at the level of Tarski's semantics [165]: it is truth-conditional in the sense that we can only know that a consequence $\Gamma \vdash_{\mathcal{L}} \Delta$ holds, not *why* it holds. However, we can ask for a representation of the *evidence* for a consequence. We write

$$\Gamma \vdash_{\mathcal{L}}^{\Phi(\Gamma,\phi)} \phi$$

to denote that $\Phi(\Gamma, \phi)$ is a proof-object which represents a derivation of ϕ from Γ.

We assume that propositions ϕ and proof-objects Φ are given by independent, inductively defined grammars over disjoint signatures Σ_ϕ and Σ_Φ. Informally, we say that our representation of proof-objects is *type-theoretic* if there is a correspondence between the structure of the propositions and the structure of proof-objects as follows:

- Let propositional consequences

$$\Gamma \vdash_{\mathcal{L}} \phi$$

be generated by a system $\mathbf{R}_\phi$ of rules and let annotated consequences

$$\Gamma \vdash_{\mathcal{L}}^{\Phi(\Gamma,\phi)} \phi$$

be generated by a system $\mathbf{R}_\Phi$ of rules. Let r be a rule and let X be a combinator (in either Σ_ϕ or Σ_Φ). We say that r *manipulates* X if X's context in the premisses of r is different from its context in the conclusion of r;

- There is a bijection f between the combinators $c \in \Sigma_\phi$ in the grammar of proof-objects and combinators, or connectives, $C \in \Sigma_\Phi$ in the grammar of propositions. Moreover, there is a bijection

$$\mathbf{R}_\Phi \xrightarrow{g} \mathbf{R}_\phi$$

such that if $r \in \mathbf{R}_{\boldsymbol{\Phi}}$ manipulates c and $f(c)$, then $g(r) \in \mathbf{R}_\phi$ manipulates $f(c)$ and if $r \in \mathbf{R}_\phi$ manipulates C, then $g^{-1}(r)$ manipulates C and $f^{-1}(C)$;

- Every pair c and $f(c)$ is manipulated by some $r \in \mathbf{R}_{\boldsymbol{\Phi}}$ and every pair C is manipulated by some $r \in \mathbf{R}_\phi$.

Example 2.2 *Let $\mathcal{L}$ be minimal propositional logic and let $\mathbf{R}_\phi$ be the usual natural deduction rules [139]. Then the representation of proofs as typed λ-terms, with $\mathbf{R}_\Phi$ given by the usual rules of the typed λ-calculus, according to the propositions-as-types correspondence [78] is type-theoretic.*

Example 2.3 *Let $\mathcal{L}$ be minimal propositional logic and let $\mathbf{R}_\phi$ be the usual natural deduction rules [139]. Then the representation of proofs as texts generated by the usual Hilbert-type system,* i.e., *axioms together with modus ponens, is* not *type-theoretic.*

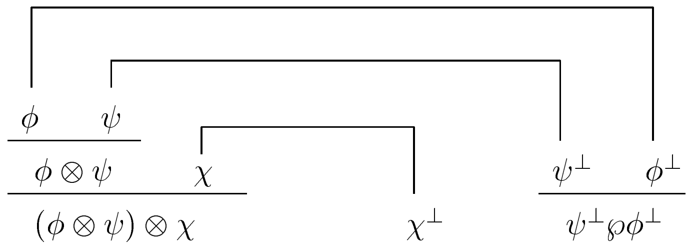
Examples 2.2 and 2.3 suggest that we can think of the relationship between proof-objects and propositions as being generated *locally* in type-theoretic representations but *globally* in non-type-theoretic representations.

Gentzen's natural deduction is a suitable deduction system for intuitionistic logic (IL) but appears less so for classical logic (CL) for which the Gentzen's sequent calculus seems better suited. In fact, the propositions-as-types correspondence allows us to naturally annotate natural deductions with terms. For IL, the language of terms is the typed λ-calculus whereas for the sequent calculus it is not clear what are the appropriate annotations and this lack leads to revisit natural deduction for CL. Although a recent proposal, which consists of a variant of multiple-conclusioned natural deduction with the so-called $\lambda\mu$-calculus as the language of annotating terms [124], provides a term language for classical logic, it does so by introducing to the classical sequent calculus some of the asymmetries inherent in intuitionistic logic. The problem of naturally representing within proof-objects the global symmetries inherent in classical logic whilst retaining type-theoretic locality has been addressed, with limited success, by Girard [61, 62] in his theory of *proof nets* for classical linear logic [63].

Example 2.4 *Classical multiplicative linear logic, with connectives $\otimes$, $\wp$ and the involutive negation $-^{\perp}$, can be described as a one-sided sequent calculus with judgements $\vdash \Gamma$. We can annotate such consequences with* proof nets *and obtain a type-theoretic presentation of linear logic. For example, in the annotated sequent*

$$\vdash^{\Phi} (\phi \otimes \psi) \otimes \chi, \phi^{\perp} \wp \psi^{\perp}, \chi^{\perp},$$

Φ is (up to some Exchanges) the proof net



Whilst proof nets provide a global *structure for proofs, they are generated locally, by the $\otimes$, $\wp$ and Axiom (and Cut) rules. It follows that our definition of type-theoretic is satisfied.*

The idea of annotating sequents of CL with a formulation of natural deduction based on the $\lambda\mu$-calculus has been adapted to classical linear logic (CLL) [18], thereby providing a type-theoretic presentation of classical linear logic based on the λ-calculus. Such a presentation retains type-theoretic locality at the price of the loss of some global symmetry.

Finally, we give an example which goes a little beyond the framework for type-theoretic languages that we have introduced.

Example 2.5 (bunched logic) BI, *the logic of bunched implications [115, 116, 147], uses sequents with antecedents structured not as lists but as* bunches. *Bunches have two combining operations, ";" and ",". Different structural properties are specified for ";", which admits weakening and contraction, and ",", which admits neither. This richer sequential structure allows additive and multiplicative implications to live side-by-side, without recourse to linear logic's exponentials. Propositional* **BI***'s proof-objects are characterized by the $\alpha\lambda$-calculus. Predicate* **BI***'s proof-objects require a dependently-typed λ-calculus [86].*

These examples illustrate that for a given logic, it is not always evident how to define proof representations that are type-theoretic. In this setting, the proposals of new logics, to deal with new problems and applications, is strongly connected to the design of new calculi to express proof-objects.

To include logics such as **BI** within our framework we should have to extend our treatment of consequence relations to account for multiple combining operations.

2.3 The Model-theoretic View of Type-theoretic Languages

The Tarski-style semantics we have discussed so far can be generalized to encompass algebraic and denotational models. The key idea is to *interpret* propositions, and hence contexts, in a semantic structure, $\mathcal{M}$, such as a set or a category which carries some specified structure. Given interpretations $[\![\Gamma]\!]_{\mathcal{M}}$ of a context Γ and $[\![\phi]\!]_{\mathcal{M}}$ of a proposition ϕ, we can ask whether there exists a map f in $\mathcal{M}$ such that

$$f : [\![\Gamma]\!]_{\mathcal{M}} \mapsto [\![\phi]\!]_{\mathcal{M}}.$$

If such a map exists, then we know that $\mathcal{M} \models \phi[\Gamma]$. However, we can extend the interpretation in a semantic structure to terms and ask if there exists an f such that $f = [\![\Phi]\!]_{\mathcal{M}}$, *i.e.*, if $\mathcal{M} \models (\Phi : \phi)[\Gamma]$. Such a model is said to be *type-theoretic*.

Example 2.6 (intuitionistic logic) *Let $\mathcal{L}$ be the $(\wedge, \supset)$-fragment of intuitionistic propositional logic, with the empty context denoted by $\langle\rangle$ and with proof-objects represented as terms of the simply typed λ-calculus. Let $\mathcal{M}$ be a cartesian closed category (CCC). Then the following interpretation determines a type-theoretic model: first, the propositions,*

$$\begin{array}{rcl} [\![\langle\rangle]\!]_{\mathcal{M}} & = & 1 \\ [\![\phi \wedge \psi]\!]_{\mathcal{M}} & = & [\![\phi]\!]_{\mathcal{M}} \times [\![\psi]\!]_{\mathcal{M}} \\ [\![\phi \supset \psi]\!]_{\mathcal{M}} & = & [\![\psi]\!]_{\mathcal{M}}{}^{[\![\phi]\!]_{\mathcal{M}}} \end{array}$$

and second, in a simplified form, the proofs,

$$\begin{array}{rcl} [\![\Gamma, x : \phi \vdash x : \phi]\!]_{\mathcal{M}} & = & [\![\Gamma, x : \phi]\!]_{\mathcal{M}} \xrightarrow{\pi_x} [\![x : \phi]\!]_{\mathcal{M}} \\ [\![\Gamma \vdash \Phi \wedge \Psi : \phi \wedge \psi]\!]_{\mathcal{M}} & = & [\![\Gamma]\!]_{\mathcal{M}} \xrightarrow{[\![\Phi]\!]_{\mathcal{M}} \times [\![\Psi]\!]_{\mathcal{M}}} [\![\phi]\!]_{\mathcal{M}} \times [\![\psi]\!]_{\mathcal{M}} \\ [\![\Gamma \vdash \lambda x : \phi . \Psi : \phi \supset \psi]\!]_{\mathcal{M}} & = & f \in [[\![\Gamma]\!]_{\mathcal{M}}, [\![\phi \supset \psi]\!]_{\mathcal{M}}] \\ & \vdots & \end{array}$$

where f is a unique functional element which commutes with application (for which the clause is omitted). Here $\models$ defined by $\mathcal{M} \models (\Phi : \phi)[\Gamma]$ iff there is an f in $\mathcal{M}$ such that $[\![\Phi]\!]_{\mathcal{M}} \simeq f : [\![\Gamma]\!]_{\mathcal{M}} \longrightarrow [\![\phi]\!]_{\mathcal{M}}$ (where $\simeq$ is Kleene equality).

Example 2.7 (linear logic) *Let $\mathcal{L}$ be the $(\otimes, \multimap)$-fragment of linear propositional logic, with the empty context denoted by $\langle\rangle$ and with proof-objects represented as terms of the simply typed linear λ-calculus [16]. Let $\mathcal{M}$ be a symmetric monoidal closed category (SMCC). Then the following interpretation determines a type-theoretic model:*

$$\begin{array}{rcl} [\![\langle\rangle]\!]_{\mathcal{M}} & = & I \\ [\![\phi \otimes \psi]\!]_{\mathcal{M}} & = & [\![\phi]\!]_{\mathcal{M}} \otimes [\![\psi]\!]_{\mathcal{M}} \\ [\![\phi \multimap \psi]\!]_{\mathcal{M}} & = & [\![\psi]\!]_{\mathcal{M}}{}^{[\![\phi]\!]_{\mathcal{M}}} \end{array}$$

with $\models$ defined by $\mathcal{M} \models (\Phi : \phi)[\Gamma]$ iff there is an f in $\mathcal{M}$ such that (where $\simeq$ is Kleene equality) $[\![\Phi]\!]_{\mathcal{M}} \simeq f : [\![\Gamma]\!]_{\mathcal{M}} \longrightarrow [\![\phi]\!]_{\mathcal{M}}$.

Classical logic naturally provides a non-example.

Example 2.8 (classical logic) *Classical propositional consequences can be interpreted in Boolean algebras but such a semantics is not type-theoretic: there is no way to distinguish between different proofs of the same sequent in such a semantics.*

Again, we conclude this point with bunched logic.

Example 2.9 (bunched logic) *Models of propositional* **BI***, including its proof-objects, [116, 149, 147] are characterized by* bicartesian doubly closed categories, i.e., *categories which enjoy two (symmetric) monoidal closed structures, one of which is cartesian, and which also have co-products. A host of examples, relying on Day's tensor product construction [38], is provided by categories of presheaves over a (symmetric) monoidal category $\mathcal{C}$,* i.e., $\mathbf{Set}^{\mathcal{C}^{op}}$. *Of particular interest is the case in which $\mathcal{C}$ is a preordered monoid, in which we obtain a Kripke-style forcing semantics. Such a semantics can be extended to predicate* **BI** *[116, 147, 150].*

In fact, these examples illustrate that for a given logic, in addition to the need of appropriate proof representations, the search and definition of a type-theoretic model is not a trivial work.

The semantic view we have sketched in this section is clearly motivated by categorical model theory. Our view essentially subsumes algebraic and truth conditional formulations of semantics.

2.4 The Propositions-as-types Correspondence

So far we have considered what it is for presentation of a logical system to be type-theoretic. However, a system of types has no *a priori* relationship with logic. Barendregt [14] gives an extensive explanation, in both the Curry and Church styles, of the point of view that types are syntactic entities which can be assigned to λ-terms [15] in order to restrict function application.

We say that there is a *propositions-as-types* correspondence between a proof system $\mathbf{S}$ for logic $\mathcal{L}$ and a type theory $\mathcal{T}$ just in case the following:

- *Transition:* Each proposition ϕ of $\mathcal{L}$ can be interpreted as a type $[\phi]$ of $\mathcal{T}$ (the transition $\phi \mapsto [\phi]$ is often called the propositions-as-types correspondence);
- *Soundness:* The transition $\phi \mapsto [\phi]$ induces a transition $\Gamma \mapsto [\Gamma]$ between contexts in $\mathcal{L}$ and contexts in $\mathcal{T}$ and a transition $\Phi \mapsto [\Phi]$ between proof-objects in $\mathcal{L}$ and terms in $\mathcal{T}$ such that if Φ is a proof of $\Gamma \vdash_{\mathcal{L}} \phi$ in $\mathbf{S}$, then $[\Gamma] \vdash_{\mathcal{T}} [\Phi] : [\phi]$ is provable in $\mathcal{T}$.

In addition, we may also have the following:

- *Completeness:* If there is a term M in $\mathcal{T}$ such that $[\Gamma] \vdash_{\mathcal{T}} M : [\phi]$ is provable in $\mathcal{T}$, then $\Gamma \vdash_{\mathcal{L}} \phi$ is provable in $\mathbf{S}$.

Originally formulated, building on ideas of Curry, by de Bruijn and Howard [78] for minimal propositional and predicate logic, the correspondence has been extended to more complex, but essentially intuitionistic, systems by several authors; see, for example, the references in [14]. More recently, the correspondence has been extended to classical propositional and predicate logic by Parigot [123, 124, 101], to propositional intuitionistic linear logic (see [166]) and to a bunched logic, combining linear and intuitionistic predicate logics [116, 147], by Ishtiaq and Pym [86].

A good view of the propositions-as-types correspondence for minimal/intuitionistic logic is given by the λ-cube [14], in which are represented eight λ-calculi (*à la Church*), λ_i, covering the possible dependencies between terms and types (terms depending on terms or types and types depending on terms or types). For instance, in $\lambda_{\rightarrow}$ (simple typed λ-calculus), one has terms depending on terms, in λP (type dependent calculus) one has types depending on terms, in $\lambda 2$ (second order λ-calculus, system F [60]) one has terms depending on types, in $\lambda\omega$ one has types depending on types. The other calculi are combination of the previous features, for instance the Calculus of Constructions (CC) [35] includes all these sort of dependencies. Therefore we can consider this calculus from a first extension of second-order (or polymorphic) λ-calculus in order to allow the binding of propositions (or types) schemas. This permits the definition of connectives within this formalism. A second extension consists in adding a first-order part with quantification and abstraction on elements. With the correspondence, this gives first-order logic and higher-order as well, since implication plays the rôle of a functional type. It stands in propositions-as-types correspondence to an $\mathcal{L}$-cube of eight logics $\mathcal{L}_i$, the correspondence being formulated uniformly for the eight systems. For instance, higher-order predicate logic corresponds to CC (for which completeness fails). Such a correspondence with logics well explains why some works on proof-search in pure logics are essential in the setting of proof-theoretic languages. In fact, first- and second-order quantification and dependent types are naturally motivated by the aim to be able to specify the terms-types dependencies and thus to have enough power for logical specification of computations and proof systems as well as more general predication, in an uniform way [82].

Important work has been devoted to the use of this propositions-as-types correspondence in programming, with the general idea being to use some results to extract programs from intuitionistic proofs [34]. This point of view has been generalized by Martin-Löf who used the correspondence to develop a system that is at the same time a programming language and a system dedicated to the development of intuitionistic mathematics [102, 103, 104]. In comparison, the Calculus of Constructions is very expressive but it is not always evident that one has a semantics for the program constructed with the functional calculus. In fact, the choice of the system depends on the power and usefulness of both logics and corresponding languages. The question of making precise what would be the analogue in terms of proofs (or λ-terms) of mechanisms used in programming (*e.g.*, general recursion) is important in this context. Moreover, one has to clarify the question of knowing whether the complexity of mathematical proofs has some counterpart in programming.

It has been possible to design general or specific proof-search methods for such expressive logics and frameworks, having in mind these strong connections between the type systems and the logics (see for instance [7, 40, 73, 143, 151]). We aim from now to illustrate the main points about proof-search in type-theoretic languages through different logics or languages, like for instance the LF logical framework [72], based on the in λP-calculus, but the main concepts we present have to be defined or analysed (generally in non-trivial way) in other so-called logical frameworks.

2.5 Logical Frameworks

Type-theoretic languages are often described as "logical frameworks" but one has to be careful about such claims. Following [72], we intend a logical framework to be a meta-logic within which object-logics are represented. It follows that, in order to describe a framework, one must provide the following:

1. A characterization of the class of (object-)logics to be represented. In the LF logical framework [11, 72, 151], we are concerned with Hilbert-type and natural deduction systems (see [9] for definitions) which admit weakening and contraction. In RLF [86], we are concerned just with natural deduction systems which do not necessarily admit weakening and contraction. In the Linear Logical Framework (LLF) [29, 31], the intended class of object-logics has to be stated;

2. An appropriate meta-language. In LF, the meta-language is the $\lambda\Pi$-calculus, a system of first-order dependent function types in propositions-as-types correspondence with first-order minimal logic.

   The $\lambda\Pi$-calculus is a formal system for deriving assertions of one of the following shapes:

$$
\begin{array}{ll}
\vdash \Sigma \ \ \text{sig} & \Sigma \text{ is a signature} \\
\vdash_\Sigma \Gamma \text{ context} & \Gamma \text{ is a context} \\
\Gamma \vdash_\Sigma K \ \ \text{kind} & K \text{ is a kind} \\
\Gamma \vdash_\Sigma A : K & A \text{ has kind } K \\
\Gamma \vdash_\Sigma M : A & M \text{ has type } A
\end{array}
$$

   where the syntax is specified by the following grammar:

$$
\begin{array}{llll}
\textit{Signatures} & \Sigma & ::= & \langle\rangle \mid \Sigma, c : K \mid \Sigma, c : A \\
\textit{Contexts} & \Gamma & ::= & \langle\rangle \mid \Gamma, x : A \\
\textit{Kinds} & K & ::= & \text{Type} \mid \Pi x : A.K \\
\textit{Types} & A & ::= & c \mid \Pi x : A.B \mid \lambda x : A.B \mid AM \\
\textit{Objects} & M & ::= & c \mid x \mid \lambda x : A.M \mid MN
\end{array}
$$

   The Π-type specializes to $\to$ in the absence of type-dependency.

   Full details of the calculus and its metatheoretic properties can be found in [11, 72, 140, 151]. The key points to note, in comparison with the simply-typed λ-calculus, are that contexts are ordered from left to right according to dependency and that abstraction and application, for the top-level judgement, are given by

$$
\frac{\Gamma, x : A \vdash_\Sigma M : B}{\Gamma \vdash_\Sigma \lambda x : A.M : \Pi x : A.B}\ \Pi\text{i} \quad \text{and} \quad \frac{\Gamma \vdash_\Sigma M : \Pi x : A.B \quad \Gamma \vdash_\Sigma N : A}{\Gamma \vdash_\Sigma MN : B[N/x]}\ \Pi\text{e}
$$

   The system is strongly normalizing, Church-Rosser and predicative; each of judgements given above is decidable.

In the RLF logical framework, the meta-language is the $\lambda\Lambda$-calculus, a system of first-order dependent function types, with a full linear dependent function space, in propositions-as-types correspondence with a first-order minimal logic of bunched implications [86, 116, 147]. The language of LLF [29] is a subsystem of $\lambda\Lambda$-calculus, lacking a linear dependent function space;

3. A characterization of the mechanism by which object-logics are represented. In LF and RLF, the mechanism is *judgements-as-types*. In LLF [29], the intended mechanism is not clearly characterized.

This point of view can be summarized by the following slogan:

$$Framework = Language + Representation.$$

Each of the previous points, which are interdependent, has an impact on the use of the frameworks as formal theories of logics and as bases for formal theories of computation in specified logics, such as a search-based model of computation like logic programming.

Representation in Logical Frameworks. The *judgements-as-types* notion of representation is described for LF informally, except for examples of particular systems, in [72, 11]. It is described more carefully in [12] and, for RLF, in [86]. Following [144, 145, 146], it can be summarized as follows:

1. Consider object-logics as systems for deriving not bare propositions but rather judged propositions. For example, intuitionistic predicate logic is formulated as a system for deriving (or satisfying) judged propositions of the form $proof(\phi)$ or alternatively, as in [72], $true(\phi)$. Similarly, modal S4 is formulated using two judgements, as a system for deriving judgements of the form $true(\phi)$ and $valid(\phi)$. In both cases, the systems have Kripke semantics in the usual style;

2. Consider a correspondence, with a definition similar to that of the propositions-as-types correspondence, between judged propositions (in the object-logic) and types in the language of the framework (constructed over a signature containing type-constructors corresponding to each judgement form of the object-logic).

With this formulation, LF's representation of object-logics now goes as follows: roughly speaking, LF is concerned with those Hilbert and natural deduction systems for which the correspondence is *uniform*. The basic idea, inspired by [71], is that there be a surjection from consequences

$$\delta : (X, J_1(\phi_1), \ldots, J_m(\phi_m) \vdash_L J(\phi)),$$

in $\mathcal{L}$, where δ is the proof-object in $\mathcal{L}$, to consequences

$$\Gamma_X, y_1 : J_1(\phi_1), \ldots, y_m : J_m(\phi_m) \vdash_{\Sigma_L} M_\delta : J(\phi),$$

in $\Sigma_{\mathcal{L}}$.

We can observe [141, 86, 85] that linear logic and other relevant (or substructural) logics cannot be uniformly encoded in LF. To obtain such an encoding it is necessary to move to the RLF logical framework which, as described above, uses the $\lambda\Lambda$-calculus, together with judgements-as-types. $\lambda\Lambda$'s linear dependent function space can be used to encode, for example, some of the "consumable preconditions" found in Hoare-like program logics. This language with a judgements-as-types notion gives a framework able to uniformly encode some linear and other relevant, or substructural, logics.

All of the logical frameworks we have considered so far have employed judgements-as-types as their representation mechanism. Before proceeding to consider briefly the semantics of logical frameworks, we discuss some distinctions within the *judgements-as-types* mechanism and also a different mechanism, called *worlds-as-parameters*:

- *Judgements-as-types:* A number of forms of representation, within the judgements-as-types mechanism, have been discussed by Avron, Honsell, Miculan and Parvano [12]. Briefly, they are as follows:

  – The most basic use of judgements-as-types is when the object-logic to be represented is described by a single consequence relation. Examples of such systems include propositional and predicate classical and intuitionistic logics. Their encodings in LF employ a single type-constructor,

  $$\mathsf{true} : o \to \text{Type},$$

  which takes a proposition ϕ and returns a type $\mathsf{true}(\phi)$. The inference rules of the object-logic are then represented as constants of appropriate type. For example, the $\vee$-elimination rule of intuitionistic propositional logic,

  $$\frac{\phi \vee \psi \quad \begin{array}{c}[\phi]\\ \vdots \\ \chi\end{array} \quad \begin{array}{c}[\psi]\\ \vdots \\ \chi\end{array}}{\chi}$$

  is represented as follows:

  $$\begin{array}{lll} \vee E & : & \Pi\phi : o.\Pi\psi : o.\Pi\chi : o. \\ & & \mathsf{true}(\phi \vee \psi) \to (\mathsf{true}(\phi) \to \mathsf{true}(\chi)) \to (\mathsf{true}(\psi) \to \mathsf{true}(\chi)) \to \mathsf{true}(\chi) \end{array}$$

  Other rules are represented similarly; see [86, 141, 151] for discussions of the general form in LF and RLF;

  – If the definition of an object-logic simultaneously uses two (or more) consequence relations, then we can view the propositions of the logic as consisting not merely of well-formed formulae but of pairs $\langle \phi, \mathsf{j} \rangle$, where j denotes a choice of consequence relation. Then the logic can be represented in LF by taking a type-constructor $\mathsf{j} : o \to \text{Type}$, for each consequence relation. For example, the modal logic **K** uses two consequence relations, for truth and validity, and can be represented using two judgements, $\mathsf{true} : o \to \text{Type}$ and $\mathsf{valid} : o \to \text{Type}$;

- *Worlds-as-parameters:* A quite different representation mechanism, called *worlds-as-parameters*, has been described by Avron, Honsell, Miculan and Parvano [12]. It is inspired by Kripke models of modal logics [32]. For example, in a representation of a Hilbert-type system for **K**, the basic idea is to parametrize the type-constructor corresponding to the consequence relation for truth over a type U of "worlds", on which no constructor is defined,

  $$\mathsf{true} : U \to o \to \mathrm{Type}.$$

  This step allows the necessitation rule to be represented as

  $$NEC : \Pi\phi : o \,.\, (\Pi w : U(\mathsf{true}(w)(\phi))) \to \Pi w : U \,.\, \mathsf{true}(w)(\Box\phi).$$

  The idea [12] is that if we make an assumption, then we must assume the existence of a world and to form the judgement at w. It follows that deriving a judgement, which is universally quantified with respect to U, as a premiss amounts to establishing the judgement for a generic world upon which no assumptions are made. The Kripke idea of accessibility can be seen to arise via the inductive construction of contexts formed in order to represent judgements [12];

- *Constrained Assumptions:* A third class of mechanisms, which are really just variations on the basic judgements-as-types scheme, deals with a range of special conditions

  – *No Assumptions:* Logics such as **K** and its extensions employ a rule of *necessitation* which may be applied only to theorems. Rather than enforcing the structure required for such a rule via two consequence relations, one can introduce a "no assumptions" judgement,

  $$\mathsf{Na} : \Pi\phi : o.\mathsf{true}(\phi) \to \mathrm{Type},$$

  and work with a single consequence relation represented by the type-constructor true.

  – *Boxed Assumptions:* Prawitz's natural deduction presentation of **S4** includes the $\Box$-introduction rule,

  $$\begin{array}{cc} & \Box\Gamma \\ & \vdots \\ \Box I & \dfrac{\phi}{\Box\phi}, \end{array}$$

  the application of which we must restrict to proofs in which all assumptions are essentially modal. This restriction is represented using a judgement

  $$\mathsf{Bx} : \Pi\phi : o.\mathsf{true}(\phi) \to \mathrm{Type},$$

  which allows the representation of $\Box$-introduction as the constant

  $$\Box I : \Pi\phi : o \,.\, \Pi d : \mathsf{true}(\phi) \,.\, (\mathsf{Bx}(\phi)(d)) \to (\mathsf{true}(\Box\phi)).$$

Other variations along the lines of "boxed assumptions" are "closed assumptions" and "boxed fringe": All of these employ judgements which constrain the logics consequence relations [12], just as in the examples above.

We remark that [12] is concerned mainly with providing systematic treatments of object-logics described as systems for truth and object-logics described as systems for validity, rather than with different choices of representation mechanism, so that the organization of the ideas in [12] is different from ours.

Semantics of Logical Frameworks. We have already explained that a logical framework should be regarded as follows:

$$Framework = Language + Representation.$$

In consideration of a semantics for a logical framework — here we restrict our attention to LF, RLF [86] being beyond our present scope — we can see from the discussion above that there are three requirements that it must satisfy:

(1) It must provide a semantics for the *type theory* as a *theory of functions*;

(2) It must provide an account of the *propositions-as-types correspondence* between the type theory and its internal logic, and the *judgements-as-types correspondence* for 'uniform' encodings;

(3) It must provide a semantics for the notion of *logic programming* induced by search in the language of the framework.

For the first requirement (1) we must have a semantics for dependent types with dependent function spaces that, for example, properly extends the ideas of Mitchell and Moggi [112]. We must also have Kripke/Beth/Joyal models of the $\{\supset, \forall\}$-fragment of minimal first-order logic. For the second requirement (2), we must consider the correspondence between Kripke models of an object-logic and Kripke models of the encoding of that logic in the meta-logic. For the third requirement (3), we must at least be able to identify a class of Herbrand models and be able to provide a least fixed point construction corresponding, as usual, to resolution. We return to the third requirement in more detail in § 3.4.

We can satisfy requirements (1), (2) and (3) within the setting of indexed, or fibred, category theory [121, 87]. Suppose we have a category $\mathcal{E}$ where the propositions will be interpreted. We index $\mathcal{E}$ for the purposes of interpreting the type theory. First, we index it by a Kripke "world" structure $\mathcal{W}$. This is to let the functor category $[\mathcal{W}, \mathcal{E}]$ have enough strength to model the $\{\rightarrow, \forall\}$-fragment of the internal logic and so correspond to Kripke-style models for intuitionistic logic.

(An aside: In the setting of the relevant, or substructural, type theory presented by Ishtiaq and Pym in [86], we must further index $[\mathcal{W}, \mathcal{E}]$ by a resource monoid R. Thus, we obtain R-indexed sets of Kripke functors $\{\mathcal{J}_r{:}[\mathcal{W}, \mathcal{E}] \mid r \in R\}$. We remark that the separation of worlds from resources considered in this structure emphasizes a sort of "phase shift" [61, 76].)

We now consider how to model the propositions and so explicate the structure of $\mathcal{E}$. The basic judgement of the internal logic of the type theory is $(X)\Delta \vdash \phi$, that ϕ is a proposition in the context Δ over the context X. One reading of this judgement, and perhaps the most natural, is to see X as an index for the propositional judgement $\Delta \vdash \phi$. This reading can be extended to the type theory, where, in the basic judgement $\Gamma \vdash_\Sigma M{:}A$, Γ can be seen as an index for $M{:}A$ or that $M{:}A$ depends on Γ for its meaning. Thus we are led to using the technology of indexed category theory. More specifically, in the case of the type theory, the judgement $\Gamma \vdash_\Sigma M{:}A$ is modelled as the arrow $1 \stackrel{[\![M]\!]}{\to} [\![A]\!]$ in the fibre over $[\![\Gamma]\!]$ in the strict indexed category $\mathcal{E}{:}\mathcal{D}^{op} \to \mathcal{V}$, where $\mathcal{V}$ is a category of "values".

We remark that this is not the only technique for modelling a typing judgement; Cartmell [28], Pitts [138] and several other authors use a more "one-dimensional" structure which relies on the properties of certain classes of maps to model type-dependency and dependent function spaces. These are essentially equivalent to the indexed approach but the latter is appealing for the main reason that it provides a technical separation of conceptually separate issues. (Moreover, indexed techniques seem better suited to generalizations concerned with weaker type theories [86].) For instance, at a logical level, the base and fibres deal, respectively, with terms and propositions.

Nevertheless, these ideas owe much to work of Cartmell [28], Pitts [138], Seely [160], Streicher [167] and others. We take a Kripke *prestructure* to be a functor

$$\mathcal{J} : [\mathcal{W}, [\mathcal{D}^{op}, \mathcal{V}]],$$

where $\mathcal{W}$ is a small category (of worlds), $\mathcal{D}^{op} = \coprod_W \mathcal{D}_W^{op}$, where W ranges over the objects of $\mathcal{W}$, and each $\mathcal{D}_W$ (the base at W) is small; $\mathcal{V}$, a subcategory of the category of small categories, **Cat**, is a category of values such that:

1. Each $\mathcal{D}_W$ has a terminal object, $1_{\mathcal{D}_W}$;

2. Each $\mathcal{J}(W)(D)$ has a terminal object, $1_{\mathcal{J}(W)(D)}$, preserved on the nose by each $f^*(= \mathcal{J}(W)(f))$, where $E \xrightarrow{f} D \in \mathcal{D}_W$;

3. For each $W \in \mathcal{W}$, $D \in \mathcal{D}_W$ and $A \in \mathcal{J}(W)(D)$, there is a $D \bullet A \in \mathcal{D}_W$ together with canonical projections $D \bullet A \xrightarrow{p_{D,A}} D$, $1_{\mathcal{J}(W)(D\bullet A)} \xrightarrow{q_{D,A}} p^*_{D,A}(A)$ and canonical pullbacks

$$\begin{tikzcd} E \bullet f^*A \arrow[r, "f \bullet A"] \arrow[d, "p_{E,f^*A}"'] \arrow[dr, phantom, "\lrcorner", very near start] & D \bullet A \arrow[d, "p_{D,A}"] \\ E \arrow[r, "f"'] & D \end{tikzcd}$$

satisfying the *strictness conditions* that $1_D{}^*(A) = A$ and $1_D \bullet A = 1_{D\bullet A}$, for each A in $\mathcal{J}(W)(D)$, and that $g^*(f^*A) = (g;f)^*A$ and $(g \bullet (f^*A));(f \bullet A) = (g;f) \bullet A$, for each appropriate A, f and g. Moreover, for each W and D, $D \bullet 1_{\mathcal{J}(W)(D)} = D$;

4. At each W, the arrow $p^*_{D,A}(= \mathcal{J}(W)(p_{D,A}))$ has a right adjoint,

$$p^*_{D,A} \dashv \Pi_{D,A} : \mathcal{J}(W)(D \bullet A) \longrightarrow \mathcal{J}(W)(D)$$

satisfying the following (strict) *Beck-Chevalley* condition: for each $E \xrightarrow{f} D$ in $\mathcal{D}_W$, each A in $\mathcal{J}(W)(D)$ and each B in $\mathcal{J}(W)(D \bullet A)$,

$$f^*(\Pi_{D,A}B) = \Pi_{E,f^*A}((f \bullet A)^*B) \quad \text{and} \quad (f \bullet A)^*(app(A,B)) = app(f^*A, (f \bullet A)^*B),$$

where *app* is the co-unit of the adjunction. Thus Π interprets the dependent function space.

To get a Kripke structure $\mathcal{K}_{\mathcal{J}}$ for Σ we must move from the category of values $\mathcal{V}$ to a (weak) arrow construction $\vec{\mathcal{V}}$ on $\mathcal{V}$ (see below), so that $\mathcal{K}_{\mathcal{J}} : [\mathcal{W}, [\mathcal{D}^{op}, \vec{\mathcal{V}}]]$. A Kripke *model* of Σ is then given as a pair $\langle \mathcal{K}_{\mathcal{J}}, [\![-]\!]^{-}_{\mathcal{K}_{\mathcal{J}}} \rangle$, where $[\![-]\!]^{-}_{\mathcal{K}_{\mathcal{J}}}$ is a partial function that interprets the syntax of $\lambda\Pi$ in $\mathcal{K}_{\mathcal{J}}$ (so that we must require that $\mathcal{K}_{\mathcal{J}}$ has "enough points" to interpret the constants in Σ).

In fact, *prestructures* would be an adequate basis for defining models that would satisfy requirement (1). We define the following notion of satisfaction of the inhabitation of a type A (intended to be of kind Type, *i.e.*, not of the form $\lambda x : A \,.\, B$) by an object M with respect to context Γ at world W: let Σ be a signature, $\mathcal{K}_{\mathcal{J}} : [\mathcal{W}, [\mathcal{D}^{op}, \vec{\mathcal{V}}]]$ be a Kripke Σ-$\lambda\Pi$-model and let Γ be a context, A be a type and M be an object.[2] In the model $\mathcal{K}_{\mathcal{J}}$, the world W *satisfies* the inhabitation of A by M with respect to Γ, *i.e.*,

$$W \models^{\mathcal{K}_{\mathcal{J}}}_{\Sigma} (M : A)[\Gamma],$$

if and only if we have that $[\![\Gamma]\!]^W_{\mathcal{K}_{\mathcal{J}}}$, $[\![A_\Gamma]\!]^W_{\mathcal{K}_{\mathcal{J}}}$ and $[\![M_\Gamma]\!]^W_{\mathcal{K}_{\mathcal{J}}}$ are defined and that $1_{\mathcal{K}_{\mathcal{J}}(W)([\![\Gamma]\!]^W_{\mathcal{K}_{\mathcal{J}}})} \xrightarrow{[\![M_\Gamma]\!]^W_{\mathcal{K}_{\mathcal{J}}}} [\![A_\Gamma]\!]^W_{\mathcal{K}_{\mathcal{J}}}$ in $\mathcal{K}_{\mathcal{J}}(W)([\![\Gamma]\!]^W_{\mathcal{K}_{\mathcal{J}}})$.

We are then able to obtain, *inter alia*, the following familiar-looking properties (all of which are subject to requirements that interpretations $[\![-]\!]^{-}_{\mathcal{K}_{\mathcal{J}}}$ be suitably defined at appropriate worlds — with no "relativization"):

- *Monotonicity*: Let Σ be a signature and let $\langle \mathcal{K}_{\mathcal{J}}, [\![-]\!]^{-}_{\mathcal{K}_{\mathcal{J}}} \rangle$, where $\mathcal{K}_{\mathcal{J}} : [\mathcal{W}, [\mathcal{D}^{op}, \vec{\mathcal{V}}]]$, be a Kripke Σ-$\lambda\Pi$-model. If $W \models^{\mathcal{K}_{\mathcal{J}}}_{\Sigma} (M : A)[\Gamma]$ and if $W \xrightarrow{\alpha} W'$, then $W' \models^{\mathcal{K}_{\mathcal{J}}}_{\Sigma} (M : A)^{[\alpha]}[\Gamma]$;[3]

- Π-*forcing*: Let Σ be a signature and let $\langle \mathcal{K}_{\mathcal{J}}, [\![-]\!]^{-}_{\mathcal{K}_{\mathcal{J}}} \rangle$, where $\mathcal{K}_{\mathcal{J}} : [\mathcal{W}, [\mathcal{D}^{op}, \vec{\mathcal{V}}]]$, be a Kripke Σ-$\lambda\Pi$-model. $W \models^{\mathcal{K}_{\mathcal{J}}}_{\Sigma} (M : \Pi x : A \,.\, B)[\Gamma]$ if and only if, for all $W \xrightarrow{\alpha} W'$ and for all N such that $W' \models^{\mathcal{K}_{\mathcal{J}}}_{\Sigma} (N : A)^{[\alpha]}[\Gamma]$, there is a P such that $W' \models^{\mathcal{K}_{\mathcal{J}}}_{\Sigma} (P : B[N/x])^{[\alpha]}[\Gamma]$ and $P =_{\beta\eta} MN$. Similarly for the non-dependent function space, $\rightarrow$;

- *Soundness*: Let Σ be a signature and let $\langle \mathcal{K}_{\mathcal{J}}, [\![-]\!]^{-}_{\mathcal{K}_{\mathcal{J}}} \rangle$, where $\mathcal{K}_{\mathcal{J}} : [\mathcal{W}, [\mathcal{D}^{op}, \vec{\mathcal{V}}]]$, be a Kripke Σ-$\lambda\Pi$-model. If $\Gamma \vdash_\Sigma M : A$ is provable, then $W \models^{\mathcal{K}_{\mathcal{J}}}_{\Sigma} (M : A)[\Gamma]$, at each W for which all the required interpretations are defined;

[2]The types, objects and contexts considered here are required only to be members of the raw syntactic categories.

[3]The superscript $-^{[\alpha]}$ should be read as "after α".

- *Model existence*: There is a Kripke-Σ-$\lambda\Pi$-model $\langle \mathcal{K}_{\mathcal{J}\mathcal{T}}, [\![-]\!]^{-}_{\mathcal{K}_{\mathcal{J}\mathcal{T}}} \rangle$ with a world W_0 such that if $\Gamma \not\vdash_{\Sigma} M : A$, then $W_0 \not\models^{\mathcal{K}_{\mathcal{J}\mathcal{T}}}_{\Sigma} (M : A)[\Gamma]$;
- *Completeness*: If $\Gamma \models_{\Sigma} M : A$ — *i.e.*, satisfaction at all worlds in all models — holds, then $\Gamma \vdash_{\Sigma} M : A$ is provable.

However, for requirements (ii) and (iii), *structures* are exploited. The basic idea is as follows: a structure is obtained from a prestructure by replacing the category of "types" over a world and "context" by a *chosen* category of arrows from the base. Corresponding to structures, we can define the following satisfaction predicate, which is a generalization of the one above:

$$W \models\!\!\Rightarrow^{\mathcal{K}_{\mathcal{J}}}_{\Sigma} (\Delta \xrightarrow{\langle M_1,\ldots,M_n\rangle} \Theta)[\Gamma],$$

i.e., in the Kripke Σ-$\lambda\Pi$-model $\mathcal{K}_{\mathcal{J}}$, W forces $\Delta \xrightarrow{\langle M_1,\ldots,M_n\rangle} \Theta$ with respect to Γ.

The following can then be obtained:

(a) A semantics of $\lambda\Pi$ as a theory of functions. Such a theory requires just prestructures;

(b) Semantic accounts of propositions-as-types and of judgements-as-types for uniform LF encodings. Such a theory exploits structures, rather than just prestructures, so that we can interpret *consequences* relative to a given context. The idea is that a sequent $(X)\Delta \vdash \phi$ of a logic $\mathcal{L}$, with proof-object Φ, will hold at world W in a model $\mathcal{K}_{\mathcal{J}}$, just in case we have that

$$W \models\!\!\Rightarrow^{\mathcal{K}_{\mathcal{J}}}_{\Sigma_{\mathcal{L}}} (\Gamma_{\Delta} \xrightarrow{\sigma_{\Phi}} A_{\phi})[\Gamma_X],$$

where Γ_X is the context representing the first-order (say) variables X, Γ_{Δ} is the context representing the propositional assumptions Δ, A_{ϕ} is the type representing the proposition ϕ and σ_{Φ} is the realizer representing the proof-object Φ;

(c) A least fixed point semantics (see, for example, [77]) of logic programming with the $\lambda\Pi$-calculus, *i.e.*, requirement (3). Such a semantics is an essential starting point for understanding logic programming with a formal metatheory; it relies directly on the semantics of $\lambda\Pi$ we have presented.

The details of the ideas described in this section can be found in [141, 144, 145, 146, 86]. We discuss how our models satisfy requirement (iii) in § 3.4.

All these points and requirements about logical frameworks from both representation and semantics points of view well illustrate the necessary and difficult work to define appropriate and useful frameworks to represent and encode logics. It implies serious research about semantics foundations with relationships with category theory.

To conclude this section on logical frameworks, we insist upon the difference between the framework and its language. Frameworks such as LF or RLF are mainly used to represent and encode logics or deductive systems and to consider meta-theorems about these systems.

We have previously mentioned the $\lambda\mu$-calculus as an annotation language but we could also think about the status of such a calculus as the language of a *classical* logical framework. There are answers, but they require careful consideration of how one should represent systems in $\lambda\mu$-calculus.

3 Proof-search in Type-theoretic Languages

3.1 From Deduction to Construction (or Reduction)

Following in the Aristotelian tradition, modern symbolic logic has focussed on *deduction* as the primary proof-theoretic notion: Given a collection of *assumptions*, we construct their *consequences* by applying *rules of inference* to established propositions in order to establish more propositions.

There is, however, an alternative proof-theoretic notion, namely *proof-search*: Given a *sequent*, we attempt to *construct* a proof of it by applying inference rules as *reduction operators*, from conclusion to premisses, in order to repeatedly simplify the problem.

For example, consider the sequent $\phi, \phi \supset \psi, \phi \supset \chi \vdash \psi \wedge \chi$. This consequence does indeed hold, and can be established as follows:

$$\dfrac{\dfrac{\vdots}{\phi, \phi \supset \psi, \phi \supset \chi \vdash \psi}\supset \mathrm{L} \qquad \dfrac{\dfrac{}{\phi, \phi \supset \psi \vdash \phi}\,\mathrm{Axiom} \qquad \dfrac{}{\phi, \phi \supset \psi, \chi \vdash \chi}\,\mathrm{Axiom}}{\phi, \phi \supset \psi, \phi \supset \chi \vdash \chi}\supset \mathrm{L}}{\phi, \phi \supset \psi, \phi \supset \chi \vdash \psi \wedge \chi}\wedge\mathrm{R}$$

Here we see an intermediate stage in the construction of a proof. The first step is to break up the conjunction $\psi \wedge \chi$ on the right. Following the right hand branch, we choose to do a $\supset$L, using $\phi \supset \chi$, thereby creating two axioms. The left hand branch, here undeveloped, is completed similarly.

From the computational point of view, we have glossed over several important issues here. For example:

- We chose to develop the right-hand branch first. Alternatively, we could have chosen the left-hand one first, or developed them together, "in parallel". In more complex situations, a parallel execution can be very attractive, yet requires *communication* between the *processes* which calculate each branch;

- At each inference, we chose a proposition to reduce: for example, on the right-hand branch, we chose to reduce, using $\supset$L, $\phi \supset \chi$ rather than $\phi \supset \psi$.

An algorithm to calculate proofs, or decide putative consequences, must make such choices: it must resolve the *non-determinism* that is inherent in the problem. This highlights a view of logic which has emerged from such computational concerns, summarized by the slogan

Logic = Inference + Control.

Thus the nature of the reasoning determined by a system of logic depends not only on the inference rules (or indeed the satisfaction relation) but also on the régime which controls their use. These issues are very clearly seen in the logic programming language Prolog [59].

So *algorithmic* proof-search, in which a specific procedure for constructing a proof is given, is a fundamental enabling technology throughout computing science. Many problems are formulated as judgements about formal texts, ranging from familiar questions about logical consequence and well-formedness to type-checking in programming languages, parsing and compilation, whose solutions are determined by searching for derivations of such judgements.

3.2 Proof-search in Classical and Non-classical logics

There is a substantial and long-standing body of theory which addresses the problem of searching proofs in classical propositional and predicate logics. Many of these techniques have been extended to modal logics, intuitionistic logic and higher-order logic. As explained in subsection 2.4, the aim of using powerful logical languages or frameworks leads one to consider such classical or non-classical logics, as well as other logics, such as (classical) linear logic [62] in which *proofs* are not considered to be functions but rather are read as *actions*. Before considering proof-search in type-theoretic languages and its impact on programming, it is important to consider proof-search in the underlying logics. One can start from known methods for classical logic and adapt them to non-classical logics like intuitionistic, modal or linear, with proof-search there being considered as a perturbation of classical search. But the consequences of the resulting proof-search methods, concerning complexity, implementation difficulty and understanding, are difficult to estimate *a priori*.

At first, to reduce non-determinism in proof-search, one can use calculi that have certain structural advantages from the point of view of mechanization; leading examples are Gentzen's sequent calculi. Moreover, in addition to the design of proofs of cut-elimination, proof-search motivates the definition and the use of single- or multiple-conclusion sequent calculi for a given logic. It is important to notice that while natural deduction systems are appropriate for propositions-as-types correspondence but, *a priori*, much less so for proof-search. In first-order logic, one can eliminate the non-determinism in term-instantiation for an existential quantifier using Herbrand's theorem and unification. Even if Herbrand's theorem cannot be applied directly to most logics, such as intuitionistic, linear or modal logics, this non-determinism can often be eliminated via other procedures [94].

Another level of non-determinism has to do with the order in which rules are applied and in many sequent calculi, order of rules is crucial for the success of the proof-search process. The permutability results of a sequent calculus indicate when the order of two inference rules can be permuted without invalidating a proof and are used to reduce the non-determinism on the ordering of rules applications. Moreover, optimizations can be based on the reduction of the amount of backtracking in the proof-search as well illustrated in [94]. It is important to notice that a way to solve proof-search problems, such as the occurrence of loops in the search procedure, consists in proposing new sequent calculi, the rules of which integrate the solution mechanisms [42].

In this section, we briefly review some of the key aspects of some of the most important techniques of proof-search. Typically, the techniques we described were developed in the absence of any attempt to develop a general theory of proof-search, *i.e.*, a theory comparable to that which obtains for deduction. Moreover, these techniques typically do not exploit type-theoretic presentations of the logic to which they apply.

Resolution and Unification. Robinson's introduction of the *resolution* procedure [156] marks perhaps the beginning of the theory of proof-search. Resolution is a refutationally complete theorem proving method. Consider a first-order formula ϕ for which we aim to test validity. We first negate it and then try to skolemize $\neg\phi$ getting a formula of the form $\forall x_1 \ldots x_n.\Phi$ where Φ is a conjunction of clauses $C_1, C_2, \ldots, C_m$. Then ϕ is valid if and only if there is a finite set S of closed instances of clauses C_i, $1 \leq i \leq m$ that is unsatisfiable.

It corresponds to deducing the empty clause from S by the *resolution* and *factoring* rules. Resolution on ground clauses is a version of the cut rule restricted to atomic formulae and factoring is an instance of weakening. In fact, the main invention was to design *unification*, which can be seen as a device of interleaving the identification of suitable ground instances of clauses and the demonstration of their unsatisfiability.

This method has been extended to higher-order logic by Huet through the proposal of a semi-decision algorithm for higher-order unification (unification in the simply typed λ-calculus $\lambda_{\rightarrow}$) [81]. In spite of the undecidability of the problem, it has many applications in theorem proving, type inference and program transformation. Unification algorithms have been proposed for $\lambda\Pi$ by Elliot [44] and Pym [152, 143, 151] and have a significant impact in two main ways: such an algorithm allows to do automated theorem proving in LF's encoded logics but also to turn the $\lambda\Pi$ type-checking algorithm into a type-checking and term-inference algorithm for the encoded languages [22]. But the drawbacks of non-determinism and undecidability, inherent in the simply-typed case, were inherited by these algorithms. Therefore a deterministic, though incomplete, unification algorithm has been also proposed, which is based on a restriction of the occurrences of variables in simply typed λ-terms [132] and was generalized for the Calculus of Constructions. More recent work deals with similar higher-order unification in case of the linear simply typed λ-calculus [30].

In this setting, we mention *generic resolution*, also called the *inverse method*, originally developed by S. Maslov and G. Mints [110], a forward-chaining proof-search method. Search starts with the set of axioms and produces new sequents from the already derived ones by applying the sequent calculus rules in a "downwards" direction until one eventually derives the formula to prove [164]. In fact, the main ideas of the general resolution framework for logics with sub-formula property are (*i*) to label sub-formulae (of the formula to prove) with new atomic formulae in order to reduce the depth of a formula and (*ii*) to start search with maximally general axioms and builds unification into derivation rules. As for Robinson resolution, unification is an essential idea. This general method has been developed in the case of intuitionistic logic [164] and linear logic [111].

Strategies for using resolution have been proposed, including *subsumption*, which preserves the completeness of the method, and *inversion*, based on invertible rules in the logic. For some kinds of formulae, we can see that it is possible to allow only a single rule to be applied. It is the basis of the so-called *reduction* strategy for which a general schema for the resolution method was proposed in [168]. Such strategy has been adapted for linear logic and led to a linear resolution prover [163].

Methods Based on Matrices. Three main types of redundancy were identified within a search space induced by a classical cut-free sequent calculus: (i) *notational* (duplication of redundant information), (ii) *irrelevance* (reductions that do not advance the search towards finding a proof) and (iii) *non-permutability* (derivations that differ in the order in which rules are applied) [170]. Building on resolution, several techniques based on *tableaux*, *connections* [17, 170] or *matings* [6] were introduced first for classical logic and then modal and intuitionistic logics to remove if possible such redundancies. As an example, we consider the matrix (*i.e.*, connections or matings) methods of Andrews [6], Bibel [17] and Wallen [170]. The basic idea is simple and elegant. Consider, drawing on [170], the following classical sequent, which is provable in the Gentzen's [58] classical sequent calculus, LK,

$$\vdash (\phi \supset \psi) \wedge (\psi \supset \chi) \supset (\phi \supset \chi)$$

A matrix characterization of validity introduces appropriate theoretical structures and techniques for removing the above mentioned redundancies. One deals with the notational problem via the notions of *formula tree* and *positions* and with irrelevance via the notions of *path*, *polarity* and *connection*. As well explained in [170], the set of paths through a formula ϕ represents the set of sequents from which any derivation of the end sequent $\vdash \phi$ can be constructed. A necessary condition for a path to represent an initial sequent is to contain a connection: two atomic formulae occurrences with the same predicate symbol and with different polarities. A search for the connections in the paths is a direct search of a set of initial sequents from which $\vdash \phi$ is derivable. When every path through a formula contains a connection, one says that the set of connections *spans* the formula. For propositional logic, there is no permutability problem (internal structure of derivation is irrelevant), the existence of a spanning set of connections for ϕ is equivalent to the classical validity of ϕ (or the existence of a sequent proof for $\vdash \phi$).

Let us return to our example to illustrate the structures used in such a characterization. A *signed formula* is a pair $< \phi, n >$ where ϕ is a formula and $n \in \{0, 1\}$ is its *polarity* and such formulae are classified through different types (for propositional logic, conjunctive or α-type and disjunctive or β-type) considering the following *uniform notation*:

α	α_1	α_2
$< \phi \wedge \psi, 1 >$	$< \phi, 1 >$	$< \psi, 1 >$
$< \phi \vee \psi, 0 >$	$< \phi, 0 >$	$< \psi, 0 >$
$< \phi \rightarrow \psi, 0 >$	$< \phi, 1 >$	$< \psi, 0 >$
$< \neg\phi, 1 >$	$< \phi, 0 >$	$< \phi, 0 >$
$< \neg\phi, 0 >$	$< \phi, 1 >$	$< \phi, 1 >$

β	β_1	β_2
$< \phi \wedge \psi, 0 >$	$< \phi, 0 >$	$< \psi, 0 >$
$< \phi \vee \psi, 1 >$	$< \phi, 1 >$	$< \psi, 1 >$
$< \phi \rightarrow \psi, 1 >$	$< \phi, 0 >$	$< \psi, 1 >$

If ϕ is the formula to be proven, then we consider the signed formula $< \phi, 0 >$. Then one decomposes this signed formula into signed sub-formulae following the previous uniform notation and therefore forms an *indexed formula tree* where the nodes are *positions k* with labels $lab(k)$ that are the sub-formulae and a *polarity*. A position is associated a *principal type* and a *secondary type*. The former (α-type or β-type) is defined from its polarity and its label and the latter by the principal type of its parent in the tree. This information, for our example, is conveniently summarized as follows:

u	pol(u)	lab(u)	Ptype(u)	Stype(u)
a_0	0	$(\phi \supset \psi) \wedge (\psi \supset \chi) \supset (\phi \supset \chi)$	α	-
a_1	1	$(\phi \supset \psi) \wedge (\psi \supset \chi)$	α	α_1
a_2	1	$\phi \supset \psi$	β	α_1
a_3	0	ϕ	-	β_1
a_4	1	ψ	-	β_2
a_5	1	$\psi \supset \chi$	β	α_2
a_6	0	ψ	-	β_1
a_7	1	χ	-	β_2
a_8	0	$\phi \supset \chi$	α	α_2
a_9	1	ϕ	-	α_1
a_{10}	0	χ	-	α_2

A *path* through $< \phi, 0 >$ is a subset of the positions of its formula tree defined from the principal α and β types. After a reduction of a path we obtain an *atomic path* only composed by atomic positions. In our example, the atomic paths are the following: $\{a_3, a_6, a_9, a_{10}\}$, $\{a_4, a_6, a_9, a_{10}\}$, $\{a_3, a_7, a_9, a_{10}\}$, $\{a_4, a_7, a_9, a_{10}\}$.

A *connection* is a pair of atomic positions in a path labelled with identical atomic formulas of different polarities. Here one has a set of connections $\{(a_3, a_9), (a_4, a_6), (a_7, a_{10})\}$ that spans the initial formulae because each atomic path contains such a connection. Then the initial sequent is classically provable. Then proof-search corresponds in this setting to find connections that span the formula.

In first-order logic, the internal structure of derivations is important since the existential quantifier rules ($\exists_L$, $\forall_R$) are constrained by the eigenvariable condition. Therefore, one needs a mapping to represent the coherence of the choices of terms (substituted to free variables) to obtain a connection. Such a connection is called a *complementary connection*. Such mappings induce reduction orderings that represent the constraints on the order in which rules have to be applied. The mappings for which reduction orderings are irreflexive (*i.e.*, consistent with the structure of the formula) are called *admissible* mappings. Moreover, one introduces a notion of *multiplicity* that indicates how many instances of particular subformulae may occur in a derivation. This leads one to work with formulae indexed with a multiplicity. Then the existence of a multiplicity and of a set of connections which spans ϕ, and which are complementary under an admissible mapping, is equivalent to the first-order validity of ϕ. Let us mention that unification can be used to compute the appropriate admissible mappings. Its rôle consists in ensuring the existence of a correct order of rule applications to produce a proof.

Wallen has also developed this method for non-classical logics like the modal logics and the intuitionistic logic [170]. The proof-theoretic basis is a multiple-conclusioned sequent calculus for intuitionistic logic that differs only from the classical calculus in three so-called *special rules*. For these rules, the succedent of the premiss is restricted to the side-formula of the rule, whereas in the classical rules, the succedent may contain multiple formulae in the succedent [41]. Such special rules induce a sort of non-permutability. The matrix characterization for intuitionistic logic is based on adaptation of the notions used for classical logic. Concerning the complementarity, one associates with each position of the formula tree a *sequence of positions* called a *prefix* that encodes the context of that position with respect to the special positions. In fact, the use of prefixes is a *syntactic* way of taking into account *semantic* information. Therefore, the complementarity for connections is defined in terms of the prefixes of the atomic positions, from the notion of intuitionistic admissible mappings (or substitutions) that allow to unify prefixes of connections. A similar approach can be taken for various modal logics.

In fact, the matrix characterizations are not rivals to resolution methods. Rather, the later can be seen as a combination of a matrix inference system and particular search strategies encoded in the syntax of the object language.

Which Method ? Such a characterization can be generalized as to find a matrix M and a substitution σ such that every path in the matrix $M\sigma$ is inconsistent (*i.e.*, contains a formula and its negation). There are different ways of constructing such a matrix and checking its paths. The way, in the *tableaux method* can be seen as a construction of the *skeleton* of a sequent calculus derivation and of finding a *substitution* σ that makes the skeleton a derivation. In intuitionistic logic some complications come from the absence of prenex and Skolem normal forms. Moreover, as an alternative to the connection method, Voronkov [169] proposes a characterization based on derivation skeletons and *constraints* satisfaction and a study of the problem of the instantiations of a skeleton to valid derivations with some complexity results. This idea to use constraints instead of substitutions, used in different areas of deduction (unification, semantics) seems interesting for further developments.

These different points illustrate the possible characterizations of provability in a given logic and their corresponding proof-search methods. A common view is to consider provability search from a skeleton of derivation (for instance, a formula tree) and some constraints to satisfy (for instance, complementarity of connections), the right definitions of these notions depending on the logic with which one deals. Moreover, one can either fix a general methodology (or a unifying framework) for the common study of logics (for instance, relevant, or substructural, logics) and their interactions (non-classical search viewed as a perturbation of classical one) or develop from the specific proof-theoretic properties of the logic, new tailored methods. Proof-search in linear logic is a good example of such a dilemma because this logic cannot be considered, from different points of view (proof-theoretic, resource-sensitivity, semantics, applications), only as a next and direct refinement of classical and intuitionistic logics, mainly when one considers more than the multiplicative fragment (for instance the additive connectives). Therefore attempts to provide a *taxonomy of logics and techniques* are important and are today motivated by a desire to develop logical systems tailored to the needs of specific applications. Labelled Deductive Systems (LDS) [37] provide some foundations of such an approach and brings out the common structures underlying different logical systems. In this approach, one uses labelled formulae, expressions of the form $a : \phi$, where ϕ is a formula and a is a term of a labelling language or algebra. In fact, one can consider the algebraic interpretation of sequents to deal with *semantic consequence relations* which could, under suitable circumstances, be reformulated in terms of "labelled refutation systems" and so lead to generalizations of classical tableaux in which the semantics is, in a sense, incorporated into the syntax. For instance, starting from a sequent one can generate a proof tree with labels, which include semantic information, such as Kripke worlds. Provability (respectively non-provability) in a given relevant or substructural logic is then established by satisfying (respectively not satisfying) the *semantic constraints* given by the labels at the leaves, with respect to the provability conditions.

It is clear that relevant, or substructural, logics can be seen as being proof-theoretically motivated and their syntax seems to be better understood than their semantics. Intuitionistic logic is a good illustration about it even if its model theory is more developed than for weaker substructural logics. It appears that new improvements on proof-search will be based on a better understanding and manipulation of classical semantics (algebraic, Kripke-style) or new semantics (phase semantics, coherence spaces, games models).

What degree of automation ? An important aim consists in automating proof-search in such logics as much as possible, to avoid tedious work and to focus on important choices. Then, theorem provers and underlying proof-search procedures can be based for instance on the definition of appropriate proof-schemas, such as uniform proofs [109] or normal proofs [57], that are complete with respect to provability. But one could also prefer to develop proofs semi-automatically, with some choices about reducing non-determinism (principal formula, rule to apply) made by the user of the proof system. In interactive theorem provers, such as LCF [128] or Isabelle [131], where one can encode many logics and then develop proofs of meta-theorems in such logics, one can define *strategies* from well-defined notions of *tactics* and *tacticals*. In the context of type-theoretic languages and the use of type theory for programming, where we focus more on the proof-object, we observe that an interactive approach allows to take into account more operational parameters during the proof design. But it is clear that when one is able to define appropriate *proof plans* (from tactics or strategies) to automatically construct some classes of proofs, it is natural to integrate them into a proof-search procedure.

Proofs-as-objects. The proof-search process is at present (and will be more and more) influenced by the specific applications one deals with (design of logical systems tailored to their needs), by the integration in actual proof environments (unified framework for various logics or interaction between specific proof methods) and by the analysis of the proof-search (complexity, non-provability, interface). It is the same in the context of type-theoretic languages, but with an emphasis on the proof-object that strongly depends on the logical fragment and on the proof-search method.

In our last example, the resulting proof-object is a set of connections or matrices from which it is possible to reconstruct the corresponding sequent proofs. Therefore some question arises: *Is such proof-object (or proof representation) obtained by an efficient method, type-theoretic ? If not, how to define proof-search methods that directly generate type-theoretic proof-objects ? What are the proof-objects that we want to manipulate inside an implementation of a proof system, such as an automatic theorem prover or logic programming language, based on such efficient methods ?* Surely a user of such a system will expect to be able to inspect a proof in, say, the sequent calculus rather than a collection of matings ? To address this problem, reconstruction procedures are proposed to finally obtain corresponding sequent proofs [159]. But, in some cases, one could perhaps mix the search for provability (via connections) with a direct construction of proof-objects. The final form of proof-object is also important consideration in the design of a theorem prover: it can influence the form (among the many possible choices) of user interface [20].

3.3 Proof-search in Type-theoretic Languages: Basics

Whilst proof-search in logical systems in general is, essentially, concerned with the problem of trying to construct a proof of a sequent $\Gamma \vdash \Delta$, in which each of Γ and Δ may contain indeterminates (or "logical variables"). In type-theoretic settings, one typically tries to construct a proof of a sequent $\Gamma \vdash \Phi : \phi$ in which, as we have described in § 2, Φ denotes a proof-object for the sequent $\Gamma \vdash \phi$ and in which each of Γ, ϕ and Φ may contain indeterminates. Consequently, in the type-theoretic setting the search space of proofs is constrained by Φ: we consider just those proofs of the shape determined by Φ (of course, Φ may just be an indeterminate, in which case the constraint is trivial).

The shift to the type-theoretic point of view has a useful consequence. It can be argued that, from an essentially combinatorial point of view, the search space determined by classical sequent calculus, LK, underlies the search spaces for other logics or systems [91, 141, 154, 155]. Given this point of view, a natural question is the following: *How can the classical search space be used as a basis for proof-search in non-classical logics ?* The paper by *Ritter, Pym and Wallen*, in this volume, addresses this question in detail for propositional intuitionistic logic. The basic idea is as follows: (*i*) work with the presentation of classical logic as the $\lambda\mu$-calculus, introduced by Parigot [124]. Such a presentation can be seen as LK annotated with a class of λ-terms which include structural or control operators; (*ii*) consider a multiple-conclusioned presentation of intuitionistic logic, such as that given by Dummett [41]; (*iii*) search for proofs of chosen endsequents using the full power of LK. Having obtained a proof, examine the $\lambda\mu$-term with which it is annotated to decide whether an intuitionistically valid proof has been determined. These techniques have been applied to classical and intuitionistic resolution by Ritter, Pym and Wallen in [141]. But what can we deduce from these results about the real impact of $\lambda\mu$-calculus as a proof-object language ?

It seems that an interesting outstanding question is whether it would be possible to have general or generic proof-objects usable for classical, intuitionistic and linear logics and to analyse, from the structure of such proof-objects, the provability in one logic from the provability in another. An example, in which a characterization of cases in which classical provability entails intuitionistic provability, is provided by *Nadathur*'s paper, in this volume. Moreover, it would be interesting to find ways of characterizing which are the more useful sorts of proof-objects for such a programme. For example, we know that classical provability can be used to determine intuitionistic provability via $\lambda\mu$-terms but what about other types of proof-object, such as proof nets ? In a recent work a similar approach for intuitionistic provability has been developed in the case of linear logic by proposing labelled sequent calculi [13]. The use of labelled proof nets and the adaptation of related algorithms for proof nets construction [53, 55] allow to propose a method for linear intuitionistic provability that is complete. But what about the definition and the use of proof nets for other logics ?

In fact, proof-search can be analysed from an initial proof-theoretic point of view with studies of permutability of inference rules and of proof transformations. Then from the proof-search point of view, some choices can be fixed, leading to normal forms (canonical [57] or uniform [109]) of proofs that are complete for provability. Such methods for classical logic are often directly adapted (or specialized) to intuitionistic logic, considering it as a restriction of classical logic. For that, multiple-conclusion sequent calculi for intuitionistic logic (as a specialization of the one for classical logic) are essential for such an adaptation [42, 170] to benefit from the classical search. It is also interesting to study specific methods directly tailored for intuitionistic logic [42].

Studies of proof-search in relevant or substructural logics, such as linear logic, are often based on such results but with some specific restrictions; see, for example, [4, 56, 57, 141]. For a more detailed example, there have recently been attempts to develop connection methods for linear logic. One possibility is to extend the previous works on intuitionistic logic and then to keep uniformity inside a global proof-environment for constructive logics [92]. An alternative proposal comes from a direct analysis of proof nets and proposes, after defining a connection-based characterization, to use a proof net construction method in LL to derive a new connection method (see *Galmiche*'s paper, in this volume). In this setting, the proof-object is not only a set of connections but also a proof net (and also possibly the derived sequent proof). The generality of such techniques, based on new semantic structures such as proof nets, remains to be understood: *Are similar analyses possible for other relevant or substructural logics ? Can such techniques be integrated into logical frameworks such as LF [11, 72, 151, 141] or RLF [86] ?*

As one can see proof-search in non-classical logics as a perturbation of classical search, one can also consider, inside a non-classical logic, proof-search in some sub-fragments from this perturbation (or specialization) point of view. For instance, the non-commutative linear logic, that seems suitable for various applications, is in fact linear logic without the commutativity property for the consequence relation for which specific semantics and proof systems have been proposed [2]. But can proof-search methods in this fragment be derived from the ones in commutative case or must tailored techniques be developed ? In the case of proof-search based on proofs nets, one can naturally use the initial algorithm, designed for the commutative case, with a specialization of one specific procedure [54], but what about sequent calculi and natural deduction systems ?

Proposals for new decision procedures in a given logic are often based on new sequent calculi that directly integrate some operational choices at the levels of formula management manipulation and of proof development. For instance, to avoid some loops in the proof-search in propositional intuitionistic logic, Dyckhoff has proposed a contraction-free sequent calculus, exploiting the invertibility of some rules in order to reduce the amount of backtracking during proof-search [42]. In addition, a calculus for refutation that generates counter-models has been proposed [136]. These aspects illustrate the importance not only of proving formulae but also of justifying failing to prove a formula by giving evidence of non-provability, such as a counter-model. In such an approach, small application-specific models could usefully influence the design of refutation systems in a logic [162].

3.4 Proof-search in Logical Frameworks

We have explained (§ 2.5) that a logical framework should be understood to consist of a language together with a mechanism for representing logics. Although, from this point of view, there has to-date been very little work published on proof-search in logical frameworks, we speculate as to what a such a theory might look like.

In [151, 143], it was shown that the language $\lambda\Pi$ admits a natural interpretation as a logic programming language, based on a calculus $\mathbf{U}$ of sequents of the form $\Gamma \Rightarrow_\Sigma A(\alpha)$, where α is an indeterminate. Such sequents are interpreted as requests to calculate terms M and N such that $\Gamma \vdash_\Sigma M : A[N/\alpha]$ is provable. Here N corresponds to the usual notion of *answer substitution*, intended to be calculated by unification [151, 143, 152, 45]. A alternative formulation of logic programming for $\lambda\Pi$, the language of LF, has been presented by Pfenning [132].

From the semantic point of view, it is useful to formulate a *resolution calculus* for $\lambda\Pi$-realizations, $\Gamma \xrightarrow{\sigma} \Delta$. The following resolution calculus, $\mathbf{R}$, relies, for its completeness property with respect to the usual calculi for $\lambda\Pi$, on a certain *clausal form* for types [140, 141, 151]:

$$\text{Axiom} \quad \frac{}{\vdash_\Sigma \Gamma \xrightarrow{\langle @_1,\ldots,@_n \rangle} \Theta} \quad \textit{each } @_i \in \Sigma \cup \Gamma; \tag{1}$$

$$\text{Resolution} \quad \frac{\vdash_\Sigma \Gamma \overset{\langle M_1,\ldots,M_i,\ldots,M_n\rangle}{\longrightarrow} \Theta \qquad \Gamma \vdash_\Sigma M'_i : D_i[M_j/y_j]_{j=1}^{i-1}}{\vdash_\Sigma \Gamma \overset{\langle M_1,\ldots,M'_i,\ldots,M_n\rangle}{\longrightarrow} \Theta}, \tag{2}$$

where the clause $@_i : \Pi z_{i1} : B_{i1} \ldots \Pi z_{ip} : B_{im} . (C_{i1} \to (C_{i2} \to (\ldots (C_{iq} \to D_i) \ldots))) \in \Sigma \cup \Gamma$, $@_i P_1 \ldots P_p Q_1 \ldots Q_q \to_{\beta\eta} M'_i$, for some $1 \leq i \leq n$ and p, q possibly 0;

$$\text{Introduction} \quad \frac{\vdash_\Sigma \Gamma, x : A \overset{\langle M_1,\ldots,M_n,x,M\rangle}{\longrightarrow} \Gamma, x : A, y : B}{\vdash_\Sigma \Gamma \overset{\langle M_1,\ldots,M_n,\lambda x:A\,.\,M\rangle}{\longrightarrow} \Gamma, y : \Pi x : A \,.\, B} \tag{3}$$

and also a rule for $\beta\eta$-equalities.

Whilst the definition of resolution is appealing both proof- (see above) and model-theoretically (see below), from the point of view of implementation, we must employ a calculus which is defined using as little non-determinism as possible. Such a theory has been provided in [151, 143]. It is based on a calculus **L** with (semi-decidable) sequential judgements of the form $\Gamma \Rightarrow_\Sigma A$, which is interpreted as a request to calculate a term M such that $\Gamma \vdash_\Sigma M : A$ in the $\lambda\Pi$-calculus. This calculus L is then refined in two ways: (i) indeterminates are introduced to cope with the choice of terms in one specific rule leading to a calculus **U**, with sequents of the $\Gamma \Rightarrow_\Sigma A(\alpha)$ where α is an *indeterminate*. Sequents in **U** are interpreted as requests to calculate terms M and N such that $\Gamma \vdash_\Sigma M : A[N/\alpha]$ is provable, with N corresponding to the notion of answer substitution; (ii) **U**'s search space is quotiented by observing that some permutations in its derivations leave unchanged their $\lambda\Pi$-term extract and thus, when searching for typable terms, it suffices to find a representative of each of the classes so generated. Resolution calculi have been proposed for this calculus [140, 141] but dealing only with types in clausal forms. However, such clausal forms are sufficient to allow each object-level natural deduction inference rule to be represented by a single meta-level resolution step [151]. For example, the $\vee$-elimination rule of propositional intuitionistic logic, represented as

$$\begin{array}{lll} \vee E & : & \Pi\phi : o.\Pi\psi : o.\Pi\chi : o. \\ & & \mathsf{true}(\phi \vee \psi) \to (\mathsf{true}(\phi) \to \mathsf{true}(\chi)) \to (\mathsf{true}(\psi) \to \mathsf{true}(\chi)) \to \mathsf{true}(\chi) \end{array},$$

gives rise to the meta-level resolution step

$$\frac{\Gamma \vdash_{\Sigma_{\mathcal{IL}}} \mathsf{true}(\phi \vee \psi) \quad \Gamma, x : \mathsf{true}(\phi) \vdash_{\Sigma_{\mathcal{IL}}} \mathsf{true}(\chi) \quad \Gamma, y : \mathsf{true}(\psi) \vdash_{\Sigma_{\mathcal{IL}}} \mathsf{true}(\chi)}{\Gamma \vdash_{\Sigma_{\mathcal{IL}}} \mathsf{true}(\chi)}.$$

Another method based on resolution and unification and developed for the λ-cube type systems could be applied to this calculus [40]. More recently, a proposal for a new sequent calculus for $\lambda\Pi$, not requiring a clausal form for types, gives a one-one correspondence between typable terms of the calculus and the normal terms of the $\lambda\Pi$-calculus. It allows no permutations in the order in which inference rules occur on derivations of typable terms and is therefore appropriate for proof-search that can be performed in a bottom-up approach [137].

In the sequel, we develop here the different aspects and problems about this proof-search and the possible impact of proof-objects on this search process and on the connected computation analysis.

From the point of view of semantics, the key step is to identify a semantic counterpart to the resolution rule (2). The basic idea goes as follows:

- Define a class of *Herbrand* models by defining Herbrand prestructures and structures together with a suitable standard Herbrand interpretation. The prestructure for Herbrand models is identical to that required for model existence in requirement (i) of § (5), the salient feature being the construction of the category of worlds as the full subcategory of the base category of contexts in which each arrow is of the form $\Gamma \xrightarrow{\sigma} \Gamma, \Gamma'$. Herbrand structures, however, are a more delicate matter. An Herbrand structure $\mathcal{H}$ at world Δ and base Γ is a subset of the homset $\mathcal{C}_\Sigma(\Gamma, \Delta)$ of realizations between Γ and Δ.[4] It follows that such Herbrand structures form a complete lattice, with the least structure $\perp$ being that which assigns the empty set of arrows at each world and base;
- Define an operator T between Herbrand structures that can be considered to be a semantic counterpart to the resolution rule (2). Given an Herbrand structure $\mathcal{H}$, $\mathsf{T}(\mathcal{H})$ is the Herbrand structure built as follows: at each world Δ and each base Γ, add to $\mathcal{H}(\Delta)(\Gamma)$ all of those arrows that can be constructed by one resolution step from arrows that are already forced by $\mathcal{H}$ according to the predicate $\models\!\!\Rightarrow^{\mathcal{H}}_{\Sigma}$.

3.5 Implementations

Implementations of (the languages of) logical frameworks fall into two main types: interactive theorem provers and logic programming languages. Interactive theorem provers, such as LCF [65, 128], Isabelle [129, 130], LEGO [98], NuPRL [47], TPS [8], PVS [119, 118], ALF [100, 36], and Coq [127] have varying degrees of automated search built into them. The two main logic programming languages based on the language of (type-theoretic) logical frameworks are Miller and Nadathur's λProlog [113] and Pfenning's Elf [132]. The language of λProlog is higher-order hereditary Harrop formulae and has been used as a basis for a logical framework using a representation mechanism amounting to judgements-as-types. The language of Elf is the $\lambda\Pi$-calculus [72, 151, 140, 132], although the model of logic programming differs from that presented by Pym and Wallen [143, 152].[5] Meta-theoretic properties of deductive systems can be directly formulated relationally in Elf [132]. However, ensuring the validity of an Elf signature does not automatically guarantee the validity of the meta-theorems it contains. One needs to prove that these relations actually represent total functions in certain arguments and thus study of proof-search in the meta-theory is necessary. A recent proposal for that is the design of a schema-checker that allows one to show inductively input and output coverages and termination of the relations [157].

A theory of proof-search for a *logical framework* would need to build on a theory of proof-search in the *language* of the framework to take account of the representation of logics. Very little work has been done in this direction, however, so we sketch a possible approach. Our

[4]This situation can be considered to be a variation on the 'programs-as-worlds' view of the semantics of logic programs [107, 151].

[5]Pym's work higher-order unification for dependent types ($\lambda\Pi$) builds directly on Huet's seminal work in [81, 80]. See also [45].

point of departure is the calculus **R** considered together with our sketch of the judgements-as-types representation mechanism. We propose a resolution calculus, **OR**, tailored to the form of represented object-logics. For a given object-logic $\mathcal{L}$, represented by signature $\Sigma_{\mathcal{L}}$, **OR** is a calculus of judgements of the form

$$\Gamma_X \vdash_{\Sigma_{\mathcal{L}}} \Gamma_\Delta \xrightarrow{\sigma_\Phi} A_\phi,$$

corresponding to the sequent $(X)\Delta \vdash_{\mathcal{L}} \phi$, with proof-object Φ, in $\mathcal{L}$. Reduction steps in **OR** can then be seen to correspond to reduction steps in $\mathcal{L}$. Such a calculus can be interpreted in *structures*.

4 Proof-search and Programming

A proof-object is not only a representation of a proof-search process. It is also a realizer for a specification encoded by the sequent. For example, in intuitionistic logic, the λ-terms can be considered as programs satisfying the logical specification expressed with formulae. In case of classical linear logic, *e.g.*, in the context of planning, proof nets can be seen as correct plans from an initial state to a final one [105]. From this point of view, can we correlate the "quality" of the proof-search (or provability search), *i.e.*, efficiency, readability, simplicity and the "quality" of the resulting proof-objects, *i.e.*, efficiency of the programs or plans, readability ? The correctness of the proof-objects is a strong argument to use a proof-theoretic language as a programming language or logic but it is not clear if a "good" proof is a "good" program or if a "good" program can be also obtained by a proof-search [125].

Beside the general analysis of the relationships between logics, formal systems and annotations, a given application can force us to fix some choices by answering to the following questions: *What are the appropriate formal systems and annotation languages ? Are there existing proof-search methods we can use ? Are new proof-search methods or procedures needed ? What is the effective use of proof-search and of proof-objects in the context of programming, for instance with respect to proofs-as-programs, proofs-as-states, proofs-as-processes, proofs-as-computations or proof-search as computation paradigms ?*

It is also important to fix what are the non-determinisms we aim to keep at the specification and proof-search levels and also inside proof-objects. Moreover, one cannot ignore the influence of complexity results through this study and complexity of search problems in type-theoretic languages has to be analysed and compared with respect to non-type-theoretic systems.

Execution mechanisms both for logic and functional programming can be seen as searches for cut-free proofs. In logic programming, an attempt to achieve a goal, or query, G with respect to a program $\mathcal{P}$ can be seen as a search for a (special kind of) proof in intuitionistic logic of the sequent $\mathcal{P} \vdash G$. Typically, we are interested in goals of the form $\exists x.G$ and aim to extract from the calculated proof a term, or answer substitution, t, such that $\mathcal{P} \vdash G[t/x]$ is provable. Alternatively, via correspondences between types and propositions, evaluation in functional programming corresponds to the normalization of proofs in a natural deduction system.

4.1 Proof-search and Logic Programming

Whilst both theorem proving and logic programming can be understood in terms of proof-search, for logic programming we must impose more stringent requirements. Basically, we must insist upon having an *operational semantics* that is *not too non-deterministic*. A leading example of such an operational semantics is often summarized as "goal-directedness", in which the goal formula can be read as a search instruction according to the connectives of which it is comprised. For example, a goal $G_1 \wedge G_2$ is read as an instruction to calculate proofs of both G_1 *and* G_2. Similarly, a goal $G_1 \vee G_2$ is read as an instruction to calculate a proof of *at least one of* G_1 and G_2. A goal of the form $G_1 \supset G_2$, however, reveals a more subtle structure. It is read as an instruction to calculate a proof of the goal G_2 under the additional assumption, *i.e.*, additional program clause, G_1. It follows that the formula G_1 must be an instance of the class of formulae permitted to occur in programs. The structure program clauses must support another aspect of goal-direct operational semantics, that clauses be invoked only if all possible goal-directed search instructions have been applied, *i.e.*, the problem has been reduced to its simplest possible form. Such an operational semantics has an acceptably low level of non-determinism, choices being restricted to the selection of a program clause.

In Prolog-like languages, goal-directedness is characterized by *uniform proofs*, restricted to *hereditary Harrop formulae*, for which they are complete for logical consequence. A uniform proof is a cut-free proof in which every occurrence of a sequent whose right-hand side is non-atomic is the conclusion of a right rule. In this setting, a sequent $\Sigma\,; \Delta \vdash G$ can be used to represent the state of an idealized logic programming interpreter where Σ is the signature, or current set of non-logical constants, Δ is the current program and G is the current goal. These ideas have been discussed, for intuitionistic, classical and linear logics, in [67, 69, 107, 109, 113, 142, 152]. Languages which adopt this point of view include λProlog [113], Lolli [74], Lygon [69, 142, 171], Forum [108] and Elf [132, 133].

This study of searching for normal proofs, together with the connected studies about permutability and reduction of non-determinism, often leads to new equivalent sequent calculi that integrate the operational semantics at levels of both sequents and proofs. For instance, the actual presentations of systems like Lolli [74], Lygon [142, 171, 69] or Forum [108] are based on variants sequent calculi from which only uniform proofs can be directly built and for which sequents are refined in different parts. Then the operational meanings are involved at the logical level.

Specification Logics. We consider at first two views of logic programming based on LF [70, 11, 151]. Elf is a logic programming language based on types through the propositions-as-types that is suited for logic definition and metaprogramming [133]. Achieving a goal (type) G with respect to a program (context) Γ corresponds to the search of a closed object M of type G, where the language is determined by a signature Σ such that $\Gamma \vdash_\Sigma M : G$ is provable in LF. The answer to a query is not only a substitution for its free variables but a term of query type. The LF language admits another natural interpretation as a logic programming language, based on the calculus **U** (see § 2.5) with sequents of the $\Gamma \Rightarrow_\Sigma A(\alpha)$, where α is an indeterminate [143]. Such sequents are interpreted as requests to calculate terms M and N such that $\Gamma \vdash_\Sigma M : A[N/\alpha]$ is provable, with N corresponding to the notion of answer substitution.

Sequent calculi can be used only to express deductions between formulae of the logic but if we take into account a given interpretation (for instance *formulae-as-processes [89]*) they can be used to represent specifications [33], properties [106] or computations (for instance, *reduction* of processes, *i.e.*, $P \vdash P'$ interpreted as $P \rightsquigarrow P'$). Consider λProlog and its linear logic refinement, Lolli. They provide for various forms of abstraction (abstract data types, modules, higher-order programming) but lack primitives for concurrency. A logic programming language like LO [5] provides for concurrency but lacks abstraction mechanisms. To include concurrency, one typically deals with linear logic programming and therefore to extend this notion of goal-directed search to multiple-conclusion sequent.

Linear logic provides a view of formulae (occuring in contexts) as resources which can be exploited to model the notion of *state*. Various works emphasise the possibility to representing resource-based logics or imperative constructs but we aim to reason effectively about such representations. The Linear Logical Framework (LLF) [31] combines the expressive power of dependent types with linear logic to permit representations of state-based deductive systems. Efficient proof-search in the style of logic programming and based on uniform proof notion can be achieved in LLF.

A better view of formulae as resources, and associated notions of state, is provided by **BI**, the logic of bunched implications [115, 116, 147, 149, 150, 153]. Corresponding to **BI** is the logical framework RLF [148, 86], based on the $\lambda\Lambda$-calculus, a language with a full linear dependent function space. RLF has been used to represent weak logics and λ-calculi and the type system of ML with references [86]. RLF can also be interpreted as a logic programming language in the sense we have discussed.

As a specification logic, Forum modularly extends previous languages and allows specifications to include both concurrency and abstraction [108]. This meta-logic deals with sequents of the form $\Sigma : \Psi; \Delta \vdash \Gamma$ and $\Sigma : \Psi; \Delta \vdash^{B} \Gamma$, where Σ is a signature, Ψ a set of Σ-formulae, Δ a multiset of Σ-formulae, Γ a list of Σ-formulae and B a Σ-formula and is based on uniform provability. Most logical systems, like λProlog, Isabelle and LF, that are used for meta-level specifications of proof systems have been developed on intuitionistic principles. But, in such systems, specifications of sequent calculus proofs often need addition of various non-logical constants. In Forum, such specifications are natural and one can also handle substructural object-logics [108]. In this setting, there is a compromise, fixed by the completeness result, between provability (with uniform proofs) and expressiveness of the logical fragment.

We can also consider concurrency in fragments of linear logic with an interpretation in which formulae are processes or messages and connectives are algebraic operations on processes; an example is provided by the ACL system [88]. In this context, the non-determinism of proof-search corresponds to the non-determinism of execution. Because of some operational interpretations, it could be natural to have refinements of the sequent syntax. Such refinements are motivated, in context of specification or programming, to have simple and readable expressions and to take into account the operational interpretation that is also encoded in the corresponding sequent calculus. For instance, the choice between single-conclusion sequents and multiple-conclusion sequents, for a given logic, has consequences both on specification and proof-search. This point is illustrated by previous works based on

FILL for which we can define canonical (but not uniform) proof forms that are complete w.r.t. provability[21].

Inside such studies, a problem arises: *whether or not to give a logical formalization of some programming features or mechanisms*? For example, useful specification logics for concurrent and object-oriented programming can be proposed from refinements of linear logic but such specification problems lead to new proof-theoretic problems, including problems for proof-search [39, 90]. Let us mention among works attempting to reduce theories or concepts to logic, the ones that aim to reduce arithmetic in logic for which strategies for higher-order proof-search can be traced to strategies for arithmetic. In the paper by *Arai and Mints*, in this volume, is a proposal for a proof-search strategy in a cut-free logic which allows an exact modelling of cut-free arithmetic. This work could form a basis for interfaces to existing well-developed proof-search engines.

Strategies and implementations. The design of new logic programming languages and theorem provers, such as for fragments of linear logic, exposes new implementation problems and challenges, not present in traditional languages. The problem of efficiently managing the linear context, in systems such as Lolli [74] and Lygon [171, 69], is very important: a way to solve it consists in designing resource-management systems to eliminate the non-determinism in the distribution of linear formulae (see the paper by *Cervesato, Hodas and Pfenning*, in this volume). Thus, the efficiency of implemented systems, expected to be used for non-trivial applications, is improved through improved proof-search processes. Such strategy has been recently adapted for multi-conclusion sequent calculi, for instance in the Forum system [75] and in Lygon [69, 171]. Let us recall that linear logic programming languages are divided into those implemented sequentially and those which are intended as concurrent languages [89, 90] where non-determinisms in proof-search corresponds to non-determinisms in the computation. A similar study presents a characterization of a range of strategies for distributing and selecting resources in linear sequent calculus proof-search [68]. It is based on a sequent calculus annotated with boolean constraints and strategies are characterized by calculations of solutions of sets of boolean equations generated by searches. Such a characterization encompasses local, global or intermediate strategies. An appropriate implementation of resource-proofs would be to use a finite domain constraint logic programming in order to provide an appropriate mix of proof-search techniques and boolean constraints solving methods. Concerning complexity, it seems possible to exploit the essential restrictions of linear logic programming to hereditary Harrop formulae and concepts like paths [142, 68] to partition the sets of boolean equations obtained into smaller solvable collections.

Extensions. There are proposals to extend simply typed hereditary Harrop formulae with definitions. In fact, the use of definitions permits the construction of clearer programs and of shorter proofs by using a definition rule similar to the Gentzen's cut rule. Such a definition mechanism can be used to find proofs that are exponentially shorter than their variants formed by means of neither the definition mechanism nor the cut rule [134]. In the extension proposed in [135], definitions can be used as abbreviations but can also be employed in searching for shorter proofs of a goal and therefore a goal-directed search procedure is defined and shown to be complete for the extended type system. In this setting, we are also interested to solve the problem of redundancy in the search space by having a search procedure that produces exactly one proof (member of each equivalence class). It is clear that open problems to address in the future consists of unification in the presence of definitions and recursion at the level of simple types. To reason, with the proof-search paradigm, a possible approach consists in introducing new inference rules both for dealing with induction and for treating logical specifications as definitions. The paper by *McDowell and Miller*, in this volume, proposes such an extension of intuitionistic logic and proves the cut-elimination theorem, which is important both for automation of proof-search and consistency of the logic.

4.2 Proof-search and Functional Programming

The proofs-as-programs paradigm in programming mainly consists in extracting programs from proofs in intuitionistic logic. This point of view has been studied in more generality by Martin-Löf who has developed a system so-called Martin-Löf Type Theory [104] that is at the same time a programming language and the analogue for constructive mathematics of what the ZF system of set theory is for classical mathematics [114].

Program Synthesis. Such a system proposes a view of programming wherein the notion of type is identified with the (intuitionistic) notion of set and the notion of specification, which may be seen as a task or problem to be solved, with the (intuitionistic) notion of proposition. The solution of such a problem is an algorithm that may be viewed again as an element of the set or an object of the corresponding type. In this spirit, we can consider the notions of type and proposition to be identical and then present a single calculus for the calculus of natural deduction and for the functional calculus. Therefore we directly manipulate judgements of the form $\Phi : \phi$ with appropriate type-theoretic deduction systems. To show that a certain type ϕ is inhabited by constructing a term Φ of that type corresponds to show that a certain logical proposition ϕ is valid by constructing a object-proof Φ that proves it and also to show that a certain logical specification ϕ is satisfied by constructing a (functional) program Φ that satisfies it. In this context, we write $\Gamma \vdash_{tt} \Phi : \phi$ to denote that Φ is a proof-object of proposition-type ϕ in the type theory tt.

Simple type theories have been extended to second- and higher-order logic [14] and so permit universal type quantification, *i.e.*, $(\forall x : Type)\phi$. Moreover, we can add dependent types without losing desirable properties of the system, such as normalization. Second-order systems like the Calculus of Constructions (CC) [35, 127] or AF_2 [122], issued from studies of the system F [64], can be used for program synthesis, *i.e.*, to extract programs from proofs of logical specifications.

Recursive data types, as well as logical connectives, can be defined in the CC type system by impredicative encoding. But it is impossible to derive an inductive schema from such a definition. To overcome this problem, extensions of the type theory by inductive definitions have been proposed. In case of LF, in which types may only depend on individuals, we can do proof-search in first order implicational predicate calculus. If we add inductive definitions to LF we can do proof-search in full first-order predicate calculus with recursive data types. But to make the search space of proofs smaller we have to add more reduction rules. Then, after proving properties (Church-Rosser and Strong Normalization) of the resulting system and then classifying normal forms, proof-search and unification procedures needs new appropriate operations. In case of the AF_2 system, an alternative second-order type system for programming with proofs [101], inductive schemas are included in the logic and allows to use so-called recursive or iterative data types defined in AF_2. Automated proof-search, based on specific normal proofs (so-called recursive) that are complete with respect to provability, has been proposed for this framework [125]. But the extracted program is not always in this case the more efficient we can obtain. In the setting of constructive program synthesis, it can be better to have more flexible and interactive proof methods to be able to synthesis the better programs and not the better proofs.

Because of the relationships between propositions and types and between proofs and programs it is clear that automated proof-search in type theory can be used not only to find a proof-object Φ but also to use it for a computational purpose [50]. Viewing proof-objects as programs can influence the study of proof-search to take into account both the computational content and the complexity of the resulting proofs.

Automated Proof-search. In the context of constructive program synthesis, as the interactive nature of proof process stands in contrast to an efficient program development, efforts have also to be made to support automated proof-search in some fragments of the type theory. There exist intuitionistic type theories that can be encoded in first-order intuitionistic logic and mainly fragments where such an encoding is direct [161]. Consequently problems of proof-search in type theory are directly translated into problems in proof-search in intuitionistic logic. Moreover one optimises such a translation for enhancing the efficiency of automated proof-search for the initial problem. Such encoding has been developed from MLTT [161] but also from LF to the logic hh^w of hereditary Harrop formulae with quantification at all non-predicate types [48]. We can also consider an alternative and dual use of such encodings. As finding proofs corresponds to finding inhabitants of types one can in the case of implicational logics, specify the set of all the λ-terms representing proofs in these logics. Therefore, new proof-search algorithms have been designed from the search of inhabitants (see *Bunder*'s paper in this volume).

Without such encodings, it is also possible to adapt or integrate more classical proof-search into a program synthesis environment. For example, in [19], a new tableaux-based calculus for first-order intuitionistic logic is proposed from the classical tableau calculus extended with λ-terms. One benefit is that one can use proof-search methods known in classical logic, avoiding the order dependence of rule applications. In a similar setting, a particular proof-search procedure designed for higher-order logic, has been recently used to improve proof-search in the Calculus of Constructions (see *Felty*'s paper in this volume). The initial search procedure in CC [40] has been reformulated, with the introduction of a new notion called "search context". Such a generic notion could be adapted for another type systems like $\lambda\Pi$ or hohH (higher-order hereditary formulae) and lead to new proof-search methods. It is clear that with use of classical theorem proving methods into a program synthesis environment, the problem of methods integration arises the related issues of proof representations, proof transformations and of complexity. For example, a system like Nuprl [47] integrates a variety of interactively controlled and automated techniques for theorem proving and algorithm design. This has led to studies of the integration of an automated theorem prover for intuitionistic logic into such an environment, mainly for the purely logical parts of a type-theoretic proof. For such parts, one can use an efficient intuitionistic matrix prover but the resulting proof, based on a multiple-conclusion sequent calculus, has to be transformed into a standard intuitionistic proof which can be integrated as a proof plan for solving the initial formula [43]. Another approach consists in embedding classical proof-objects, like first-order tableaux, into a type system (adapted or extended for this purpose) by conversion into λ-terms representing proofs in such a system.

Extraction and Verification. Parts of a constructive proof may be irrelevant to the actual computation which results; for example, a proof by induction may require a particular measure to demonstrate that the induction is well-founded. Although it is possible to reason classically in a constructive system, such reasoning is not generally well-supported. Then one may wish to use a proof-assistant for classical logic as an aid to develop a program via a proof in a constructive type theory. Some steps have been taken towards such a connection providing automatic assistance for program verification, for instance with Nuprl [79] and Coq [120]. Such work is motivated by the possibility of applying classical tools in constructive proofs developments but also from the classical side of considering classical proofs as programs [126] and identifying executable subsets of classical theory to execute specifications as prototypes. The paper by *Caldwell, Gent and Underwood*, in this volume, presents a fine analysis of the problem "from extraction to verification and back", using the classical language of PVS [118] and the type theory of Nuprl.

Proof Transformations. Recall that a proof transformation procedure is mainly based on hierarchical system of permutation steps. Several works have proposed proof transformations (and their implementation) applicable to the optimization of extracted programs for instance in systems such as LF [3] or Nuprl [99]. An alternative approach of program transformation in type theories consists in formalizing methods and techniques of program transformations directly in the considered type theory — with a clear impact on the proof-search process. It leads to the study of the proof theory of certain elementary program derivations and to provide a semantics for the transformations as derivations in a type theory. In the setting of [43] another type of proof transformations, from a formal system to another one, has to be studied but with accurate investigation of the *complexity* of the resulting proofs. Moreover, the transformations must preserve the intended sub-specification of the program to be synthesised. One could have an exponential increase of proof length by such transformation but the program term can benefit from such an approach.

4.3 Induction

The automation of inductive proof has been appreciated to be a difficult problem for a long time [23]. We conclude this section by briefly remarking upon the work of Bundy *et al.* on a collection of systems designed to find proofs involving induction rules, in the context of program synthesis and extraction.

Bundy *et al.*'s key idea is that of a *proof plan* to guide (inductive) proof [24, 25]. Proof plans work by analyzing the syntactic structure of the inductive search problem to identify appropriate induction hypotheses, the key technique being *rippling*. Recent work has been to extend this approach to higher-order proofs, via an interface to HOL [66], and to consider the dynamic generation of plans. The idea of a *critic* [83, 84] has been introduced to make productive use of failure. Applications include problems in *hardware verification* [27]. Let us mention that similar works have been developed in the setting of the AF_2 framework with the definition of specific induction strategies and proof plans for given specifications [51] and illustrate some difficulties to design appropriate inductive proof-search methods in such type theories dedicated to program synthesis.

A number of systems have been implemented, allowing for a good deal of experimental work, including the automation, using rippling, of inductive proofs. Clam is a proof planning system for Oyster [26], a tactic-based implementation of the constructive type theory of Martin-Löf. Clam uses pre- and post-conditions of Oyster tactics as a basis for searching for plans. Once a plan for a given goal has been found, it can be expected that the resulting tactic will solve the goal. Experimental experience shows that the search space for plans is often small enough to allow the automatic calculation of plans to be tractable. A more recent implementation is the XBarnacle system [97].

5 Looking Forward: Semantics and Proof-search

It is clear that the semantics of a logic can, in many cases, have a strong influence on the design of proof-search procedures; indeed, the semantics of the logic is often used, more or less explicitly, to specify procedures. It follows that one possible way to analyse or propose new proof-search procedures could consist in analysing the semantics of the logic and considering its connection to the proof-search process. Model elimination and resolution [49] can both be understood in this way.

5.1 Which Semantics ?

Relevant, or substructural, logics are often syntactically or proof-theoretically motivated and their syntax often seems to be better understood than their semantics. They have typically *algebraic* and sometimes *categorical* models that are not far removed from the syntax. One also considers Kripke-style models that are something where formulae are mapped by valuations to set of *worlds* and relations and operations on worlds are used to formulate semantic clauses for the connectives. In fact, it defines an algebraic model in the powerset of the set of the worlds. For example, Kripke models for intuitionistic logic appear natural because everything is reduced to an ordering of the set of worlds and the semantic clauses for connectives are quite natural. Other kinds of semantics have been introduced and studied in recent years: phase structures [61], quantales [172], coherence semantics [61] and one has a deeper understanding of their strong relationships [117]. Therefore it is possible to grasp the essential meaning of *completeness theorems* with respect to these semantics. Moreover, the notion of resource can be studied from the semantic point of view, for example, by requiring a resource to be a preordered commutative monoid as in the *Kripke resource semantics* [116, 147, 115, 149, 150, 153]. It is not clear if Kripke-style semantics is what is needed for some relevant, or substructural, logics and their applications. Different sorts of semantics, inspired by proof theory or category theory [93], may be better adapted.

5.2 Semantics and Proofs

We illustrate some points about semantics and proof-search by considering some studies of fragments of linear logic. For instance, Avron [10] proposes several simple algebraic models of multiplicative and multiplicative-additive fragments and demonstrates the interest of such models by proving some unexcepted proof-theoretical properties of these logical fragments. Related studies in **BI**, the logic of bunched implications, can be found in [116, 147, 149, 150, 153].

The study of completeness results is important. For instance, it is well-known that Petri nets provide a sound and complete model for the $\oplus$-free fragment of intuitionistic linear logic (ILL) [46]. But with such a model, one cannot show, for example, that the following sequent, $(\phi \oplus \psi) \,\&\, \chi \vdash (\phi \oplus \chi) \,\&\, (\psi \oplus \chi)$ is not provable in ILL. For that we need a new interpretation that is complete and it leads to revisit the semantics of the logic and the basics about completeness (closure, completion). From the relationships between the notions of *ordered monoid* and of *quantale* and a new closure operator, one defines a new interpretation of ILL on Petri nets that allows to naturally give a counter-example for the above formula [52]. Then, such revised semantics leads to disprove some properties. Further work could be to derive new Kripke-style or algebraic semantics as foundations for new proof-and-refutation systems with, if possible, methods for the effective construction of counter-models.

The usual completeness theorems are stated with respect to provability. But there is a challenge of obtaining a *full completeness theorem*, of the kind found in categorical logic, that is with respect to proofs. Such a requirement is particularly strong in the presence of *indeterminates*, a pervasive tool in proof-search. With such a completeness, one has a strong connection between syntax and semantics. A first attempt has been made with a *games semantics* for linear logic [1] in which formulae denote games and proofs denote winning strategies. The completeness result for it says that every strategy in the model is the denotation of some proof. This semantics could naturally have a strong impact on automated construction of proofs or proof nets. It seems to capture more naturally the dynamical intuitions behind linear logic and it potentially provides a unifying framework for the semantics of computation in which concurrent processes, typed functional languages and complexity are handled in a integrated way.

5.3 Semantics and Complexity

To summarize these different aspects of studying semantics for proof-search purposes, we can say that such improvements on semantics encourage to transfer them at the syntactical level in the design of new methods. It could be done with annotations (or labels) that allow better characterizations of the dependencies and relationships between formulae that lead to (non-)provability. In fact, such works on semantics have complementary goals. One is to develop structures and algorithms for proof-search dedicated to various logics having in fact the same semantical foundations (see [37]). Another is, in a given logic, to improve the proof-search and to analyse its complexity. For example, in linear logic, connections between proof-search and probabilistic games, used in complexity theory, have been investigated [95] and so it follows that linear logic proof-search can be seen as a game. Such game is played on formulae and its moves are instances of inference rules and the results issued from this approach are for instance lower bounds for local proof-search [96]. The investigations about semantics can be the source of the design of new equivalent (sequent) calculi and induced proof-search methods with good complexity results (for instance polynomial proof-search systems).

5.4 Semantics for Search-based Computation

We have previously explained, in § 4.1 how the requirements of *operational semantics* can be used to design logic programming languages and their execution procedures. We remark that the operational semantics can be viewed both in terms of proof systems — via uniform proofs or resolution, *etc.* — or in terms of models — via a fixed point construction of a term model. We also remark that neither of these points of view provides an adequate account of of the execution dynamics of logic programming. For example, least fixed point models of the kind described in, for example, [77, 107, 151] provide no account of details such as clause selection, backtracking or, indeed, Prolog's imperative constructs such as "assert", "retract" and "!".

Towards a New Approach. We suggest that one potentially valuable approach towards solving these problems lies in the provision of more sophisticated *denotational semantics* for search-based computation. Such as semantics must capture the use of a proof system not as a calculus for deduction but rather as a calculus for construction (or reduction) as discussed in § 3.1 and this, as we sketch below, does indeed seem possible. However, we must also ask that such a semantics be capable of interpreting the dynamic aspects of the construction of proofs so as, for example, to be able to distinguish between breadth- and depth-first strategies. A first example is, perhaps, provided by Schmidt's denotational treatment of backtracking using *Success continuations* and *Failure continuations* [158]. Given such a semantic account, we may, for example, aim to focus on different specific models for different purposes. For example, we may wish to work with non-provability via the generation of counter-models.

Semantic Foundations for Search Spaces. We have seen, in § 2.3, that a *derivation* of a sequent $\Gamma \vdash \phi$ is interpreted by a map $f : [\![\Gamma]\!]_{\mathcal{M}} \longrightarrow [\![\phi]\!]_{\mathcal{M}}$ in a model $\mathcal{M}$. Such a map is, typically, defined by induction on structure of the (semantics of) the sequent. In this setting, a (two-premiss, say) rule of inference R is interpreted as a a map $F : \mathcal{M} \times \mathcal{M} \longrightarrow \mathcal{M}$, so that if f interprets a proof Φ of $\Gamma \vdash \phi$, f' interprets a proof Φ' of $\Gamma' \vdash \phi'$ and g interprets a proof $\Psi(\Phi, \Phi')$ of $\Delta(\Gamma, \Gamma') \vdash \psi(\phi, \phi')$, where

$$\frac{\Gamma \vdash^{\Phi} \phi \qquad \Gamma' \vdash^{\Phi'} \phi'}{\Delta(\Gamma, \Gamma') \vdash^{\Psi(\Phi, \Phi')} \psi(\phi, \phi')} \quad R,$$

then $F : (f, f') \mapsto g$.

Recall that in search-based computation, we do not have a given derivation. Rather, we start, prototypically, with a sequent

$$(\Gamma \vdash \phi)(X)$$

in which there is a "logical variable", or indeterminate, X. We aim to calculate a term t such that there is a derivation Φ of the sequent $\Gamma \vdash \phi[t/X]$. However, it may be that there is no such term and no such proof, even in the propositional case, without logical variables. It follows that the objects of semantic interest are sequents $\Gamma \vdash^{\Phi} \phi$ in which all of Γ, ϕ *and* ϕ may depend upon logical variables. Prototypically, an endsequent is annotated with a proof-object consisting exactly of an logical variable, $\Gamma \vdash^{X} \phi$, so that a reduction operator O, *i.e.*, a rule, such as R,

$$\frac{\Gamma \vdash^{Y} \phi \qquad \Gamma' \vdash^{Y'} \phi'}{\Gamma \vdash^{X} \psi} \quad O,$$

read backwards, from conclusion to premisses, is interpreted in a model $\mathcal{M}$ not as a map from $\mathcal{M}$ to $\mathcal{M} \times \mathcal{M}$ but rather as a map

$$F : \mathcal{M}(\alpha) \longrightarrow \mathcal{M}(\beta) \times \mathcal{M}(\beta')$$

where α, β and β' are *indeterminates* freely adjoined to $\mathcal{M}$ and used to interpret X, Y and Y', respectively. Thus the semantics of a calculus viewed as a system of reduction operators can be seen as being given by a polynomial construction [93] over the semantics of the calculus view as a system of inference rules.

6 Summary

We have presented our view of the scope and state of study of proof-search in type-theoretic languages. We have tried to place the study in its broader logical and computational context, paying particular attention to topics which seem to us to be somewhat underdeveloped. The key points can be conveniently summarized as follows:

- We considered what it is, in logical terms, to be a "type-theoretic language";
- We considered the logical status of proof-search and reviewed some of the key techniques that have been developed;
- We have considered what is the proof-search problem in type-theoretic languages and have, using a leading example drawn from logical frameworks, illustrated some of the objectives and techniques;
- Turning, more specifically, to programming we have discussed the rôle of proof-search theory in both the logic and functional programming paradigms. We have paid attention to a number of specific issues, including:
  - Strategies and Implementations of Logic Programming;
  - Extensions of Logic Programming;
  - (Functional) Program Synthesis (including automation);
  - Extraction and Verification;
  - Transformations; and

– Induction;

- Finally, we have proposed the value of semantic methods in the study of proof-search, drawing upon our examples and suggesting some directions for research.

Throughout this development, we have indicated how the papers in this volume contribute to the development of the ideas and techniques discussed. Indeed, our overview has been partly driven by the scope of the contributed papers.

Finally, we remark that our view is necessarily incomplete and biased towards our own interests. We offer our apologies to anyone to whom we may have been inadvertently unjust.

Acknowledgments. We are grateful to Maurice Nivat for encouraging us to produce this article and for acting as a referee. Any remaining errors are our responsibility.

References

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

[2] V.M. Abrusci. Phase semantics and sequent calculus for pure noncommutative classical linear propositional logic. *Journal of Symbolic Logic*, 56(4):1403–1451, 1991.

[3] P. Anderson. Representing proof transformations for program optimization. In *12th Int. Conference on Automated Deduction, CADE-12, LNAI 814*, pages 575–589, Nancy, France, July 1994.

[4] J.M. Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3):297–347, 1992.

[5] J.M. Andreoli and R. Pareschi. Linear objects: logical processes with built-in inheritance. *New Generation Computing*, 9:445–473, 1991.

[6] P. Andrews. Theorem proving via general mating. *Journal of ACM*, 28(2):193–214, 1981.

[7] P.B. Andrews. On connections and higher-order logic. *Journal of Automated Reasoning*, 5:257–291, 1989.

[8] P.B. Andrews, M. Bishop, S. Issar, D. Nesmith, F. Pfenning, and H. Xi. TPS: A theorem proving system for classical type theory. *Journal of Automated Reasoning*, 16:321–353, 1996.

[9] A. Avron. Simple consequence relations. *Information and Computation*, 92:105–139, 1991.

[10] A. Avron. Some properties of linear logic proved by semantics methods. *Journal of Logic and Computation*, 4(6):929–938, 1994.

[11] A. Avron, F. Honsell, I.A. Mason, and R. Pollack. Using typed lambda calculus to implement formal systems on a machine. *Journal of Automated Reasoning*, 9:309–354, 1992.

[12] A. Avron, F. Honsell, M. Miculan, and C. Paravano. Encoding modal logics in logical frameworks. *Studia Logica*, 60(1), 1998.

[13] V. Balat and D. Galmiche. Proof systems for Intuitionistic Provability in Linear Logic. In *Int. Workshop on Labelled Deduction LD'98*, Freiburg, Germany, September 1998.

[14] H. P. Barendregt. Lambda calculi with types. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Background: Computational Structures*, volume 2 of *Handbook of Logic in Computer Science*, pages 117–309. Oxford University Press, Oxford, England, 1992.

[15] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, Amsterdam, (revised edition) 1984.

[16] N. Benton, G. Bierman, V. de Paiva, and M. Hyland. Linear λ-calculus and categorical models revisited. In *6th Workshop CSL'92, LNCS 702*, pages 61–84, San Miniato, Italy, 1992.

[17] W. Bibel. On matrices with connections. *Journal of ACM*, 28(4):633–645, 1981.

[18] G.M. Bierman. Towards a classical linear λ-calculus (preliminary report). *Electronic Notes in Theoretical Computer Science*, 3, 1996.

[19] O. Bittel. Tableau-based theorem proving and synthesis of λ-terms in the intuitionistic logic. In *European Workshop JELIA '92, LNAI 633*, pages 262–278, 1992.

[20] R. Bornat and B. Sufrin. Animating formal proof at the surface: the Jape proof calculator. To appear: *The Computer Journal*.

[21] E. Boudinet and D. Galmiche. Proofs, concurrent objects and computations in a FILL framework. In *Workshop on Object-based Parallel and Distributed Computation, OBPDC'95, LNCS 1107*, pages 148–167, Tokyo, Japan, 1996.

[22] J. Brown and L. Wallen. Representing unification in a logical framework. September 1995.

[23] A. Bundy. *The Computer Modelling of Mathematical Reasoning*. Academic Press, 1983.

[24] A. Bundy. The use of explicit plans to guide proofs. In E. Lusk and R. Overbeek, editors, *LNCS 310: Proc. CADE-9*, pages 111–120. Springer, 1988.

[25] A. Bundy. A science of reasoning. In J.-L. Lassez and G.D. Plotkin, editors, *Computational Logic: Essays in Honor of Alan Robinson*, pages 178–198. MIT Press, 1991.

[26] A. Bundy, F. van Harmelen, C.Horn, and A. Smaill. The use of explicit plans to guide proofs. In M. Stickel, editor, *LNCS 449: Proc. CADE-10*, pages 647–648. Springer, 1990.

[27] F. Cantu, A. Bundy, A. Smaill, and D. Basin. Experiments in automating hardware verification using inductive proof planning. In M. Srivas and A Camilleri, editors, *LNCS 1166*, pages 94–108. Springer, 1996.

[28] J. Cartmell. Generalised algebraic theories and contextual categories. *Annals of Pure and Applied Logic*, 32:209–243, 1994.

[29] I Cervesato. *A Linear Logical Framework*. PhD thesis, Università di Torino, 1996. (In Italian.).

[30] I. Cervesato and F. Pfenning. Linear higher-order pre-unification. In *CADE Workshop on Proof-search in Type-theoretic Languages*, pages 41–48, Rutgers University, New Brunswick, USA, 1996.

[31] I. Cervesato and F. Pfenning. A linear logical framework. In *11th IEEE Symposium on Logic in Computer Science*, pages 264–275, New Brunswick, New Jersey, July 1996.

[32] B. Chellas. *Modal Logic.* Cambridge University Press, 1980.

[33] J. Chirimar. *Proof Theoretic Approach to Specification Languages.* PhD thesis, Department of Computer and Information Science, University of Pennsylvania, 1995.

[34] T. Coquand. *On the analogy between propositions and types*, pages 399–417. The UT Year of Programming series: Logical Foundations of Functional Programming. Addison-Wesley, 1990.

[35] T. Coquand and G. Huet. The Calculus of Constructions. *Information and Computation*, 76:95–120, 1988.

[36] T. Coquand, B. Nordström, J. Smith, and B. von Sydo. Type Theory and Programming. *Bulletin of the EATCS*, 52:203–228, 1994.

[37] M. D'Agostino and D.M. Gabbay. A Generalization of Analytic Deduction via Labelled Deductive Systems. Part I: Basic substructural logics. *Journal of Automated Reasoning*, 13:243–281, 1994.

[38] B. J. Day. On closed categories of functors. In S. Mac Lane, editor, *Reports of the Midwest Category Seminar*, volume 137 of *Lecture Notes in Mathematics*, pages 1–38. Springer-Verlag, Berlin-New York, 1970.

[39] G. Delzanno, D. Galmiche, and M. Martelli. A specification logic for concurrent object-oriented programming. *Math. Struct. in Comp. Science*, 1999. To appear.

[40] G. Dowek. A complete proof synthesis method for the cube of type systems. *Journal of Logic and Computation*, 3(3):287–315, June 1993.

[41] M. Dummett. *Elements of Intuitionism.* Oxford University Press, 1977.

[42] R. Dyckhoff. Contraction-free sequent calculi for intuitionistic logic. *Journal of Symbolic Logic*, 57:795–807, 1992.

[43] U. Egly and S. Schmitt. Intuitionistic proof transformations and their application to constructive program synthesis. In D. Galmiche, editor, *CADE Workshop on Proof-search in Type-theoretic Languages*, pages 19–30, Lindau, Germany, 1998.

[44] C.M. Elliot. Higher-order unification with dependent function types. In *3rd International Conference RTA 89, LNCS 355*, pages 121–136, Chapell Hill, North Carolina, April 1989.

[45] C. Elliott. Extensions and applications of higher-order unification. Ph.D. thesis, Carnegie Mellon University, 1990.

[46] U. Engberg and G. Winskel. Completeness results for linear logic on petri nets. *Annals of Pure and Applied Logic*, 86:101–135, 1997.

[47] R.L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, 1986.

[48] A. Felty and D. Miller. Encoding a dependent-type λ-calculus in a logic programming language. In *10th International Conference on Automated Deduction, LNAI 449*, pages 221–235, Kaiserslautern, FRG, July 1990.

[49] J. Gallier. *Logic for Computer Science: Foundations of Automatic Theorem Proving.* John Wiley & Sons, 1986.

[50] D. Galmiche. Program development in constructive type theory. *Theoretical Computer Science*, 94(2):237–259, 1992.

[51] D. Galmiche and O. Hermann. SKIL: A system for programming with proofs. In *LPAR'93, International Conference on Logic Programming and Automated Reasoning, LNAI 698*, pages 348–350, St. Petersburg, Russia, July 1993.

[52] D. Galmiche and D. Larchey-Wendling. Provability in intuitionistic linear logic from a new interpretation on petri nets - extended abstract. *Electronic Notes in Theoretical Computer Science*, 17, 1998.

[53] D. Galmiche and B. Martin. Proof search and proof nets construction in linear logic. In *4th Workshop on Logic, Language, Information and Computation, Wollic'97*, Fortaleza, Brasil, August 1997. Logic Journal of IGPL, vol. 5-6, pp 883-885.

[54] D. Galmiche and B. Martin. Proof nets construction and automated deduction in non-commutative linear logic - extended abstract. *Electronic Notes in Theoretical Computer Science*, 17, 1998.

[55] D. Galmiche and G. Perrier. A procedure for automatic proof nets construction. In *LPAR'92, International Conference on Logic Programming and Automated Reasoning, LNAI 624*, pages 42–53, St. Petersburg, Russia, July 1992.

[56] D. Galmiche and G. Perrier. Foundations of proof search strategies design in linear logic. In *Logic at St Petersburg '94, Symposium on Logical Foundations of Computer Science, LNCS 813*, pages 101–113, St Petersburg, Russia, July 1994.

[57] D. Galmiche and G. Perrier. On proof normalization in linear logic. *Theoretical Computer Science*, 135(1):67–110, 1994.

[58] G. Gentzen. Untersuchungen über das logische Schliessen. *Mathematische Zeitschrift*, 39:176–210, 405–431, 1934.

[59] D. Gillies. *Artificial Intelligence and Scientific Method.* Oxford University Press, 1997.

[60] J.Y. Girard. The system F of variables types, fifteen years later. *Theoretical Computer Science*, 45:159–192, 1986.

[61] J.Y. Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–102, 1987.

[62] J.Y. Girard. Linear logic: its syntax and semantics. In J.Y. Girard, Y. Lafont, and L. Regnier, editors, *Advances in Linear Logic*, pages 1–42. Cambridge University Press, 1995.

[63] J.Y. Girard. Proof nets: the parallel syntax for proof theory. In Ursini and Agliano, editors, *Logic and Algebra*, New York, 1995. M. Dekker.

[64] J.Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types.* Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1989.

[65] M.J. Gordon, R. Milner, and C.P. Wadsworth. *Edinburgh LCF.* LNCS 78, Springer Verlag, 1979.

[66] M.J.C Gordon. Hol : A proof generating system for higher-order logic. In G. Birtwistle and P.A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 73–128, 1988.

[67] J. Harland. A proof-theoretic analysis of goal-directed provability. *Journal of Logic and Computation*, 4(1):69–88, 1994.

[68] J. Harland and D. Pym. Resource-distribution via boolean constraints (extended abstract). In *14th Int. Conference on Automated Deduction, CADE-14, LNAI 1249*, pages 222–236, Townsville, North Queensland, Australia, July 1997.

[69] J.A. Harland, D.J. Pym, and M. Winikoff. Programming in Lygon: an overview. In M. Wirsing and M. Nivat, editors, *Proc. AMAST '96*, pages 391–405. LNCS 1101, 1996.

[70] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of ACM*, 40:143–184, 1993.

[71] R. Harper, D. Sannella, and A. Tarlecki. Structured theory presentations and logic representations. *Annals of Pure and Applied Logic*, 67:113, 1994.

[72] Robert W. Harper, Furio Honsell, and Gordon D. Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, January 1993.

[73] L. Helmink. Resolution and type theory. In *ESOP 90, LNCS 432*, pages 197–211, Copenhagen, Denmark, May 1990.

[74] J. Hodas and D. Miller. Logic programming in a fragment of intuitionistic linear logic. *Journal of Information and Computation*, 110:327–365, 1994.

[75] J. Hodas and J. Polakow. Forum as a Logic Programming Language (Preliminary Report). *Electronic Notes in Theoretical Computer Science*, 3, 1996.

[76] JS Hodas and D Miller. Logic programming in a fragment of intuitionistic linear logic. *Information and Computation*, 110(2):327–365, 1994.

[77] C.J. Hogger. *Essentials of Logic Programming.* Clarendon Pres, Oxford, 1990.

[78] W.A. Howard. The formulae-as-types notion of construction. In *To H.B. Curry : Essays in Combinatory Logic, λ-calculus and Formalism*, pages 479–490. Academic Press, 1980.

[79] D.J. Howe. Sematic foundations for embedding HOL in Nuprl. In *5th Int. Conf. on Algebraic Methods and Software Technology, AMAST'96, LNCS 1101*, pages 85–101, Munich, Germany, July 1996.

[80] G. Huet. Résolution d'équations dans les languages d'ordre 1, 2 , ... , ω. Thèse de Doctorat d'État, Université de Paris VII, 1976.

[81] G. Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1:27–57, 1975.

[82] G. Huet. *A uniform approach to type theory*, chapter 16, pages 337–398. The UT Year of Programming Series. Addison-Wesley, 1990.

[83] A. Ireland. The use of planning critics in mechanizing inductive proofs. In A. Voronkov, editor, *LNCS 624: Proc. LPAR 92*, pages 178–189. Springer, 1992.

[84] A. Ireland and A. Bundy. Productive use of failure in inductive proof. *Journal of Automated Reasoning*, 16(1-2):79–111, 1996.

[85] S.S. Ishtiaq and D.J. Pym. Kripke resource models of a dependently-typed, bunched λ-calculus. In Preparation.

[86] S.S. Ishtiaq and D.J. Pym. A relevant analysis of natural deduction. *Journal of Logic and Computation*, 8(6):809–838, 1998.

[87] B. Jacobs. *Categorical Type Theory.* PhD thesis, Catholic University, Nijmegen, September 1991.

[88] N. Kobayashi and A. Yonezawa. ACL - a concurrent linear logic programming paradigm. In *Int. Symposium on Logic Programming*, pages 279–294, Vancouver, October 1993.

[89] N. Kobayashi and A. Yonezawa. Asynchronous communication model based on linear logic. *Formal Aspects of Computing*, 3:279–294, 1994.

[90] N. Kobayashi and A. Yonezawa. Type-theoretic foundations for concurrent object-oriented programming. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA'94*, pages 31–45, 1994.

[91] R. Kowalski. *Logic for Problem Solving.* Elsevier, 1979.

[92] C. Kreitz, H. Mantel, J. Otten, and S. Schmitt. Connection-based proof construction in linear logic. In *14th Int. Conference on Automated Deduction*, pages 207–221, Townsville, North Queensland, Australia, 1997.

[93] J. Lambek and P.J. Scott. *Introduction to higher order categorical logic*, volume 7 of *Cambridge studies in advanced mathematics.* Cambridge University Press, 1987.

[94] P. Lincoln and N. Shankar. Proof search in first-order linear logic and other cut-free sequent calculi. In *9th IEEE Symposium on Logic in Computer Science*, pages 282–291, Paris, France, July 1994.

[95] P.D. Lincoln, J.C. Mitchell, and A. Scedrov. Stochastic interaction and linear logic. In J.Y. Girard, Y. Lafont, and L. Regnier, editors, *Advances in Linear Logic*, pages 147–166. Cambridge University Press, 1995.

[96] P.D. Lincoln, J.C. Mitchell, and A. Scedrov. Linear logic proof games and optimization. *Bulletin of Symbolic Logic*, 2(3), September 1996.

[97] H. Lowe and D. Duncan. Xbarnacle: Making theorem provers more accessible. In W. McCune, editor, *14th Conference on Automated Deduction, LNCS 1249*, pages 404–408. Springer, 1997.

[98] Z. Luo and R. Pollack. *LEGO Proof Development System: User's Manual.* LFCS Technical report ECS-LFCS-92-211, Edinburgh, 1992.

[99] P. Madden. Automatic program optimization through proof transformation. In *11th Conference on Automated DEduction, LNAI 607*, pages 446–460, Saratoga Springs, June 1992.

[100] L. Magnussen and B. Nordström. The ALF proof editor and its proof engine. In *Workshop on Types for Proofs and Programs, LNCS 806*, pages 213–237, 1993.

[101] P. Manoury, M. Parigot, and M. Simonot. Propre: A programming language with proofs. In *International Conference on Logic Programming and Automated Reasoning, LNAI 624*, pages 484–486, St. Petersburg, Russia, July 1992.

[102] P. Martin-Löf. An intuitionnistic theory of types. In *Logic Colloquium '73*, pages 73–118. North-Holland, 1973.

[103] P. Martin-Löf. Constructive mathematics and computer programming. In *6th Congress for Logic, Methodology and Philosophy of Science*, pages 153–175. North-Holland, 1982.

[104] P. Martin-Löf. *Intuitionistic Type Theory.* Studies in Proof Theory, Lectures Notes. Bibliopolis, 1984.

[105] M. Masseron, C. Tollu, and J. Vauzeilles. Generating plans in linear logic I: Actions as proofs. *Theoretical Computer Science*, 113(2):349–371, 1993.

[106] J. Meseguer and N. Marti-Oliet. From Petri nets to Linear Logic. *Math. Struct. in Comp. Science*, 1:69–101, 1991.

[107] D. Miller. A logical analysis of modules in logic programming. *Journal of Logic Programming*, 6(2):79–108, 1989.

[108] D. Miller. Forum: A multiple-conclusion specification logic. *Theoretical Computer Science*, 165(1):201–232, 1996.

[109] D. Miller, G. Nadathur, F. Pfenning, and A. Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

[110] G. Mints. Gentzen type systems and resolution rules. In P. Martin-Löf and G. Mints, editors, *COLOG 88, LNCS 417*, pages 198–231, Tallinn, Estonia, 1988.

[111] G. Mints. Resolution calculus for the first order linear logic. *Journal of Logic, Language and Information*, 2:59–83, 1993.

[112] J. Mitchell and E. Moggi. Kripke-style models for typed lambda calculus. *Annals of Pure and Applied Logic*, 51:99–124, 1981.

[113] G. Nadathur and D. Miller. Higher-order logic programming. *Handbook of Logic in AI and Logic Progamming*, 5, Oxford University Press,.

[114] B. Nordström, K. Petersson, and J. Smith. *Programming in Martin-Löf's Type Theory, An Introduction*, volume 7 of *Monographs on Computer Science.* Oxford Press, 1990.

[115] P.W. O'Hearn. Doubly closed categories, resource interpretations and the $\alpha\lambda$-calculus. To appear: TLCA 99. Available on the web at http://www.dcs.qmw.ac.uk/ ˜ ohearn.

[116] P.W. O'Hearn and D.J. Pym. The logic of bunched implications. To appear: Bulletin of Symbolic Logic, 1999. Available on the web at http://www.dcs.qmw.ac.uk/˜pym.

[117] H. Ono. *Substructural Logics*, chapter Semantics for Substructural Logics, pages 259–291. Oxford University Press, 1993.

[118] S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M.K. Srivas. PVS: Combining specification, proof checking, and model checking. In *Computer-Aided Verification, CAV '96, LNCS 1102*, pages 411–414, New Brunswick, NJ, July/August 1996.

[119] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Language Reference.* Computer Science Laboratory, SRI International, Menlo Park, CA, September 1998.

[120] C. Parant. Synthesizing proofs from programs in the calculus of inductive constructions. In *Mathematics for Program Construction, MPC'95, LNCS 947*, pages 351–379, Kloster Irsee, Germany, July 1995.

[121] R. Paré and D. Schumacher. *Abstract families and the adjoint functor theorems*, volume 661 of *Lecture Notes in Mathematics*, pages 1–125. Springer-Verlag, Berlin, 1978.

[122] M. Parigot. Programming with proofs : a second order type theory. In *European Symposium On Programming 88, LNCS 300*, pages 145–159, Nancy, France, March 1988. Springer Verlag.

[123] M. Parigot. Free deduction: an analysis of computations in classical logic. In *1th and 2th Russian Conference on Logic Programming, LNAI 592*, pages 361–380, St. Petersburg, Russia, July 1991.

[124] M. Parigot. $\lambda\mu$-calculus: an algorithmic interpretation of classical natural deduction. In *LPAR'92, International Conference on Logic Programming and Automated Reasoning, LNAI 624*, pages 190–201, St. Petersburg, Russia, July 1992.

[125] M. Parigot. Recursive programming with proofs. *Theoretical Computer Science*, 94(2):335–356, March 1992.

[126] M. Parigot. Classical proofs as programs. In *Computational Logic and Proof Theory, KGC'93, LNCS 713*, pages 263–276, Brno, Czech. Rep., August 1993.

[127] C. Paulin-Mohring and B. Werner. Synthesis of ML programs in the system Coq. *Journal of Symbolic Computation*, 15:607–640, 1993.

[128] L.C. Paulson. *Logic and Computation : Interactive Proof with Cambridge LCF.* Cambridge University Press, 1987.

[129] L.C. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5:363–397, 1989.

[130] L.C. Paulson. *Logic and Computer Science*, chapter The next 700 theorem provers. Academic Press, 1990.

[131] L.C. Paulson. *Isabelle: a generic theorem prover, Lecture Notes in Computer Science vol. 828.* Springer Verlag, Berlin, 1994.

[132] F. Pfenning. *Logical Frameworks*, chapter Logic Programming in the LF Logic Framework, pages 149–181. Cambridge University Press, 1991.

[133] F. Pfenning and E. Rohwedder. Implementing the meta-theory of deductive systems. In *11th Int. Conference on Automated Deduction, CADE-11, LNAI 607*, pages 537–551, Saratoga Springs, New York, June 1992.

[134] L. Pinto. Cut formulae and logic programming. In *4th International Workshop on Extensions of Logic Programming, LNCS 798*, pages 282–300, St Andrews, U.K., April 1993.

[135] L. Pinto and R. Dyckhoff. A constructive type system to integrate logic and functional programming. In D. Galmiche and L. Wallen, editors, *CADE Workshop on Proof-search in Type-theoretic Languages*, pages 70–81, Nancy, France, 1994.

[136] L. Pinto and R. Dyckhoff. Loop-free construction of counter-models for intuitionistic propositional logic. In Behara and al., editors, *Symposia Gaussiana*, pages 225–232, 1995.

[137] L. Pinto and R. Dyckhoff. Sequent calculi for the normal terms of the $\lambda\Pi$- and $\lambda\Pi\Sigma$-calculi. In D. Galmiche, editor, *CADE Workshop on Proof-search in Type-theoretic Languages*, pages 93–106, Lindau, Germany, 1998.

[138] A. Pitts. Categorical logic. In S. Abramsky, D. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science, Volume 6*, pages 264–275. Oxford University Press, 1992.

[139] D. Prawitz. *Natural Deduction: A Proof-Theoretical Study.* Almquist and Wiksell, Stockholm, 1965.

[140] D. Pym. A note [on] the proof theory of the $\lambda\Pi$-calculus. *Studia Logica*, 54:175–207, 1995.

[141] D. Pym. A note on representation and semantics in logical frameworks. In D. Galmiche, editor, *CADE Workshop on Proof-search in Type-theoretic Languages*, pages 101–108, New Brunswick, USA, 1996.

[142] D. Pym and J. Harland. A uniform proof-theoretic investigation of linear logic programming. *Journal of Logic and Computation*, 4(2):175–207, 1994.

[143] D. Pym and L. Wallen. *Logical Frameworks*, chapter Proof-search in the $\lambda\Pi$-calculus, pages 283–294. Cambridge University Press, 1991.

[144] D.J. Pym. Functorial Kripke-Beth-Joyal Models of the $\lambda\Pi$-calculus I: Type Theory and Internal Logic. 1998. In preparation. Note that [144], [145], [146] may appear within a single publication.

[145] D.J. Pym. Functorial Kripke-Beth-Joyal Models of the $\lambda\Pi$-calculus II: The LF Logical Framework. 1998. In preparation. Note that [144], [145], [146] may appear within a single publication.

[146] D.J. Pym. Functorial Kripke-Beth-Joyal Models of the $\lambda\Pi$-calculus III: Logic Programming and Its Semantics. 1998. In preparation. Note that [144], [145], [146] may appear within a single publication.

[147] D.J. Pym. On bunched predicate logic. To appear: Proc. LICS, 1999, IEEE. Available on the web at http://www.dcs.qmw.ac.uk/˜pym.

[148] D.J. Pym. A relevant analysis of natural deduction. Lecture at EU Types Workshop, Baastad, Sweden.

[149] D.J. Pym. The semantics and proof theory of the logic of bunched implications, I: Propositional **BI**. 1998. Available on the web at http://www.dcs.qmw.ac.uk/ ˜ pym.

[150] D.J. Pym. The semantics and proof theory of the logic of bunched implications, II: Predicate **BI**. 1998. In Preparation.

[151] D.J. Pym. *Proofs, Search and Computation in General Logic.* Ph.D. thesis, University of Edinburgh, 1990. See also: Errata and Remarks, ECS-LFCS-93-265, University of Edinburgh, 1993.

[152] D.J. Pym. A unification algorithm for the $\lambda\Pi$-calculus. *International J. of Foundations of Computer Science*, 3(2):333–378, 1992.

[153] D.J. Pym. Logic programming with bunched implications (extended abstract). *Electronic Notes in Theoretical Computer Science*, 17, 1998.

[154] E. Ritter D.J. Pym and L.A. Wallen. On the intuitionistic force of classical search. In *Proc. Tableaux 96, LNCS 1071*, pages 295–31, 1996.

[155] E. Ritter, D.J. Pym, and L.A. Wallen. Proof-terms for classical and intuitionistic resolution. In *13rd Conference on Automated Deduction, LNCS 1104*, pages 17–31, 1996.

[156] J.A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12:23–41, 1965.

[157] E. Rohwedder. Proof-search in the meta-theory of deductive systems. In D. Galmiche and L. Wallen, editors, *CADE Workshop on Proof-search in Type-theoretic Languages*, pages 82–86, Nancy, France, 1994.

[158] D.A. Schmidt. *Denotational Semantics.* Allyn and Bacon, 1986.

[159] S. Schmitt and C. Kreitz. On transforming intuitionistic matrix proofs into standard sequent proofs. In *4th Workshop on Theorem Proving with Analytic Tableaux and Related Methods, LNAI 918*, pages 106–121, St Goar am Rhein, Germany, 1995. Springer Verlag.

[160] R. A. G. Seely. Hyperdoctrines, natural deduction and the Beck condition. *Zeitschr. für Math. Logik und Grundlagen der Math.*, 29:505–542, 1983.

[161] J. Smith. An interpretation of Martin-Löf's type theory in a type fre theory of propositions. *Journal of Symbolic Logic*, 49(3):730–753, 1984.

[162] A. Stoughton. Porgi: a proof-or-refutation generator for intuitionistic propositional logic. In D. Galmiche, editor, *CADE Workshop on Proof-search in Type-theoretic Languages*, pages 109–116, New Brunswick, USA, 1996.

[163] T. Tammet. Proof strategies in linear logic. *Journal of Automated Reasoning*, 12:273–304, 1994.

[164] T. Tammet. A resolution theorem prover for intuitionistic logic. In *13th Int. Conference on Automated Deduction, CADE-13, LNAI1104*, pages 2–16, New Brunswick, NJ, USA, 1996.

[165] A. Tarski. *Logic, Semantics, Metamathematics.* Clarendon Press, Oxford, 1956.

[166] A. Troelstra. *Lectures on Linear Logic.* CSLI, 1992.

[167] T.Streicher. Correctness and completeness of a categorical semantics of the calculus of constructions. PhD thesis, Universität Passau, 1988.

[168] A. Voronkov. Theorem proving in non-standard logics based on the inverse method. In *11th Conference on Automated DEduction, LNAI 607*, pages 648–662, Saratoga Springs, June 1992.

[169] A. Voronkov. Proof-search in intuitionistic logic based on constraint satisfaction. In *5th Int. Workshop TABLEAUX'96, LNAI 1071*, pages 312–327, Terrasini, Palermo, Italy, May 1996.

[170] L.A. Wallen. *Automated Proof Search in Non-Classical Logics.* MIT Press, 1990.

[171] M. Winikoff and J. Harland. Implementing the linear logic programming language Lygon. *Proc. ILPS '95 (J. Lloyd, editor)*, page 66, MIT Press, 1995.

[172] D.N. Yetter. Quantales and (noncommutative) linear logic. *Journal of Symbolic Logic*, 55(1):41–64, 1990.