To appear in: H.-D. Ebbinghaus et al. (eds.) Proc. Logic Colloquium'87, ©North-Holland, 1989.

General Logics*

José Meseguer
SRI International, Menlo Park, CA 94025, and
Center for the Study of Language and Information
Stanford University, Stanford, CA 94305

Dedicated with affection to my father, Francisco Meseguer, who died unexpectedly a few weeks after the Granada Logic Colloquium

1 Introduction

The connections between logic and computer science are growing rapidly and are becoming deeper. Besides theorem proving, logic programming, and program specification and verification, other areas showing a fascinating mutual interaction with logic include type theory, concurrency, artificial intelligence, complexity theory, databases, operational semantics, and compiler techniques. The concepts presented in this paper are motivated by the need to understand and relate the many different logics currently being used in computer science, and by the related need for new approaches to the rigorous design of computer systems. Logic programming is of course one of the areas where logic and computer science interact most strongly. The attempt to better understand the nature of this interaction, as well as its future prospects, motivates the following basic question:

What is Logic Programming?

This paper tries to make precise the meaning of this question, and to answer it in terms of general axioms which apply to a wide variety of different logics. In doing so, we are inevitably led to ask the more fundamental question:

What is a Logic?

That is, how should general logics be axiomatized? This is because an axiomatic notion of logic programming must necessarily rest on an axiomatic notion of logic itself. Most

*Supported by Office of Naval Research Contracts N00014-82-C-0333 and N00014-86-C-0450, NSF Grant CCR-8707155 and by a grant from the System Development Foundation.

of the paper will be devoted to the second question. With an axiomatic notion of logic already in place, it will then answer the first.

Beyond their application to logic programming, the axioms for general logics given here are sufficiently general to have wide applicability within logic and computer science. Thus, this work has goals that are in full agreement with those of J.A. Goguen and R. Burstall's theory of institutions [27,26]; however, it addresses proof-theoretic aspects not addressed by institutions. In fact, institutions can be viewed as the model-theoretic component of the present theory. The main new contributions include a general axiomatic theory of *entailment* and *proof*, to cover the proof-theoretic aspects of logic and the many proof-theoretic uses of logic in computer science; they also include new notions of *mappings* that interpret one logic (or proof calculus) in another, an axiomatic study of *categorical logics*, and the axioms for logic programming.

Logic has of course a long tradition of reflecting upon itself. In the process of seeking the present axiomatization, I have reviewed a variety of previous axiomatizations of logic. In essence, they tend to fall within two main approaches:

- a model-theoretic approach that takes the satisfaction relation between models and sentences as basic, and

- a proof-theoretic approach that takes the entailment relation between sets of sentences as basic.

I have found that neither of these approaches is by itself sufficient to axiomatize logic programming, and that similar difficulties remain for many other computer science applications. The axioms presented here unify both approaches, and yield as one of their fruits the desired axioms for logic programming.

The entailment relation asserts provability, but says nothing about how a sentence is actually proved. To account for proofs and for their internal structure, I also present a new axiomatic notion of *proof calculus* that is very general and does not favor any particular proof theory style. The computer science counterpart of a proof calculus is the notion of an "operational semantics." The flexibility of the axioms given for proof calculi permits putting them and operational semantics on a common abstract basis. This offers the possibility of a more intense *mutual* interaction between the two fields. As a fruit of this interaction, operational semantics could be placed on a firmer logical basis and proof theory could be enriched with new, more flexible, proof systems. The value of establishing a closer link between proof theory and operational semantics has been recognized by many authors, and has led to specific proposals such as the one by G. Plotkin [57]; it is also emphasized in the recent work of J.-Y. Girard [20].

The methods of category theory have taught us that the crucial mathematical properties of a given subject do not reside in the *structures* in question, and even less in the particular *representations* chosen for those structures. Rather, they reside in the *mappings* that preserve those structures. In our present case, the structures are logics themselves and the mappings should be natural ways of interpreting one logic, or one proof calculus, into another. Although instances of such interpretations abound in logic, and are of great practical importance in computer science, most existing axiomatic treatments of general logics, with the exception of Goguen and Burstall's theory of institutions [27,26] and work related to institutions, do not include a formal treatment of such mappings. In this paper, mappings of this sort play a central role. New general notions of mapping between entailment systems, between logics, and between proof systems, are given and illustrated with examples. The mappings in question are far more general than syntactic mappings translating languages in one logic into languages in another; they can also map languages into theories, or even perform theory transformations of a more general nature. In addition, they also provide a systematic way of relating models in different logics. Such notions of mapping should have wide applicability in logic. They also have many potential applications in computer science.

Axiomatic treatments such as "abstract model theory" usually come with a built-in notion of *structure*, such as a set-theoretic structure with functions, relations, etc. Although very useful within their own scope, such approaches lack the flexibility needed to deal with logics such as nonclassical and higher order logics, for which the appropriate notion of "model" may be widely different from traditional set-theoretic structures. For many of these nonstandard logics the categorical approach initiated by F.W. Lawvere [45,46] is ideally suited. In particular, there is at present a thriving interest in categorical logics in connection with applications of constructive type theory to programming languages and to concurrent systems. To demonstrate the flexibility of the axioms proposed in this paper, I give a general definition of *categorical logic*, discuss examples, study their main properties, and show how any such logic is a particular instance of a logic in the sense proposed here. I reexamine this topic in connection with the axioms for logic programming, in order to show how in this way functional programming languages based on type theory can be naturally unified with logic programming.

The main purpose of this paper is to propose new concepts that I believe can be useful in many areas of computer science and also in logic. Given this purpose, there are more definitions and examples than there are theorems; this situation should of course be reversed in the future. Indeed, the paper as a whole is in a sense more a promise of things to come than an actual fulfillment of that promise. The concept of a logic as a harmonious relationship between entailment and satisfaction is particularly simple and, once the observation is made, seems the obvious thing to do. The concept of a proof calculus is less obvious and is perhaps one of the main new contributions; I believe that for computer science applications one needs to have great flexibility about what counts as a "proof;" therefore, I have avoided making any commitments to particular proof theory styles in the proof calculus axioms. Many future computer science applications of these ideas will heavily use *mappings* between logics, between proof calculi, etc. I believe that mappings may prove to be the most important concept. They play a key role in relating different logics and different computer systems, and such relationships are conceptually and practically very important. Also, experience with them may suggest new methods for the rigorous design and development of computer systems. Two topics with more particular scopes but still quite important are categorical logics —specially in their applications to type theory and to concurrency— and the axioms for logic programming, because of the new possibilities for language design that they suggest.

The paper assumes an acquaintance with elementary concepts of category theory, such as: category, subcategory, functor, natural transformation and natural equivalence, horizontal and vertical composition of natural transformations, pullbacks and pushouts, and adjoint functors. All the relevant concepts are clearly explained in Mac

Lane's book [47].

1.1 General Logics

As already mentioned, one can speak of two approaches to the axiomatization of general logics, a model-theoretic approach that focuses on the satisfaction relation

$$M \models \varphi$$

between a model M and a sentence φ, and a proof-theoretic approach that seeks to axiomatize the entailment relation

$$\Gamma \vdash \varphi$$

between a set of sentences Γ and a sentence φ derivable from Γ. The model-theoretic approach is exemplified by Barwise's axioms for abstract model theory [3] (see also [16] and the volume [2] for a good survey of this field). The framework of institutions of Goguen and Burstall [27,26] also belongs to this model-theoretic approach, but it achieves much greater generality by using category theory and avoiding a commitment to particular notions of "language" and "structure." The proof-theoretic[1] approach has a long tradition dating back to work of Tarski [69] on "consequence relations" and of Hertz and Gentzen on the entailment relation $\vdash$. This approach owes much to the work of Dana Scott [62] and other authors. The recent work of Fiadeiro and Sernades [17] belongs to this proof-theoretic tradition, but uses methods from the theory of institutions; however, that work does not propose a common axiomatization connecting model theory and proof theory. In my attempt to axiomatize logic programming languages, I have found that neither of these two approaches is enough by itself to yield a satisfactory axiomatization. This paper proposes a unified approach that integrates model-theoretic and proof-theoretic aspects into a single axiomatization of a logic. The axiomatization in question is quite simple. It consists of an "entailment system", specifying an entailment relation $\vdash$, together with a "satisfaction system" (specifically, an institution in the Goguen-Burstall sense) specifying a satisfaction relation $\models$. The entailment and satisfaction relations are then linked by a soundness axiom. Therefore, institutions provide the model-theoretic component of a logic in the precise sense given to this term in this paper. As done in the theory of institutions, the axioms for a logic are expressed in a categorical way, to avoid building in particular choices of structures or languages. As a consequence, they are very general.

1.2 Proof Calculi

The entailment relation $\vdash$ says nothing about the internal structure of a proof. To have a satisfactory account of proofs, we need the additional concept of a "proof calculus" $\mathcal{P}$ for a logic $\mathcal{L}$. The definition of a proof calculus is very general, and does not favor any particular style of proof theory. The *same* logic may of course have many different proof calculi. Therefore, when wishing to include a specific proof calculus as part of a logic, the resulting logic plus proof calculus is instead called a *logical system*. The axioms for a proof calculus $\mathcal{P}$ state that each language (also called a *signature*) in the logic $\mathcal{L}$ has an associated space of proofs, which is an object of an appropriate

[1]This is an oversimplification, since semantic considerations are also included in it.

category. From such a space we can then extract the actual set of proofs supporting a given entailment $\Gamma \vdash \varphi$.

We actually need the more general concept of a "proof subcalculus" where proofs are restricted to some given class of axioms and conclusions are also restricted to some given class of sentences. It is by systematically exploiting such restrictions that the structure of proofs can be simplified; for example, we can in this way arrive at proof theories that are efficient and can be used in practice as the *operational semantics* of a logic programming language. For programming and other purposes we need not just proof subcalculi, but *effective* ones.

1.3 Relating Logics

Interpretations mapping one logic, or one proof theory, into another are very common in logic. They permit the transfer of results between different logics and serve to provide a relative foundation for a logical system by reducing it to a simpler one. In computer science there are compelling practical reasons for establishing similar mappings. Even if not explicitly recognized as such, the need for mappings of this sort manifests itself in a variety of ways. For example, for programming languages it may take the form of trying to combine language features from different languages based on different logics, or of trying to compile an inefficient operational semantics into an efficient one. Since the need is there, like a river that nobody can stop, *something* of the sort will be done. However, if the approach taken lacks a logical basis to serve as a criterion for correctness the result may be quite *ad hoc* and unsatisfactory, it will probably involve a good deal of costly engineering trial and error, and may in the end lack a clear intellectual content, making it impossible to be transmitted as a lasting scientific contribution. Also, since the development of computer systems is a very expensive and labor intensive activity, there is a great need for reusing systems as much as possible. One may for example want to use a theorem prover in one logic to prove theorems in a different logic, and the soundness of such a procedure must then be justified by an appropriate map of logics.

This work proposes a general axiomatic theory of mappings that is flexible enough to encompass and unify the maps used in logic and the maps needed in computer science. The most promising computer science application of this theory is probably not to the *a posteriori* justification of computer science practice, but to the development of new methods for the rigorous design of computer systems. For the model-theoretic component, the maps are similar to the institution morphisms of Goguen and Burstall [27,26], which provided the original inspiration. In addition, the following other types of mappings are defined:

- Maps of entailment systems, relating languages in different logics and preserving the entailment relation.

- Maps of proof calculi, which in addition relate proofs in different logics.

- Maps of logics, that map their underlying entailment systems and their underlying institutions.

- Maps of logical systems, that map both logics and their corresponding proof calculi.

As already mentioned, these mappings permit very general translations. They are not restricted to translating one language into another, but allow complex transformations between theories in different logics that also occur in practice.

1.4 Categorical Logics

Axiomatizing a notion always presents a dilemma. In trying to achieve generality, an axiomatization can become too weak and somewhat vacuous in its results. Yet, if the axiomatization is too specific, it will fail to include relevant examples and, furthermore, it may hinder subsequent developments. The goal of Section 5 is to show by example that the proposed notion of logic has, indeed, a wide applicability. I axiomatize a large and important class of logics, namely categorical logics, as particular instances of the general notion of logic proposed in this paper, show that they have remarkably nice model-theoretic properties, and prove a general structure-semantics adjointness theorem.

The basic insight provided by categorical logic in the sense of Lawvere is that the essential aspects of a logic, independently of particular syntactic choices that must be made to talk about it, are categorical properties. For example, the key categorical property behind the linguistic notion of an equation is the existence of finite products. Semantics can then be made functorial, by reinterpreting theories as categories with appropriate structure and viewing models as functors that preserve that structure.

1.5 Axioms for Logic Programming

The logic programing dream has only begun to be realized. So far, the overwhelming majority of work in this area has dealt with the Horn clause fragment of first order logic but the idea is obviously much broader, and its potential in other contexts remains to a good extent unexploited. This paper tries to contribute to a scientific discussion of logic programming concepts in as broad a context as possible. It does so by using the concepts proposed for general logics as the basis for an axiomatic notion of logic programming. The axioms proposed are a further step in a series of attempts to make precise a broad view of logic programming shared with Joseph Goguen. This view was expressed informally in the joint paper [32], and formally by Joseph Goguen using the notion of an institution [24]. Logic programming presents a very interesting combination of proof theory and model theory; the axioms that I propose try to cover and relate both aspects. I also include a discussion of different logic programming languages and styles. Within first order logic there are already several possibilities that are quite different from the traditional Prolog style. A direct fruit of the study of categorical logics undertaken in Section 5 is the establishment of a precise connection between the very active research area of functional programming languages based on constructive type theory and higher order instances of logic programming.

1.6 Acknowledgements

I wish to thank the organizers of the Logic Colloquium'87 for having given me the opportunity of presenting these ideas in the magic city of Granada and for having made the meeting such a wonderful experience. I specially thank Hainz-Dieter Ebbinghaus and José F. Prida for their kindness, and for their patience as editors with my delays.

Regarding my ideas on general logics, I am particularly indebted to the pioneering work of Joseph Goguen and Rod Burstall on institutions, and indeed the reader will find this paper much in the same spirit as theirs. I have been fortunate to learn from them many ideas about logic, and to be present, always as an enthusiastic observer and often as a commentator, at the moment when those ideas were developed.

The axiomatization of logic programming presented in Section 6 articulates a broad view of logic programming that I have shared with Joseph Goguen over the past nine years, and has its roots in our joint work on the OBJ, Eqlog, FOOPS and FOOPlog languages, as well as in previous attempts, some joint, some by Goguen himself, to make this view precise.

I also wish to express my gratitude to Joseph Goguen and Rod Burstall in a special way for their kind encouragement and their important technical suggestions along the maturing process that these ideas have undergone since they were first presented at the Logic Colloquium in Granada. Joseph Goguen provided also very helpful comments to the final draft.

It is a pleasure for me to express my warmest thanks to F. William Lawvere with whom I have discussed the ideas of this paper in many conversations; beyond his many important suggestions, I owe to him a crucial influence in my ways of understanding logic and mathematics that goes back to my years as a graduate student and has been a permanent inspiration ever since.

An earlier version benefited from very valuable suggestions by Jon Barwise, Keith Clark, Brian Mayoh, Gordon Plotkin, Johan van Benthem and an anonymous referee that have led to substantial improvements and I am indebted to them for their kind criticism. Narciso Martí-Oliet deserves special thanks for having suggested numerous improvements to several drafts and for his generous help with the preparation of the LaTeX diagrams.

2 General Logics

The proof-theoretic and model-theoretic approaches to general logics are unified in the following way. First, I introduce the notion of an *entailment system* that formalizes the provability relation; the axioms are similar to those of Scott [62], but they also account for translations across different signatures. Given an entailment system, we can define the notions of a *theory* and a *theory morphism.* The model-theoretic aspects of a logic are covered by Goguen and Burstall's notion of an *institution* [27,26]. The notion of a *logic* is then obtained by combining an entailment system with an institution in such a way that a *soundness* condition (relating provability and satisfaction) holds. The logic is called *complete* if, in addition, satisfaction implies provability. I also show how a complete logic can naturally be associated with an entailment system or with an institution.

2.1 Entailment Systems

Consider the familiar example of first order logic. We usually fix a particular *signature* Σ, consisting of a pair (F, P) where F is a ranked alphabet of function symbols and P a ranked alphabet of predicate symbols. Relative to such a signature, we have a notion of a *sentence*. The rules of first order logic then specify for us an *entailment relation*,

$$\Gamma \vdash \varphi$$

which holds between a set of sentences Γ and a sentence φ when φ is derivable from Γ. For the moment, we shall make abstraction of the particular rules used to generate the relation $\vdash$ and concentrate on the relation itself. Indeed, the entailment relation plays a more central role, since it remains the same across the many different proof calculi that exist for first order logic.

The first order entailment relation $\vdash$ satisfies three basic properties that can more generally be justified on intuitive grounds as properties that any reasonable entailment relation should indeed satisfy. These are:

- *reflexivity*, i.e., we can always prove a sentence if we can assume it;
- *monotonicity*, i.e., we can always prove with more assumptions what we can prove with fewer, and
- *transitivity*, i.e., using as an additional assumption something already proved should not give us more conclusions than those already entailed by our original assumptions.

We may at times wish to change the signature Σ and move to a new signature Σ'. The signature Σ' may just be an enlargement of our original syntax. In other cases, Σ' may be a genuinely different signature so that a translation of the old symbols to the new ones must take place. In any event, the general form of such translations $H : \Sigma \longrightarrow \Sigma'$, called *signature morphisms*, consists, for first order signatures, of a pair of rank-preserving functions, one for function symbols and another for predicate symbols, both used for mapping the old symbols into the new ones. Therefore, signatures form a category $\underline{Sign}$ with signatures as objects and signature morphisms as morphisms. Any signature morphism H induces a corresponding translation at the level of sentences, so that we can associate to each Σ-sentence φ a corresponding Σ'-sentence $H(\varphi)$ by systematically replacing the old symbols by the new ones, according to the signature morphism H. All this can be expressed in a compact way by saying that the process associating to each signature Σ its set of sentences $sen(\Sigma)$ is in fact a *functor* $sen : \underline{Sign} \longrightarrow \underline{Set}$ from the category of signatures to the category of sets. We of course have no problem accepting the fact that entailments are *stable under translation* by signature morphisms: i.e., if we were able to prove a conclusion from some axioms, we are then able, for any translation H, to prove the translated conclusion from the translated axioms.

Definition 1 An *entailment system* is a triple $\mathcal{E} = (\underline{Sign}, sen, \vdash)$ with $\underline{Sign}$ a category whose objects are called *signatures*, sen a functor[2] $sen : \underline{Sign} \longrightarrow \underline{Set}$ and $\vdash$ a function

[2]By convention, the function $sen(H)$ associated by the functor sen to a signature morphism H will also be denoted by H.

associating to each Σ in $\underline{Sign}$ a binary relation $\vdash_\Sigma \subseteq \mathcal{P}(sen(\Sigma)) \times sen(\Sigma)$ called Σ-*entailment* such that the following properties are satisfied:

1. *reflexivity*: for any $\varphi \in sen(\Sigma)$, $\{\varphi\} \vdash_\Sigma \varphi$;
2. *monotonicity*: if $\Gamma \vdash_\Sigma \varphi$ and $\Gamma' \supseteq \Gamma$ then $\Gamma' \vdash_\Sigma \varphi$;
3. *transitivity*: if $\Gamma \vdash_\Sigma \varphi_i$, for $i \in I$, and $\Gamma \cup \{\varphi_i \mid i \in I\} \vdash_\Sigma \psi$, then $\Gamma \vdash_\Sigma \psi$;
4. $\vdash$-*translation*: if $\Gamma \vdash_\Sigma \varphi$, then for any $H : \Sigma \longrightarrow \Sigma'$ in $\underline{Sign}$, $H(\Gamma) \vdash_{\Sigma'} H(\varphi)$.

In addition, an entailment system is called *compact* if whenever $\Gamma \vdash_\Sigma \varphi$, then we can find a finite $\Gamma_0 \subseteq \Gamma$ such that $\Gamma_0 \vdash_\Sigma \varphi$. $\square$

Remarks:

1. The reflexivity, monotonicity and transitivity axioms are similar to those given by Scott in [62]. However, the present formulation avoids any compactness assumptions.

2. The entailment relation $\vdash$ induces a function mapping each set of sentences Γ to the set $\Gamma^\bullet = \{\varphi \mid \Gamma \vdash \varphi\}$. We call $\Gamma^\bullet$ the set of *theorems* provable from Γ. It follows easily from the reflexivity, monotonicity and transitivity axioms that the assignment $\Gamma \mapsto \Gamma^\bullet$ is a *closure* function, i.e., it satisfies:

   (a) $\Gamma \subseteq \Gamma^\bullet$

   (b) $\Gamma \subseteq \Gamma' \Rightarrow \Gamma^\bullet \subseteq \Gamma'^\bullet$

   (c) $\Gamma^{\bullet\bullet} = \Gamma^\bullet$

   Tarski's original axioms [69] were given in terms of this closure.

3. I am indebted to Joseph Goguen for pointing out to me that the recent work of Fiadeiro and Sernades on "π-institutions" [17] proposes a notion quite similar to entailment systems. Their notion is expressed in terms of a closure operator and includes a compactness assumption.

4. The $\vdash$-translation axiom can be equivalently expressed by saying that $\vdash$ is a functor $\vdash : \underline{Sign} \longrightarrow \underline{Set}$, that is a subfunctor of the functor mapping each signature Σ to the set $\mathcal{P}(sen(\Sigma)) \times sen(\Sigma)$.

5. One should keep in mind that the entailment relation $\vdash$ is independent of any rules for its generation. Therefore, the reflexivity, monotonicity and transitivity conditions should be viewed as abstract properties of $\vdash$, and should not be confused with particular *rules* of a specific *proof calculus* for generating $\vdash$, so that, say, reflexivity would then be understood as an axiom scheme, monotonicity as a weakening rule, and transitivity as a cut rule. In fact, it may very well be the case that an entailment system satisfying the reflexivity, monotonicity and transitivity conditions is generated by a proof calculus that *rejects* most structural rules and imposes restrictions on cut. For example, in linear logic [22] weakening and contraction are forbidden so that the calculus is in a sense "nonmonotonic." We have the sequent $A \rightarrow A$ as an axiom, but we cannot derive either $A, B \rightarrow A$ or

even $A, A \rightarrow A$ as consequences. The point is that, for Σ a linear logic signature, the elements of $sen(\Sigma)$ should not be identified with *formulas* but with *sequents*. Viewed as a way of generating sequents, i.e., identifying our $\vdash$ with the closure of the horizontal bar relation among linear logic sequents, the entailment of linear logic is indeed reflexive, monotonic and transitive. The idea that sequents are a good choice for a notion of sentence in linear logic is very much in keeping with Girard's intuition of a sequent $A \rightarrow B$ as an *action* from A to B; it is also supported by recent work on categorical models for linear logic [64,14,56].

2.2 Theories

Given a signature Σ, a theory is presented by a set Γ of Σ-sentences called its axioms. We can therefore define a *theory* as a pair $T = (\Sigma, \Gamma)$. For some purposes, one deals not with the original axioms Γ but rather with their closure under entailment $\Gamma^\bullet$, so that it is tempting to call $T = (\Sigma, \Gamma)$ a *presentation* of the theory $T = (\Sigma, \Gamma^\bullet)$. However, the view of theories as presentations allows us to make finer distinctions that are important for both proof-theoretic and computational purposes. We can, for example, distinguish between a sentence that is a basic axiom and another that is a derived theorem. Also, although (Σ, Γ) and $(\Sigma, \Gamma^\bullet)$ are isomorphic in the general category $\underline{Th}$ of theories, they are not isomorphic in a more restricted but useful category of axiom-preserving theory morphisms.

Definition 2 Given an entailment system $\mathcal{E}$, its category $\underline{Th}$ of *theories* has as objects pairs $T = (\Sigma, \Gamma)$ with Σ a signature and $\Gamma \subseteq sen(\Sigma)$ and as morphisms, $H : (\Sigma, \Gamma) \longrightarrow (\Sigma', \Gamma')$, called *theory morphisms*, signature morphisms $H : \Sigma \longrightarrow \Sigma'$ such that if $\varphi \in \Gamma$, then $\Gamma' \vdash_{\Sigma'} H(\varphi)$.

A theory morphism $H : (\Sigma, \Gamma) \longrightarrow (\Sigma', \Gamma')$ is called *axiom-preserving* iff it satisfies the condition that $H(\Gamma) \subseteq \Gamma'$. This defines a subcategory $\underline{Th_0}$ with the same objects as $\underline{Th}$ but with morphisms restricted to be axiom-preserving theory morphisms. Since given an *arbitrary* theory morphism $H : (\Sigma, \Gamma) \longrightarrow (\Sigma', \Gamma')$ the theories (Σ', Γ') and $(\Sigma', \Gamma' \cup H(\Gamma))$ are always isomorphic, the restriction to $\underline{Th_0}$ is not very serious. Notice that the category $\underline{Th_0}$ does not depend at all on the entailment relation $\vdash$, i.e., any other entailment system with identical signatures and sentences will yield the same $\underline{Th_0}$. □

Remarks:

1. Projection to the first component yields a forgetful functor $sign : \underline{Th} \longrightarrow \underline{Sign} : (\Sigma, \Gamma) \mapsto \Sigma$. Associating to each signature Σ the theory $(\Sigma, \emptyset)$ yields a functor $F : \underline{Sign} \longrightarrow \underline{Th}$ left adjoint to $sign$, i.e., such that there is a natural isomorphism $\underline{Th}(F(\Sigma), T) \cong \underline{Sign}(\Sigma, sign(T))$.

2. The category $\underline{Th}$ is equivalent to the full subcategory determined by theories of the form $T = (\Sigma, \Gamma^\bullet)$. Theory morphisms $H : (\Sigma, \Gamma^\bullet) \longrightarrow (\Sigma', \Gamma'^\bullet)$ are always axiom-preserving, i.e., they satisfy: $H(\Gamma^\bullet) \subseteq \Gamma'^\bullet$. This corresponds to the distinction between entailment closed theories and presentations mentioned above.

3. By composing with the forgetful functor $sign : \underline{Th} \longrightarrow \underline{Sign}$, we can extend the functor $sen : \underline{Sign} \longrightarrow \underline{Set}$ to a functor $sen : \underline{Th} \longrightarrow \underline{Set}$, i.e., we define $sen(T) = sen(sign(T))$.

4. The assignment to each theory $T = (\Sigma, \Gamma)$ of the set $\Gamma^{\bullet}$ of its theorems is a functor $thm : \underline{Th} \longrightarrow \underline{Set}$. Indeed, thm is a subfunctor of the functor sen just defined for theories.

5. We can extend the functor $\vdash : \underline{Sign} \longrightarrow \underline{Set}$ to a functor $\vdash: \underline{Th} \longrightarrow \underline{Set}$ by defining $\Delta \vdash_{(\Sigma,\Gamma)} \varphi$ iff $\Delta \cup \Gamma \vdash_{\Sigma} \varphi$. The original $\vdash : \underline{Sign} \longrightarrow \underline{Set}$ is then recovered by composing with the functor $F : \underline{Sign} \longrightarrow \underline{Th}$.

2.3 Institutions

In first order logic, given a sign. ɜ Σ we associate to it a category $\underline{Mod}(\Sigma)$. Its objects, called Σ-*models* (or Σ-structures), consist of a set together with an interpretation of each n-ary function symbol as an n-ary operation and of each n-ary predicate symbol as an n-ary predicate. Its morphisms are functions that preserve the operations and the predicates. Given a Σ-model M and a Σ-sentence φ we have the notion of *satisfaction* of the sentence φ by the model M, written $M \models_{\Sigma} \varphi$.

A signature morphism $H : \Sigma \longrightarrow \Sigma'$ allows us to view a Σ'-model M' as a Σ-model $H^{\flat}(M')$, just by giving to each function symbol f in Σ the interpretation of $H(f)$ in M', and doing the same for predicate symbols. This extends trivially to homomorphisms, so that we have a functor $H^{\flat} : \underline{Mod}(\Sigma') \longrightarrow \underline{Mod}(\Sigma)$. Globally, this means that $\underline{Mod}$ is actually a functor $\underline{Mod} : \underline{Sign}^{op} \longrightarrow \underline{Cat}$, from the dual of the category of signatures (same objects, reversed direction for morphisms) to the category of categories, where we have adopted the notation $\underline{Mod}(H) = H^{\flat}$. It follows easily from the definition of the satisfaction relation that satisfaction is invariant under the process of changing signatures, i.e.,

$$H^{\flat}(M') \models_{\Sigma} \varphi \text{ iff } M' \models_{\Sigma'} H(\varphi).$$

Definition 3 An *institution* [27,26] is a quadruple $\mathcal{I} = (\underline{Sign}, sen, \underline{Mod}, \models)$ with $\underline{Sign}$ a category whose objects are called *signatures*, $sen : \underline{Sign} \longrightarrow \underline{Set}$ a functor associating to each signature a set of *sentences*, $\underline{Mod} : \underline{Sign}^{op} \longrightarrow \underline{Cat}$ a functor associating to each signature a corresponding category of *models*[3], and $\models$ a function associating to each signature Σ a binary relation $\models_{\Sigma} \subseteq |\underline{Mod}(\Sigma)| \times sen(\Sigma)$, called *satisfaction*, where $|\underline{Mod}(\Sigma)|$ denotes the class[4] of all objects in the category $\underline{Mod}(\Sigma)$ in such a way that the following property holds for any $M' \in \underline{Mod}(\Sigma')$, $H : \Sigma \longrightarrow \Sigma'$, $\varphi \in sen(\Sigma)$:

$$\models\text{-}\textit{invariance}\text{: } H^{\flat}(M') \models_{\Sigma} \varphi \text{ iff } M' \models_{\Sigma'} H(\varphi).$$

□

[3] As before, on morphisms we adopt the notation $\underline{Mod}(H) = H^{\flat}$.

[4] We shall not worry about foundational issues here. Let whoever worries take refuge in a Grothendieck universe!

Given a set of Σ-sentences Γ, we define the category $\underline{Mod}(\Sigma, \Gamma)$ as the full subcategory of $\underline{Mod}(\Sigma)$ determined by those models $M \in \underline{Mod}(\Sigma)$ that satisfy all the sentences in Γ, i.e., $M \models_\Sigma \varphi$ for each $\varphi \in \Gamma$. We can define a relation between sets of sentences and sentences, also denoted $\models$, as follows:

$$\Gamma \models_\Sigma \varphi \text{ iff } M \models_\Sigma \varphi \text{ for each } M \in \underline{Mod}(\Sigma, \Gamma).$$

The naturalness of the definition of entailment system given in Section 2.1 is reinforced by the fact that any institution yields an entailment system.

Proposition 4 For $\mathcal{I} = (\underline{Sign}, sen, \underline{Mod}, \models)$ an institution, the triple $\mathcal{I}^+ = (\underline{Sign}, sen, \models)$, with $\models$ now denoting the associated relation between sets of sentences and sentences, is an entailment system. $\square$

Of course, since this entailment system has been defined by entirely model-theoretic methods, we should not in general expect to find an effective proof calculus to generate it. However, for $\mathcal{I}$ the first order logic institution, the completeness theorem assures us that $\mathcal{I}^+$ coincides with the entailment system for first order logic already discussed in Section 2.1; this of course can be generated from a variety of effective proof calculi.

We shall denote by $\underline{Th}_\models$ the category of theories associated to the entailment system $\mathcal{I}^+$. Let $H : (\Sigma, \Gamma) \longrightarrow (\Sigma', \Gamma')$ be a theory morphism in $\underline{Th}_\models$; given $\varphi \in \Gamma$ we have $\Gamma' \models_{\Sigma'} H(\varphi)$ by definition of theory morphisms. Therefore, for any $M' \in \underline{Mod}(\Sigma', \Gamma')$ we have $M' \models_{\Sigma'} H(\varphi)$ which is equivalent to $H^\flat(M') \models_\Sigma \varphi$ and consequently $H^\flat(M') \in \underline{Mod}(\Sigma, \Gamma)$. This shows that the functor $H^\flat : \underline{Mod}(\Sigma') \longrightarrow \underline{Mod}(\Sigma)$ restricts to a functor:

$$H^\flat : \underline{Mod}(\Sigma', \Gamma') \longrightarrow \underline{Mod}(\Sigma, \Gamma).$$

Globally, this means that we can extend our original functor $\underline{Mod} : \underline{Sign}^{op} \longrightarrow \underline{Cat}$ to a functor:

$$\underline{Mod} : \underline{Th}_\models{}^{op} \longrightarrow \underline{Cat}.$$

We shall call an institution $\mathcal{I}$ *liberal* [27,26] if for any theory morphism $H : T \longrightarrow T'$ in $\underline{Th}_\models$ the functor $H^\flat : \underline{Mod}(T') \longrightarrow \underline{Mod}(T)$ always has a left adjoint, denoted H^*, i.e., there is a natural isomorphism $\underline{Mod}(T')(H^*(M), M') \cong \underline{Mod}(T)(M, H^\flat(M'))$. First order logic is not liberal, but (first order) equational logic and Horn logic are. Liberality of an institution is an abstract measure of a logic's algebraic character. Lawvere showed in his thesis [45] that all the usual free constructions of algebra are direct consequences of the general fact that equational logic is a liberal institution. For example, for T the theory of commutative monoids, T' the theory of commutative rings, and $H : T \rightarrow T'$, the theory morphism that interprets the monoid operation as ring multiplication, the functor H^*, left adjoint to $H^\flat$, is the monoid-ring construction that, for free commutative monoids, specializes to the polynomial ring construction.

We say that an institution $\mathcal{I}$ *admits initial models* if for any theory $T \in \underline{Th}_\models$ the category $\underline{Mod}(T)$ has an initial object, denoted I_T. The general definition of initial objects is as follows.

Definition 5 In any category $\mathcal{A}$ an object I is said to be *initial* if for any object X in $\mathcal{A}$ there is a unique morphism $I \longrightarrow X$ in $\mathcal{A}$. $\square$

We shall call an institution I *exact* if the functor $\underline{Mod} : \underline{Th_{\models}}^{op} \longrightarrow \underline{Cat}$ preserves pullback diagrams.

The institution of first order equational logic is exact and admits initial models, which are the relatively free algebras on an empty set of generators. For example, for the theory of rings, the initial algebra is the ring of integers. In Section 5 we shall encounter many other logics whose underlying institutions are liberal, admit initial models and are exact.

2.4 Logics

We are now ready to give axioms that cover both the provability and the model-theoretic sides of a logic. The solution is very simple: a logic has two components consisting of an entailment system and an institution that share the same signatures and sentences. In addition, the logic must be *sound*, i.e., we must have

$$\Gamma \vdash \varphi \Rightarrow \Gamma \models \varphi.$$

For complete logics, such as first order logic, this implication is actually an equivalence.

Definition 6 A *logic* is a 5-tuple $\mathcal{L} = (\underline{Sign}, sen, \underline{Mod}, \vdash, \models)$ such that:

1. $(\underline{Sign}, sen, \vdash)$ is an entailment system;
2. $(\underline{Sign}, sen, \underline{Mod}, \models)$ is an institution, and
3. the following *soundness* condition is satisfied: for any $\Sigma \in \underline{Sign}$, $\Gamma \subseteq sen(\Sigma)$ and $\varphi \in sen(\Sigma)$,
$$\Gamma \vdash_{\Sigma} \varphi \Rightarrow \Gamma \models_{\Sigma} \varphi.$$

A logic is *complete* if, in addition,

$$\Gamma \vdash_{\Sigma} \varphi \Leftrightarrow \Gamma \models_{\Sigma} \varphi.$$

A logic is *compact*[5] if its underlying entailment system is so. Similarly, a logic is *liberal*, *admits initial models* or is *exact* if its underlying institution is so. Given a logic $\mathcal{L}$, its underlying entailment system will be denoted $ent(\mathcal{L})$; similarly, its underlying institution will be denoted $inst(\mathcal{L})$. □

Remarks:

1. In Section 2.3 I presented the original definition of an institution as given by Goguen and Burstall [27,26]. More recently, Goguen and Burstall [28] have added the proof-theoretic requirement that the set $sen(\Sigma)$ is a category, whose morphisms $\varphi \longrightarrow \psi$ are understood as proofs. By postulating additional assumptions, such as compactness and existence of conjunction for sentences, one could associate to proofs in this sense an entailment relation $\Gamma \vdash \varphi$. However, it does not seem possible to treat in this way the general case of an arbitrary institution.

[5]Strictly speaking, for an incomplete logic, compactness can be a property of either $\vdash$, or of $\models$ (i.e., the relation between sets of sentences and sentences induced by satisfaction). Therefore, we could speak of $\vdash$-compactness and $\models$-compactness for a logic. In practice, however, if anything is going to be compact at all, it will probably be $\vdash$.

2. Notice that the inclusion $\vdash_\Sigma \subseteq \models_\Sigma$ is natural in Σ. Therefore, the soundness axiom can be equivalently expressed by saying that the functor $\vdash: \underline{Sign} \longrightarrow \underline{Set}$, is a *subfunctor* of the functor $\models: \underline{Sign} \longrightarrow \underline{Set}$, and the completeness axiom by saying that the two functors are identical.

3. A logic $\mathcal{L}$ determines two categories of theories that have the same objects, but in general have different morphisms. One, $\underline{Th}$, comes from its underlying entailment system; the other, $\underline{Th_{\models}}$, comes from its underlying institution. The soundness axiom gives us a subcategory inclusion $\underline{Th} \hookrightarrow \underline{Th_{\models}}$, and completeness makes the categories identical.

2.5 Going to Extremes

Both entailment systems and institutions provide one-sided accounts of logic. The general notion of logic given in Section 2.4 has the pleasing flexibility of allowing us to regard an entailment system or an institution as a full logic of a special kind. In this way, the proof-theoretic and model-theoretic opposites are reconciled. In the end, from this abstract perspective, each can claim to have in some measure what the other contended it lacked.

From our discussion in Section 2.3 we immediately obtain the following,

Proposition 7 An institution $\mathcal{I}$ determines a complete logic having $\mathcal{I}^+$ as its underlying entailment system and having $\mathcal{I}$ itself as its underlying institution. By abuse of language, this logic is also denoted $\mathcal{I}^+$. □

A somewhat surprising fact is that, thanks to the generality of the axioms for a logic, we can pull a logic out of the proof-theoretic thin air of an entailment system. Fiadeiro and Sernades [17] associate an institution to a π-institution in a somewhat similar way. The notion of a *slice category* is used in the proof of Proposition 9 and will appear several other times in this paper. Slice categories are an instance of Lawvere's [45] "comma category" construction (see [47]).

Definition 8 For B an object in a category $\mathcal{C}$, the *slice category* $B/\mathcal{C}$ has as objects morphisms $f : B \longrightarrow A$ and as morphisms from, say, $f : B \longrightarrow A$ to $g : B \longrightarrow C$ those $h : A \longrightarrow C$ in $\mathcal{C}$ such that $h \circ f = g$; morphism composition is exactly as in $\mathcal{C}$. Dually, the slice category $\mathcal{C}/B$ has as objects morphisms $f : A \longrightarrow B$ and as morphisms from, say, $f : A \longrightarrow B$ to $g : C \longrightarrow B$ those $h : A \longrightarrow C$ in $\mathcal{C}$ such that $g \circ h = f$; again, morphism composition is as in $\mathcal{C}$. □

Proposition 9 We can associate to any entailment system $\mathcal{E}$ a logic $\mathcal{E}^\dagger$ that has $\mathcal{E}$ as its underlying entailment system. Besides, $\mathcal{E}^\dagger$ is complete, exact, and admits initial objects. If $\underline{Th}$ has pushouts, $\mathcal{E}^\dagger$ is also liberal.

Proof: For Σ a signature we define $\underline{Mod}(\Sigma)$ as the slice category $(\Sigma, \emptyset)/\underline{Th}$. A signature morphism $H : \Sigma \longrightarrow \Sigma'$ induces a functor $H^\flat : \underline{Mod}(\Sigma') \longrightarrow \underline{Mod}(\Sigma)$ which is just composition with H, i.e., $H^\flat(G') = G' \circ H$. Given $(G : (\Sigma, \emptyset) \longrightarrow (\Sigma', \Gamma')) \in \underline{Mod}(\Sigma)$ satisfaction is defined by $G \models_\Sigma \varphi$ iff $\Gamma' \vdash_{\Sigma'} G(\varphi)$. Therefore, G satisfies a set of Σ-sentences Γ iff there is a theory morphism $G : (\Sigma, \Gamma) \longrightarrow (\Sigma', \Gamma')$. Thus, for $T \in \underline{Th}$,

$\underline{Mod}(T)$ can be identified with the slice category $T/\underline{Th}$. The logic is complete, since the identity theory morphism $1_{(\Sigma,\Gamma)} \in \underline{Mod}(\Sigma,\Gamma)$ is such that $1_{(\Sigma,\Gamma)} \models_\Sigma \varphi$ iff $\Gamma \vdash_\Sigma \varphi$. Any slice category B/C has 1_B as its initial object and therefore $\mathcal{E}^\dagger$ admits initial models, with $I_T = 1_T$. When $\underline{Th}$ has pushouts, exactness and liberality follow easily from the elementary properties of a pushout; in the latter case, for $G \in \underline{Mod}(T)$ and $H : T \longrightarrow T'$ a theory morphism, we define $H^*(G)$ as the pushout of G along H. □

3 Proof Calculi and Logical Systems

A reasonable objection to the above definition of logic is that *it abstracts away* the structure of proofs, since we know only that a set Γ of sentences entails another sentence φ, but no information is given about the internal structure of such a $\Gamma \vdash \varphi$ entailment. This observation, while entirely correct, may be a virtue rather than a defect, because the entailment relation $\vdash$ is precisely what remains *invariant* under the many equivalent proof calculi that can be used for a logic. For example, in first order logic we have many different proof calculi: Hilbert styled, sequent, natural deduction, etc., each leading to a different notion of proof. However, the logic always remains the same, first order logic, precisely because all proof calculi yield the same entailment relation $\vdash$. Therefore, rather than building a particular proof calculus into the definition of a logic, it seems more satisfactory to axiomatize separately a proof calculus $\mathcal{P}$ for a logic $\mathcal{L}$, so that many different such calculi can be used in connection with the same logic. This point is directly relevant to computer science, because it shows that we can change the operational semantics (i.e., the proof calculus) of a logic programming language without altering its mathematical semantics, provided that the old and the new operational semantics have the same entailments. If we want to choose a specific proof calculus for a logic, we call the resulting logic plus proof calculus a *logical system*. In usual practice, and specially in logic programming applications, we often find proof calculi where certain restrictions are placed on the signatures, the sentences used as axioms, and on those used as conclusions. This leads to the notion of a *proof subcalculus*. In addition, we must introduce the notion of an *effective* proof subcalculus.

The basic idea of a proof calculus is that we can associate to each theory T a proof-theoretic structure $P(T)$ consisting of all proofs that use the sentences of T as axioms. The structure of $P(T)$ will typically relate such proofs in some algebraic manner. For example, $P(T)$ may be a *multicategory*[6]. However, the general axioms of a proof calculus to be given below will not impose any particular structure; they will postulate that $P(T)$ has *some* structure, by declaring it an object of some category of structures.

Definition 10 A *multicategory* consists of a set $\mathcal{O}$ of *basic objects* together with a category $\mathcal{C}$ whose objects are finite strings $\Gamma = A_1, \ldots, A_n$ of elements of $\mathcal{O}$, and such that, denoting by Γ, Δ the concatenation of two strings and denoting by $\emptyset$ the empty string, the morphisms of $\mathcal{C}$ have a monoid structure, with the multiplication of two morphisms $\alpha : \Gamma_1 \longrightarrow \Delta_1$ and $\beta : \Gamma_2 \longrightarrow \Delta_2$ denoted α, β and being of the form $\alpha, \beta : \Gamma_1, \Gamma_2 \longrightarrow \Delta_1, \Delta_2$. In addition, the multiplication α, β is actually a functor

[6] I take ample liberties with this notion, due to Lambek [44], and give a definition that is not equivalent to Lambek's but allows viewing multicategories in Lambek's sense as a particular case.

$\_,\_ : C^2 \longrightarrow C$, i.e., it preserves identities and composition. For readers familiar with monoidal categories, we can rephrase the definition by saying that a multicategory is a strict monoidal category [47] whose monoid of objects is free. The general notion of homomorphism between multicategories is that of a functor that preserves the monoid operation $\_,\_$ on the nose, i.e., a strictly monoidal functor. However, I shall impose the additional restriction that the functor maps basic objects to basic objects. I denote by $\underline{MultCat}$ the category with objects multicategories and with morphisms the strictly monoidal functors that satisfy this additional restriction. □

Example 11 (Natural Deduction) Given a theory $T = (\Sigma, \Delta)$ in, say, first order logic, we can associate to it a multicategory $P(T)$ with $sen(\Sigma)$ as its set of basic objects[7] and with morphisms[8] $\alpha : A_1, ..., A_n \longrightarrow B_1, ..., B_m$ consisting of sequences $\alpha = \alpha_1, ..., \alpha_m$ with α_i a natural deduction proof tree of B_i whose leaves only involve formulas among those in Δ and in $A_1, ..., A_n$. The identity $id_{A_1,...,A_n}$ is the sequence $A_1, ..., A_n$ viewed as a sequence of proof trees; the multiplication α, β is the concatenation of the two strings of proof trees. Composition of $\alpha : A_1, ..., A_n \longrightarrow B_1, ..., B_m$ with $\beta : B_1, ..., B_m \longrightarrow C_1, ..., C_k$ is a sequence $\gamma = \gamma_1, ..., \gamma_k$ with γ_i the proof tree obtained from the tree β_i by glueing the tree α_j at each leaf occurrence of B_j. □

We would like to view P as a functor $\underline{Th} \longrightarrow \underline{MultCat}$. However, some caution is required. The problem is that, given a theory morphism $H : (\Sigma, \Gamma) \longrightarrow (\Sigma', \Gamma')$ a sentence $\varphi \in \Gamma$ is mapped by H to an element of $\Gamma'^{\bullet}$, but $H(\varphi)$ does not necessarily belong to Γ'. In seeking a natural translation of proof trees to define a morphism $P(H) : P(\Sigma, \Gamma) \longrightarrow P(\Sigma', \Gamma')$ in $\underline{MultCat}$, we run into a problem of indeterminacy. This problem appears when we try to map the proof $\emptyset \longrightarrow \varphi$, of an axiom $\varphi \in \Gamma$, which consists of the one node tree φ, to a proof $\emptyset \longrightarrow H(\varphi)$, of $H(\varphi)$, since when $H(\varphi)$ is not in Γ' many different proofs may be possible. This difficulty has an easy solution by restricting our attention to the subcategory $\underline{Th_0} \hookrightarrow \underline{Th}$ of axiom-preserving theory morphisms. In this way, we get a functor $\underline{Th_0} \longrightarrow \underline{MultCat}$.

We can forget about the compositionality of proofs, and extract from $P(T)$ the *set* of all proofs of theorems of T, $proofs(T) = \{\alpha : \emptyset \longrightarrow \varphi \text{ in } P(T) \mid \varphi \in sen(T)\}$. We can obtain $proofs(T)$ as the set $Pr(P(T))$, where Pr is a functor $Pr : \underline{MultCat} \longrightarrow \underline{Set}$ sending each multicategory (O, C) to the set $Pr(O, C) = \{\alpha : \emptyset \longrightarrow A \text{ in } C \mid A \in O\}$. The way in which the proof calculus and the entailment system are linked is quite natural; it is given by the function $\pi_T : proofs(T) \longrightarrow sen(T)$ mapping each proof $\alpha : \emptyset \longrightarrow \varphi$ to its corresponding theorem φ. Therefore, the inverse image $\pi_T^{-1}(\varphi)$ yields the set of all proofs of φ; if φ is not a theorem this is the empty set. It is then easy to check that π_T is a natural transformation $\pi : Pr \circ P \Rightarrow sen$.

Definition 12 A *proof calculus* is a 6-tuple $\mathcal{P} = (\underline{Sign}, sen, \vdash, P, Pr, \pi)$ with:

1. $(\underline{Sign}, sen, \vdash)$ an entailment system;

[7] Actually, we want the basic objects to be *formulas* rather than sentences; however, this is a minor point, since we may assume that $sen(\Sigma)$ has been defined as consisting of formulas, and the notion of satisfaction extends easily to formulas.

[8] Notice that the sequences of sentences have *conjunctive* meaning in both the domain and the codomain of a morphism α.

2. $P : \underline{Th_0} \longrightarrow \underline{Struct_{\mathcal{P}}}$ a functor; for each theory T, the object $P(T) \in \underline{Struct_{\mathcal{P}}}$ is called its ***proof-theoretic structure***;
3. $Pr : \underline{Struct_{\mathcal{P}}} \longrightarrow \underline{Set}$ a functor; for each theory T, the set $Pr(P(T))$ is called its ***set of proofs***. We shall denote by ***proofs*** the composite functor $Pr \circ P : \underline{Th_0} \longrightarrow \underline{Set}$;
4. $\pi : \mathit{proofs} \Longrightarrow \mathit{sen}$ a natural transformation, such that for each theory $T = (\Sigma, \Gamma)$, the image of $\pi_T : \mathit{proofs}(T) \longrightarrow \mathit{sen}(T)$ is the set $\Gamma^\bullet$. The map π_T is called the ***projection from proofs to theorems*** for the theory T.

A proof calculus $\mathcal{P}$ is called ***compact*** iff its underlying entailment system, denoted $\mathit{ent}(\mathcal{P})$, is compact. □

We are now ready to axiomatize the notion of a logical system, consisting of a logic together with a choice of a proof calculus for it.

Definition 13 A *logical system* is an 8-tuple $\mathcal{S} = (\underline{Sign}, sen, \underline{Mod}, \vdash, \models, P, Pr, \pi)$ such that:

1. $(\underline{Sign}, sen, \underline{Mod}, \vdash, \models)$ is a logic, and
2. $(\underline{Sign}, sen, \vdash, P, Pr, \pi)$ is a proof calculus.

A logical system is called ***complete, compact, liberal, exact***, or is said to ***admit initial models*** iff the corresponding properties hold for its underlying logic. Given a logical system $\mathcal{S}$, its underlying logic will be denoted $\mathit{log}(\mathcal{S})$; similarly, its underlying proof calculus will be denoted $\mathit{pcalc}(\mathcal{S})$. □

3.1 Proof Subcalculi

It is quite common to encounter proof systems of a specialized nature. In these calculi, only certain signatures are admissible as syntax, e.g., finite signatures, only certain sentences are allowed as axioms, and only certain sentences (possibly different from the axioms) are allowed as conclusions. The obvious reason for introducing such calculi is that proofs are ***simpler*** under the given restrictions. This may serve technical or esthetical and expository purposes in logic; in computer science, however, the choice between an efficient and an inefficient calculus may have dramatic practical consequences. For logic programming languages, such calculi do (or should) coincide with what is called their ***operational semantics***, and mark the difference between a hopeless theorem prover and a very efficient programming language. Indeed, one of the main tasks of logic programming is finding efficient proof calculi by imposing judicious restrictions on the choice of axioms and conclusions. For example, the language Prolog emerged from the realization that resolution could be made much more efficient when the axioms are restricted to Horn clauses, and equational programming languages such as OBJ [19,29] exploit the fact that term rewriting, which is complete for equations that are Church-Rosser and terminating, is enormously more efficient than unrestricted equational deduction.

Definition 14 A *proof subcalculus* is a 9-tuple $\mathcal{Q} = (\underline{Sign}, sen, \vdash, \underline{Sign_0}, ax, concl, P, Pr, \pi)$, with:

1. $(\underline{Sign}, sen, \vdash)$ an entailment system.
2. $\underline{Sign}_0$ a subcategory of $\underline{Sign}$ called the subcategory of ***admissible signatures***. We denote by sen_0 the restriction of the functor sen to the subcategory $\underline{Sign}_0$.
3. $ax : \underline{Sign}_0 \longrightarrow \underline{Set}$ a subfunctor of the functor obtained by composing sen_0 with the powerset functor, i.e., there is a natural inclusion $ax(\Sigma) \subseteq \mathcal{P}(sen(\Sigma))$ for each $\Sigma \in \underline{Sign}_0$. Each $\Gamma \in ax(\Sigma)$ is called a set of ***admissible axioms*** specified by $\mathcal{Q}$. This defines a subcategory $\underline{Th}_{ax}$ of $\underline{Th}_0$ whose objects are theories $T = (\Sigma, \Gamma)$ with $\Sigma \in \underline{Sign}_0$ and $\Gamma \in ax(\Sigma)$, and whose morphisms are axiom-preserving theory morphisms H such that H is in $\underline{Sign}_0$.
4. $concl : \underline{Sign}_0 \longrightarrow \underline{Set}$ a subfunctor of the sen_0 functor. The sentences $\varphi \in concl(\Sigma)$ are called the ***admissible conclusions*** specified by $\mathcal{Q}$.
5. $P : \underline{Th}_{ax} \longrightarrow \underline{Struct}_{\mathcal{Q}}$ a functor; for each $T \in \underline{Th}_{ax}$, the object $P(T) \in \underline{Struct}_{\mathcal{Q}}$ is called its ***proof-theoretic structure.***
6. $Pr : \underline{Struct}_{\mathcal{Q}} \longrightarrow \underline{Set}$ a functor; for each $T \in \underline{Th}_{ax}$, the set $Pr(P(T))$ is called its ***underlying set of proofs*** of admissible conclusions. We denote by $proofs$ the composite functor $Pr \circ P$.
7. $\pi : proofs \Longrightarrow sen_0$ a natural transformation, such that for each $T = (\Sigma, \Gamma) \in \underline{Th}_{ax}$ the image of $\pi_T : proofs(T) \longrightarrow sen(T)$ is the set $\Gamma^\bullet \cap concl(\Sigma)$. The map π_T is called the ***projection from proofs to admissible theorems*** for the theory T.

Given a proof subcalculus $\mathcal{Q}$, $ent(\mathcal{Q})$ will denote its underlying entailment system.

□

Notice that when no restrictions at all are placed on signatures, axioms and conclusions, i.e., when $\underline{Sign}_0 = \underline{Sign}$, $ax(\Sigma) = \mathcal{P}(sen(\Sigma))$ and $concl(\Sigma) = sen(\Sigma)$, a proof subcalculus is the same thing as a proof calculus.

We can, finally, axiomatize the notion of a logical subsystem, consisting of a logic together with a choice of a proof subcalculus for it.

Definition 15 A *logical subsystem* is an 11-tuple $\mathcal{S} = (\underline{Sign}, sen, \underline{Mod}, \vdash, \models, \underline{Sign}_0, ax, concl, P, Pr, \pi)$ such that:

1. $(\underline{Sign}, sen, \underline{Mod}, \vdash, \models)$ is a logic, and
2. $(\underline{Sign}, sen, \vdash, \underline{Sign}_0, ax, concl, P, Pr, \pi)$ is a proof subcalculus.

Given a logical subsystem $\mathcal{S}$, its underlying logic will be denoted $log(\mathcal{S})$; similarly, its underlying proof subcalculus will be denoted $pscalc(\mathcal{S})$. □

3.2 Effective Proof Subcalculi

This section gives additional axioms for proof calculi that are effective in the intuitive sense of being mechanizable by an (idealized) computer. The axioms are not as expressive or as general as possible[9]; however, they will suffice for many purposes, including our axiomatization of logic programming in Section 6.

[9]For example, they only consider theories with a finite set of axioms, and proof-theoretic structures are involved only indirectly, through their underlying set of proofs.

The challenge in a topic like this is to avoid boring and annoying the reader (and the writer!) with horrible encodings of everything into the natural numbers. To this purpose, I will follow the axiomatic approach to computability outlined by Shoenfield in [66]. The basic notions are that of a *finite object*, a *space* of finite objects, and an *algorithm*. In Shoenfield's own words, a *finite object* is an "object which can be specified by a finite amount of information;" computer scientists would call this a finite *data structure*. A *space* is "an infinite class X of finite objects such that, given a finite object x, we can decide whether or not x belongs to X." Computer scientists would call this a *data type*. Given spaces X and Y, a *recursive function* $f : X \longrightarrow Y$ is then a (total!) function that can be computed by an *algorithm*, i.e., by a computer program, when we disregard space and time limitations; more generally, if the algorithm may not terminate for some inputs, we call the corresponding f a *partial recursive function* from X to Y. An *r.e. subset* of a space Y is a subset of the form $f(X)$ for some partial recursive function $f : X \longrightarrow Y$. Spaces and recursive functions form a category $\underline{Space}$, and there is an obvious forgetful functor $\mathcal{U} : \underline{Space} \longrightarrow \underline{Set}$ to the category of sets and functions. Notice that if X is a space, then the set $\mathcal{P}_{fin}(X)$ of finite subsets of X is also a space. We are now ready to axiomatize effective proof calculi. Since proof subcalculi generalize proof calculi, only effective proof subcalculi are defined. The reader may keep in mind the case of first order deduction for theories with a finite signature and a finite set of axioms —possibly with additional restrictions on the axioms and on the theorems that we wish to prove— as a standard example. Another interesting example is the effective proof subcalculus of equational logic provided by Church-Rosser and terminating term rewriting systems, in which the admissible signatures are finite signatures, and the sets of admissible axioms are finite sets of Church-Rosser and terminating equations, with proofs being performed by term rewriting. Note that the Church-Rosser property is a property of an entire *set* of equations, not of the individual equations. Many other examples of effective proof subcalculi are discussed in Section 6.

Definition 16 An *effective proof subcalculus* is a 10-tuple $\mathcal{Q} = (\underline{Sign}, sen, \vdash, \underline{Sign}_0, sen_0, ax, concl, P, Pr, \pi)$, such that:

1. $(\underline{Sign}, sen, \vdash)$ is an entailment system.
2. $\underline{Sign}_0$ is a subcategory of $\underline{Sign}$ called the subcategory of *admissible signatures*. We denote by J the subcategory inclusion functor $\underline{Sign}_0 \hookrightarrow \underline{Sign}$.
3. $sen_0 : \underline{Sign}_0 \longrightarrow \underline{Space}$ is a functor such that $\mathcal{U} \circ sen_0 = sen \circ J$.
4. $ax : \underline{Sign}_0 \longrightarrow \underline{Space}$ is a subfunctor of the functor obtained by composing $sen_0 : \underline{Sign}_0 \longrightarrow \underline{Space}$ with the functor $\mathcal{P}_{fin} : \underline{Space} \longrightarrow \underline{Space}$, that sends each space to the space of its finite subsets. This defines a subcategory $\underline{Th}_{ax}$ of $\underline{Th}_0$ whose objects are theories $T = (\Sigma, \Gamma)$ with $\Sigma \in \underline{Sign}_0$ and $\Gamma \in ax(\Sigma)$, and whose morphisms are axiom-preserving theory morphisms H such that H is in $\underline{Sign}_0$.
5. $concl : \underline{Sign}_0 \longrightarrow \underline{Space}$ is a subfunctor of the functor $sen_0 : \underline{Sign} \longrightarrow \underline{Space}$.
6. $P : \underline{Th}_{ax} \longrightarrow \underline{Struct}_{\mathcal{Q}}$ is a functor.
7. $Pr : \underline{Struct}_{\mathcal{Q}} \longrightarrow \underline{Space}$ is a functor. We denote by *proofs* the composite functor $Pr \circ P$.

8. $\pi : proofs \Longrightarrow sen_0$ is a natural transformation.

9. Denoting also by *ax*, *concl*, *Pr*, and π the results of composing with $\mathcal{U}$ each of the above functors and the natural transformation π, the 9-tuple $\mathcal{U}(\mathcal{Q}) = (\underline{Sign}, sen, \vdash, \underline{Sign_0}, ax, concl, P, Pr, \pi)$ is a proof subcalculus.

□

Notice that if $\mathcal{Q}$ is an effective proof subcalculus, for each theory $T \in \underline{Th_{ax}}$ there is an associated partial recursive function $search_T : concl(T) \times \mathbb{N} \longrightarrow \mathcal{P}_{fin}(proofs(T))$ such that for each pair (φ, n), consisting of an admissible conclusion φ and a natural number n, $search_T(\varphi, n)$ is undefined if $\pi_T^{-1}(\varphi)$ has cardinality strictly less than n, and otherwise yields a subset of n elements in $\pi_T^{-1}(\varphi)$ such that, when $\pi_T^{-1}(\varphi)$ is infinite, we have $\pi_T^{-1}(\varphi) = \bigcup_{n\in\mathrm{Nat}} search_T(\varphi, n)$. An algorithm to compute $search_T$ can be obtained as follows: since all spaces are isomorphic [66], there is a *listing* of $proofs(T)$, i.e., a bijective recursive function $\alpha : \mathbb{N} \longrightarrow proofs(T)$. We define $search_T(\varphi, n) = \emptyset$; to compute $search_T(\varphi, n+1)$, we just scan through the listing α until we find the first $n+1$ proofs $\alpha(i_1), ..., \alpha(i_{n+1})$ such that $\pi_T(\alpha(i_j)) = \varphi$. At times, it is possible to do better than this, and provide an algorithm that behaves like the above $search_T$ when $search_T$ is defined, but such that for some of the inputs (φ, n) for which $search_T$ is undefined, it yields the value $\emptyset$, understood as positive *failure in finite time* to find a set of n proofs. We shall call any such function a *search function* for T in the subcalculus $\mathcal{Q}$.

4 Relating Logics

In this section, different notions of map are introduced and motivated with examples for the different logical structures, i.e., for entailment systems, institutions, logics, proof (sub)calculi, and logical (sub)systems. Each such type of logical structure together with its corresponding maps determines a category, and those categories are then related by forgetful functors and by adjoint functors. In all cases there is a notion of *substructure*, such as a subentailment system, a sublogic, etc., that is always expressed as a map satisfying special properties, thus giving an axiomatic expression to the corresponding intuitive notion.

4.1 Mapping Entailment Systems

The intuitive idea is simple enough: we want to map the syntax and the sentences between two entailment systems in a way that is consistent, i.e., such that it preserves entailments.

Example 17 Consider the relationship between the entailment systems $ent(Eqtl)$ of equational logic and $ent(Fol)$ of first order logic with equality. We map a functional signature, consisting of a ranked alphabet F of function symbols, to the first order signature $\Phi(F) = (F, \emptyset)$ with same function symbols and with no predicate symbols. Similarly, we can define a map α sending an equation $t = t'$ to the first order sentence $\forall x_1 ... \forall x_n\ t = t'$ where $x_1, ..., x_n$ are the variables occurring in either t or t'. For Γ a set of equations and $\alpha(\Gamma)$ the set of its corresponding first order sentences we of course have

$$\Gamma \vdash_{P} t = t' \ \Rightarrow\ \alpha(\Gamma) \vdash_{\Phi(P)} \alpha(t = t').$$

Indeed, we actually have an equivalence rather than just an implication between the two entailments, so that this particular map is *conservative*. $\Box$

Maps of entailment systems that send a signature to another signature are called *plain*. However, there are natural examples of maps that are not plain.

Example 18 Let $\forall Fol$ denote the fragment of first order logic without equality consisting of sentences that are the universal quantification of a quantifier free formula. Let $\forall FnFol_2^=$ be the fragment of two-sorted first order logic with equality having signatures that involve only function symbols, and with sentences also restricted to universal sentences. We shall use the symbol u to denote one of the sorts, and the symbol *bool* to denote the other. The idea of viewing every predicate as a characteristic function yields a map relating the entailment systems of these two logics. However, an unsorted signature (F, P) should not be mapped to another signature, but to a *theory* $\Phi(F, P)$ whose signature consists only of the following function symbols:

1. for each n-ary function symbol $f \in F$ there is a function symbol $f : u^n \longrightarrow u$;
2. for each n-ary predicate symbol $p \in P$ there is a function symbol $p^\circ : u^n \longrightarrow bool$;
3. there are constants *true, false*, a unary operation *not* and binary operations *and, or, implies*, all with sort *bool* for their arguments and their result.

The axioms of the theory $\Phi(F, P)$ are those needed to force the interpretation of the sort *bool* to be a two-element boolean algebra. The key axioms are $true \neq false$ and $(\forall b : bool)\ (b = true) \vee (b = false)$. In addition, we must give equational axioms forcing *not, and, or* and *implies* to have the standard meaning.

We can now define a translation α between sentences in the expected way. We translate a quantifier free formula φ into a term φ° of sort *bool* as follows. An atomic formula $p(t_1, ..., t_n)$ is translated into the term $p^\circ(t_1, ..., t_n)$, and the connectives are translated in the obvious way, e.g., $\varphi \wedge \psi$ is translated into $and(\varphi^\circ, \psi^\circ)$, etc. We then translate a sentence $\forall x_1 ... \forall x_n\ \varphi$ with φ quantifier free into the equational sentence $\forall x_1 ... \forall x_n\ \varphi^\circ = true$. As in the previous example we have

$$\Gamma \vdash_{\Sigma} \varphi \ \Rightarrow\ \alpha(\Gamma) \vdash_{\Phi(\Sigma)} \alpha(\varphi)$$

and, indeed, the implication is also in this case an equivalence. The map from signatures to theories is in fact a functor[10] $\Phi : \underline{Sign}_{\forall Fol} \longrightarrow \underline{Th}_{\forall FnFol_2^= 0}$, and the translation α is a natural transformation $\alpha : sen_{\forall Fol} \Longrightarrow sen_{\forall FnFol_2^=} \circ \Phi$. $\Box$

Another interesting example of a map of entailment systems that is not plain is furnished by the translation of the second order lambda calculus into Martin-Löf type theory described in [52].

Notice that any functor $\Phi : \underline{Sign} \longrightarrow \underline{Th}'_0$ together with a natural transformation $\alpha : sen \Longrightarrow sen' \circ \Phi$ can easily be extended to a functor $\Phi : \underline{Th}_0 \longrightarrow \underline{Th}'_0$, called

[10]As before, we denote by $\underline{Th}_0$ the subcategory of $\underline{Th}$ whose theory morphisms map axioms to axioms.

the *α-extension to theories* of the original functor, by mapping a theory $T = (\Sigma, \Gamma)$ to the theory $\Phi(T)$ with same signature as that of $\Phi(\Sigma)$ and with axioms those of $\Phi(\Sigma)$ together with the axioms $\alpha_\Sigma(\Gamma)$. Notice also that the natural transformation $\alpha : sen \Longrightarrow sen' \circ \Phi$ can be similarly extended to a natural transformation $\alpha : sen \Longrightarrow sen' \circ \Phi$ between the functors $sen : \underline{Th_0} \longrightarrow \underline{Set}$ and $sen' \circ \Phi : \underline{Th_0} \longrightarrow \underline{Set}$; we just define $\alpha_{(\Sigma,\Gamma)} = \alpha_\Sigma$. Therefore, we may as well view Φ as a functor $\Phi : \underline{Th_0} \longrightarrow \underline{Th'_0}$ mapping theories to theories, and call a functor $\Psi : \underline{Th_0} \longrightarrow \underline{Th'_0}$ *α-simple* iff it is in fact the α-extension to theories of a functor $\Psi : \underline{Sign} \longrightarrow \underline{Th'_0}$. In addition, we call Ψ *α-plain* if it is the α-extension to theories of a functor Ψ that factors through the functor $F : \underline{Sign'} \longrightarrow \underline{Th'_0}$ sending each signature Σ' to the theory $(\Sigma', \emptyset)$, i.e., if it is the α-extension to theories of a functor mapping signatures to signatures.

This way of relating entailment systems is already quite general, but is it general enough? The answer is "no." There are natural and interesting examples of maps between entailment systems that map theories in a more subtle way.

Example 19 (Unfailing Knuth-Bendix Completion) Consider the entailment system of equational logic $ent(Eqtl)$. By using an unfailing Knuth-Bendix algorithm [1], we can associate to an equational theory $T = (\Sigma, \Gamma)$ a (possibly infinite) Knuth-Bendix completion $KB(T) = (\Sigma, KB(\Gamma))$ so that equational deduction in T can be treated by term rewriting in $KB(T)$. This can be viewed as a functor, $KB : \underline{Th_{Eqtl0}} \longrightarrow \underline{Th_{Eqtl0}}$ which is not a simple functor. Since T and $KB(T)$ are isomorphic theories we have

$$\Gamma \vdash_T t = t' \iff \Gamma \vdash_{KB(T)} t = t'.$$

After defining the notion of a mapping of entailment systems we shall be able to see that the functor KB, together with the identity natural transformation 1_{sen} from sen to itself, give us an entailment system map $(KB, 1_{sen}) : ent(Eqtl) \longrightarrow ent(Eqtl)$. □

An even simpler example is provided by closure under entailment.

Example 20 (Entailment Closure) Given an entailment system $\mathcal{E} = (\underline{Sign}, sen, \vdash)$, the functor $(\_)^\bullet : \underline{Th_0} \longrightarrow \underline{Th_0}$ mapping a theory $T = (\Sigma, \Gamma)$ to the theory $T^\bullet = (\Sigma, \Gamma^\bullet)$, together with the identity natural transformation 1_{sen} from sen to itself, will give us an entailment system map $((\_)^\bullet, 1_{sen}) : \mathcal{E} \longrightarrow \mathcal{E}$. □

Notice that the functors KB and $(\_)^\bullet$ map theories having the same signature to theories having the same signature; actually, in these two examples signatures are left unchanged. Let us denote by (Σ', Γ') the image obtained by applying to a theory (Σ, Γ) a functor $\Phi : \underline{Th_0} \longrightarrow \underline{Th'_0}$ that maps theories with the same signature to theories with the same signature. In particular, we denote by $(\Sigma', \emptyset')$ the theory $\Phi(\Sigma, \emptyset)$. If the functor Φ is α-simple, we have the following property:

$$\Gamma' = \emptyset' \cup \alpha_\Sigma(\Gamma).$$

This is not satisfied by KB and $(\_)^\bullet$; however, they satisfy the weaker condition

$$(\Gamma')^\bullet = (\emptyset' \cup \alpha_\Sigma(\Gamma))^\bullet.$$

These functors, although more general, are "sensible" in the following sense.

Definition 21 Given entailment systems $\mathcal{E} = (\underline{Sign}, sen, \vdash)$ and $\mathcal{E}' = (\underline{Sign}', sen', \vdash')$, a functor $\Phi : \underline{Th_0} \longrightarrow \underline{Th'_0}$ and a natural transformation $\alpha : sen \Longrightarrow sen' \circ \Phi$, we call Φ *α-sensible* iff the following conditions are satisfied:

1. There is a functor $\Phi^\diamond : \underline{Sign} \longrightarrow \underline{Sign'}$ such that $sign' \circ \Phi = \Phi^\diamond \circ sign$.
2. $(\Gamma')^\bullet = (\emptyset' \cup \alpha_\Sigma(\Gamma))^\bullet$.

□

α-sensible functors have the nice property that their natural transformation α only depends on the signatures, not on the theories. This is a consequence of the following lemma.

Lemma 22 Given entailment systems $\mathcal{E} = (\underline{Sign}, sen, \vdash)$ and $\mathcal{E}' = (\underline{Sign'}, sen', \vdash')$, and a functor $\Phi : \underline{Th_0} \longrightarrow \underline{Th'_0}$ satisfying condition (1) in Definition 21, then any natural transformation $\alpha : sen \Longrightarrow sen' \circ \Phi$ can be obtained by horizontal composition with the functor $sign : \underline{Th_0} \longrightarrow \underline{Sign}$ from a natural transformation $\alpha^\diamond : sen \Longrightarrow sen' \circ \Phi^\diamond$.

Proof: Again, we use the notation $\Phi(\Sigma, \Gamma) = (\Sigma', \Gamma')$. What we have to show is that for any $T_1 = (\Sigma, \Gamma)$, $T_2 = (\Sigma, \Delta)$ in $\underline{Th_0}$, we have $\alpha_{T_1} = \alpha_{T_2}$. Then, we can define $\alpha^\diamond_\Sigma = \alpha_{T_1}$. Notice that, for any $T_2 = (\Sigma, \Delta)$, we have a theory morphism $1_\Sigma : (\Sigma, \emptyset) \longrightarrow (\Sigma, \Delta)$ in $\underline{Th_0}$. Therefore, it is enough to establish this property when $T_1 = (\Sigma, \emptyset)$. Since Φ satisfies condition (1), we must have:

$$\Phi(1_\Sigma : (\Sigma, \emptyset) \longrightarrow (\Sigma, \Delta)) = (1_{\Sigma'} : (\Sigma', \emptyset') \longrightarrow (\Sigma', \Delta')).$$

Therefore, we have $sen(1_\Sigma) = 1_{sen(\Sigma)}$, and $sen'(\Phi(1_\Sigma)) = sen'(1_{\Sigma'}) = 1_{sen'(\Sigma')}$. The naturality of α then forces $\alpha_{T_1} = \alpha_{T_2}$, as desired. □

By abuse of language, in the following we shall drop the "diamond" and write α_Σ instead of $\alpha^\diamond_\Sigma$.

Definition 23 Given entailment systems $\mathcal{E} = (\underline{Sign}, sen, \vdash)$ and $\mathcal{E}' = (\underline{Sign'}, sen', \vdash')$, a *map of entailment systems* $(\Phi, \alpha) : \mathcal{E} \longrightarrow \mathcal{E}'$ consists of a a natural transformation $\alpha : sen \Longrightarrow sen' \circ \Phi$ and an α-sensible functor $\Phi : \underline{Th_0} \longrightarrow \underline{Th'_0}$ satisfying the following property:

$$\Gamma \vdash_\Sigma \varphi \;\Rightarrow\; \alpha_\Sigma(\Gamma) \vdash'_{\Phi(\Sigma,\emptyset)} \alpha_\Sigma(\varphi).$$

We call (Φ, α) *conservative* if in addition we have,

$$\Gamma \vdash_\Sigma \varphi \;\Leftrightarrow\; \alpha_\Sigma(\Gamma) \vdash'_{\Phi(\Sigma,\emptyset)} \alpha_\Sigma(\varphi).$$

We call (Φ, α) *plain* if Φ is α-plain, and, similarly, we call (Φ, α) *simple* if Φ is α-simple.

A *subentailment system* is a map $(\Phi, \alpha) : \mathcal{E} \longrightarrow \mathcal{E}'$ of entailment systems that is plain, conservative, with Φ faithful and injective on objects, and with α injective. We write $(\Phi, \alpha) : \mathcal{E} \hookrightarrow \mathcal{E}'$ to denote a subentailment system. □

Example 17 shows that the entailment system of equational logic is a subentailment system of that of first order logic. Note that in that example neither Φ nor α are actual inclusions; they are only injections.

The following lemma follows easily from the basic properties of sensible functors ana is left as an exercise.

Lemma 24 For $(\Phi, \alpha) : \mathcal{E} \longrightarrow \mathcal{E}'$ a map of entailment systems, the following property is satisfied:

$$\Gamma \vdash_{(\Sigma,\Delta)} \varphi \;\Rightarrow\; \alpha_\Sigma(\Gamma) \vdash'_{\Phi(\Sigma,\Delta)} \alpha_\Sigma(\varphi).$$

Furthermore, if (Φ, α) is conservative, the above implication is actually an equivalence. $\Box$

Given maps of entailment systems $(\Phi_1, \alpha_1) : \mathcal{E} \longrightarrow \mathcal{E}'$ and $(\Phi_2, \alpha_2) : \mathcal{E}' \longrightarrow \mathcal{E}''$ we can define their *composition* $(\Phi_2, \alpha_2) \circ (\Phi_1, \alpha_1) : \mathcal{E} \longrightarrow \mathcal{E}''$ as the pair (Φ_3, α_3) with $\Phi_3 = \Phi_2 \circ \Phi_1$ and with α_3 the natural transformation obtained by pasting together the two cells

$$\begin{tikzcd}
\underline{Th_0} \arrow[r, "\Phi_1"] \arrow[dr, "sen"'] & \underline{Th'_0} \arrow[r, "\Phi_2"] \arrow[d, "sen'"] \arrow[dl, phantom, "\overset{\alpha_1}{\Rightarrow}"] \arrow[dr, phantom, "\overset{\alpha_2}{\Rightarrow}"] & \underline{Th''_0} \arrow[dl, "sen''"] \\
 & \underline{Set} &
\end{tikzcd}$$

Using Lemma 24, it is easy to check that this composition is itself also a map of entailment systems and that, therefore, we have a category $\underline{Ent}$ whose objects are entailment systems and whose morphisms are maps between them.

The following lemma is left as an exercise.

Lemma 25 The composition of conservative maps of entailment systems is conservative. Furthermore, if for $(\Phi_1, \alpha_1) : \mathcal{E} \longrightarrow \mathcal{E}'$ and $(\Phi_2, \alpha_2) : \mathcal{E}' \longrightarrow \mathcal{E}''$ maps of entailment systems the composition $(\Phi_2, \alpha_2) \circ (\Phi_1, \alpha_1) : \mathcal{E} \longrightarrow \mathcal{E}''$ is conservative, then $(\Phi_1, \alpha_1) : \mathcal{E} \longrightarrow \mathcal{E}'$ is conservative. $\Box$

4.2 Mapping Institutions

The idea of a map of institutions is somewhat counterintuitive. Although the syntax part is mapped just as for entailment systems, the models are mapped in the opposite direction. This can be best illustrated by an example.

Example 26 Let *Eqtl* be equational logic, and let *MSEqtl* be many-sorted equational logic. The process of "omitting sorts" should map the underlying institution of *MSEqtl* to the underlying institution of *Eqtl*. A many-sorted theory is a pair (S, F) with S a set of sorts, and F an alphabet of function symbols, each with a rank consisting of a pair (w, s) with $w \in S^*$ giving the sorts of the arguments and s the sort of the result. By omitting sorts, we map such a signature to the signature F such that f has rank n

iff, in its original rank (w, s), w had length n. This defines a functor $\Phi : \underline{Sign}_{MSEqtl} \longrightarrow \underline{Sign}_{Eqtl}$ from many-sorted to unsorted signatures. In many-sorted equational logic, the number and sort of the variables being quantified in an equation must be made explicit [31], but this is not required for unsorted equational logic. Omitting sorts and explicit quantification transforms a many-sorted equation into an unsorted equation, and we can view this as a natural transformation $\alpha : sen_{MSEqtl} \Longrightarrow sen_{Eqtl} \circ \Phi$. However, for (S, F) a many-sorted signature, there is no natural way of associating an unsorted F-algebra to a many-sorted (S, F)-algebra. What is natural is to associate to the F-algebra A the (S, F)-algebra $\beta(A)$ with carrier $\{A_s\}_{s \in S}$ such that $A_s = A$ for all $s \in S$, and such that each function symbol f with rank (w, s) such that w has length n is interpreted by the same function $A_f : A^n \longrightarrow A$ that interpreted f in the F-algebra A. This gives us a functor $\beta_{(F,S)} : \underline{Mod}_{Eqtl}(\Phi(F, S)) \longrightarrow \underline{Mod}_{MSEqtl}(F, S)$, and globally defines a natural transformation $\beta : \underline{Mod}_{Eqtl} \circ \Phi \Longrightarrow \underline{Mod}_{MSEqtl}$. By definition of $\beta(A)$, for any many-sorted signature (S, F) and any (S, F)-equation φ we have,

$$A \models_F \alpha(\varphi) \text{ iff } \beta(A) \models_{(S,F)} \varphi.$$

Notice, finally, that instead of restricting ourselves to equational logic we could just as well have considered unsorted first order logic and many-sorted first order logic. □

Definition 27 Given institutions $\mathcal{I} = (\underline{Sign}, sen, \underline{Mod}, \models)$ and $\mathcal{I}' = (\underline{Sign}', sen', \underline{Mod}', \models')$, a *map of institutions* $(\Phi, \alpha, \beta) : \mathcal{I} \longrightarrow \mathcal{I}'$ consists of a natural transformations $\alpha : sen \Longrightarrow sen' \circ \Phi$, an α-sensible functor[11] $\Phi : \underline{Th}_0 \longrightarrow \underline{Th}'_0$, and a natural transformation $\beta : \underline{Mod}' \circ \Phi^{op} \Longrightarrow \underline{Mod}$ such that for each $\Sigma \in \underline{Sign}$, $\varphi \in sen(\Sigma)$, and $M' \in \underline{Mod}'(\Phi(\Sigma, \emptyset))$ the following property is satisfied:

$$M' \models_{sign'(\Phi(\Sigma,\emptyset))} \alpha_\Sigma(\varphi) \text{ iff } \beta_{(\Sigma,\emptyset)}(M') \models_\Sigma \varphi.$$

We call (Φ, α, β) *plain* iff Φ is α-plain, and similarly call (Φ, α, β) *simple* iff Φ is α-simple.

A *subinstitution* is a map $(\Phi, \alpha, \beta) : \mathcal{I} \longrightarrow \mathcal{I}'$ of institutions that is plain, with Φ faithful and injective on objects, with α injective, and with β a natural isomorphism. We write $(\Phi, \alpha, \beta) : \mathcal{I} \hookrightarrow \mathcal{I}'$ to denote a subinstitution. For example, equational logic is a subinstitution of first order logic. □

Given maps of institutions $(\Phi_1, \alpha_1, \beta_1) : \mathcal{I} \longrightarrow \mathcal{I}'$ and $(\Phi_2, \alpha_2, \beta_2) : \mathcal{I}' \longrightarrow \mathcal{I}''$ we can define their *composition* $(\Phi_2, \alpha_2, \beta_2) \circ (\Phi_1, \alpha_1, \beta_1) : \mathcal{I} \longrightarrow \mathcal{I}''$ as the pair $(\Phi_3, \alpha_3, \beta_3)$ with $\Phi_3 = \Phi_2 \circ \Phi_1$ and with α_3 the natural transformation obtained by pasting together the two cells

$$\begin{tikzcd}
\underline{Th_0} \arrow[r, "\Phi_1"] \arrow[dr, "sen"'] & \underline{Th'_0} \arrow[r, "\Phi_2"] \arrow[d, "sen'"] & \underline{Th''_0} \arrow[dl, "sen''"] \\
& \underline{Set} &
\end{tikzcd}$$

with $\overset{\alpha_1}{\Longrightarrow}$ between sen and sen', and $\overset{\alpha_2}{\Longrightarrow}$ between sen' and sen''.

[11]The functor Φ is α-sensible for the entailment systems $(\underline{Sign}, sen, \models)$ and $(\underline{Sign}', sen', \models')$ associated to $\mathcal{I}$ and $\mathcal{I}'$. Since the categories $\underline{Th}_{\models 0}$ and $\underline{Th}'_{\models 0}$ do not depend at all on the entailment relations $\models$ and $\models'$, we write $\underline{Th_0}$ and $\underline{Th'_0}$ instead.

and β_3 the natural transformation obtained by pasting together the two cells

$$\begin{tikzcd}
\underline{Th_0}^{op} \arrow[rr, "\Phi_1^{op}"] \arrow[ddrr, "\underline{Mod}"'] & & \underline{Th'_0}^{op} \arrow[rr, "\Phi_2^{op}"] \arrow[dd, "\underline{Mod'}"] & & \underline{Th''_0}^{op} \arrow[ddll, "\underline{Mod''}"] \\
& \overset{\beta_1}{\Leftarrow} & & \overset{\beta_2}{\Leftarrow} & \\
& & \underline{Cat} & &
\end{tikzcd}$$

It is easy to check that this composition is itself a map of institutions. Therefore, we have a category $\underline{Inst}$ whose objects are institutions, and whose morphisms are maps between them. Recall now that, by Proposition 4, every institution $\mathcal{I}$ has an associated entailment system $\mathcal{I}^+$. This is just the object part of a functor $(\_)^+ : \underline{Inst} \longrightarrow \underline{Ent}$, as shown by the following lemma.

Lemma 28 If $(\Phi, \alpha, \beta) : \mathcal{I} \longrightarrow \mathcal{I}'$ is a map of institutions, then $(\Phi, \alpha) : \mathcal{I}^+ \longrightarrow \mathcal{I}'^+$ is a map of entailment systems.

Proof: We have to show

$$\Gamma \models_\Sigma \varphi \;\Rightarrow\; \alpha_\Sigma(\Gamma) \models'_{\Phi(\Sigma,\emptyset)} \alpha_\Sigma(\varphi)$$

which, rephrased in terms of the closures and adopting the notation $\Phi(\Sigma, \Gamma) = (\Sigma', \Gamma')$, reads

$$\varphi \in \Gamma^\bullet \;\Rightarrow\; \alpha_\Sigma(\varphi) \in (\emptyset' \cup \alpha_\Sigma(\Gamma))^\bullet,$$

but $\varphi \in \Gamma^\bullet$ iff for each $M \in \underline{Mod}(\Sigma, \Gamma)$ we have $M \models_\Sigma \varphi$. Notice that, since Φ is α-sensible, we have $\Phi(1_\Sigma : (\Sigma, \emptyset) \longrightarrow (\Sigma, \Gamma)) = 1_{\Sigma'} : (\Sigma', \emptyset') \longrightarrow (\Sigma', \Gamma')$ and therefore $\emptyset' \subseteq \Gamma'$ so that $\underline{Mod}(\Sigma', \Gamma') \subseteq \underline{Mod}(\Sigma', \emptyset')$. Now consider the functor

$$\beta_{(\Sigma,\Gamma)} : \underline{Mod'}(\Sigma', \Gamma') \longrightarrow \underline{Mod}(\Sigma, \Gamma);$$

by naturality of β this is just the restriction of the functor

$$\beta_{(\Sigma,\emptyset)} : \underline{Mod'}(\Sigma', \emptyset') \longrightarrow \underline{Mod}(\Sigma, \emptyset).$$

This implies that for each $M' \in \underline{Mod'}(\Phi(\Sigma, \Gamma))$, $\beta_{(\Sigma,\emptyset)}(M') \models_\Sigma \varphi$, which is equivalent to $M' \models_{\Sigma'} \alpha_\Sigma(\varphi)$. This shows $\alpha_\Sigma(\varphi) \in \Gamma'^\bullet$; and since Φ is α-sensible this is equivalent to $\alpha_\Sigma(\varphi) \in (\emptyset' \cup \alpha_\Sigma(\Gamma))^\bullet$, as desired. $\Box$

Remark. Although closely related and inspired by them, the maps of institutions defined above are different from what Goguen and Burstall call *institution morphisms* [27,26]. These are of the form $(\Phi, \alpha, \beta) : \mathcal{I} \longrightarrow \mathcal{I}'$, with $\Phi : \underline{Sign} \longrightarrow \underline{Sign'}$ a functor, and $\alpha : sen' \circ \Phi \Longrightarrow sen$, $\beta : \underline{Mod} \Longrightarrow \underline{Mod'} \circ \Phi^{op}$ natural transformations satisfying a condition entirely similar to the one above. Besides their restriction to a "plain" Φ, the main difference is that their α and β go in exactly the opposite direction than mine. Since the Φ still goes in the same direction, the concepts are not dual and their relationship is not entirely clear unless special properties, such as adjointness, are assumed for Φ. Both notions of mapping will probably be needed to account for all relevant examples. In this presentation, I have favored the notion of a map of institutions because it fits well many natural examples and permits flexible ways of mapping theories. Also, one of the motivations for defining maps of institutions is to introduce the concept of a subinstitution as a map of institutions that satisfies additional properties; this does not seem possible using institution morphisms.

4.3 Mapping Logics

Mapping logics is now easy. We just map their underlying entailment and institution components.

Definition 29 Given logics $\mathcal{L} = (\underline{Sign}, sen, \underline{Mod}, \vdash, \models)$ and $\mathcal{L}' = (\underline{Sign}', sen', \underline{Mod}', \vdash', \models')$, a *map of logics* $(\Phi, \alpha, \beta) : \mathcal{L} \longrightarrow \mathcal{L}'$ consists of a functor[12] $\Phi : \underline{Th}_0 \longrightarrow \underline{Th}'_0$, and natural transformations $\alpha : sen \Longrightarrow sen' \circ \Phi$ and $\beta : \underline{Mod}' \circ \Phi \Longrightarrow \underline{Mod}$ such that:

1. $(\Phi, \alpha, \beta) : inst(\mathcal{L}) \longrightarrow inst(\mathcal{L}')$ is a map of institutions, and
2. $(\Phi, \alpha) : ent(\mathcal{L}) \longrightarrow ent(\mathcal{L}')$ is a map of entailment systems.

Therefore, we have a category $\underline{Log}$ whose objects are logics and whose morphisms are maps of logics, and there are forgetful functors $inst : \underline{Log} \longrightarrow \underline{Inst}$ and $ent : \underline{Log} \longrightarrow \underline{Ent}$ yielding the underlying institution and the underlying entailment system of a logic respectively.

We call (Φ, α, β) *plain* iff Φ is α-plain, and, similarly, call (Φ, α, β) *simple* iff Φ is α-simple. We call (Φ, α, β) *conservative* iff (Φ, α) is so as a map of entailment systems[13].

A *sublogic* is a map $(\Phi, \alpha, \beta) : \mathcal{L} \longrightarrow \mathcal{L}'$ of logics that is both a subinstitution for the underlying institutions and a subentailment system for the underlying entailment systems. We write $(\Phi, \alpha, \beta) : \mathcal{L} \hookrightarrow \mathcal{L}'$ to denote a sublogic. □

Example 30 All the previous examples of maps of entailment systems and maps of institutions were fragmentary descriptions of maps of logics:

1. Example 17 comes from a sublogic *Eqtl* $\hookrightarrow$ *Fol*$^{=}$ that views equational logic as a fragment of first order logic with equality.
2. Omitting sorts is a (plain) map of logics *MSEqtl* $\longrightarrow$ *Eqtl* (or more generally, *MSFol*$^{=}$ $\longrightarrow$ *Fol*$^{=}$). The paper [31] shows that this map is *not* conservative, and characterizes the largest subcategory of $\underline{Sign}_{MSEqtl}$ for which omitting sorts is conservative.
3. Notice that, viewing each unsorted equational signature as a many-sorted signature with one sort gives us a sublogic *Eqtl* $\hookrightarrow$ *MSEqtl*, that when composed with the above "omitting sorts" map *MSEqtl* $\longrightarrow$ *Eqtl* yields the identity map on *Eqtl*; therefore, omitting sorts is a *retract* map.

[12]Notice that, since $\underline{Th}_0$ does not depend on the entailment relation, we have $\underline{Th}_0 = \underline{Th}'_{\models 0}$, so that, indeed, the domain and codomain of Φ are not altered by viewing Φ as a map of entailment systems or, alternatively, as a map of institutions.

[13]Strictly speaking, for a noncomplete logic we could distinguish between $\vdash$-conservative and $\models$-conservative maps. However, conservativeness of the underlying map of entailment systems seems the most important notion.

4. The encoding of predicates as characteristic functions is a conservative map of logics $\forall Fol \longrightarrow \forall FnFol_2^=$ which is not plain.

$\square$

The construction given in Section 2.5 of the logic $\mathcal{I}^+$ associated to an institution $\mathcal{I}$ is actually a functor $(\_)^+ : \underline{Inst} \longrightarrow \underline{Log}$. This follows easily from Lemma 28. Actually, we have two natural ways of associating a logic to an institution, one associates the most complete logic, and the other associates the least complete one.

Proposition 31 The forgetful functor $inst : \underline{Log} \longrightarrow \underline{Inst}$ has both a left and a right adjoint.

Proof: The right adjoint is precisely $(\_)^+$, and the unit map of the adjunction is the sublogic $\mathcal{L} \hookrightarrow (inst(\mathcal{L}))^+$. The left adjoint is the functor $(\_)^- : \underline{Inst} \longrightarrow \underline{Log}$ sending an institution $\mathcal{I}$ to the logic having $\mathcal{I}$ as its underlying institution and such that $\Gamma \vdash \varphi$ iff $\varphi \in \Gamma$. The counit of this adjunction is the sublogic $(inst(\mathcal{L}))^- \hookrightarrow \mathcal{L}$. $\square$

The construction given in Section 2.5 of the logic $\mathcal{E}^\dagger$ associated to the entailment system $\mathcal{E}$ does not seem to correspond to a functor $(\_)^\dagger : \underline{Ent} \longrightarrow \underline{Log}$. Given a map $(\Phi, \alpha) : \mathcal{E} \longrightarrow \mathcal{E}'$, the natural choice for a transformation β for the models seems to go in the wrong direction, since it sends a model $H : T \longrightarrow T'$ in $\mathcal{E}^\dagger$ to a model $\Phi(H) : \Phi(T) \longrightarrow \Phi(T')$ in $\mathcal{E}'^\dagger$. There are several possible choices for the directions of the natural transformations α and β corresponding to different notions of "map" betwen entailment systems and between institutions, and this construction happens to be functorial for a different choice of directions.

4.4 Mapping Proof Calculi

The intuitive idea is very simple. Given two proof calculi, a mapping between them must first of all involve a map (Φ, α) of their underlying entailment systems. In addition, we want to map a proof p of a theorem φ of a theory $T \in \underline{Th_0}$ to a proof $\gamma(p)$ of the theorem $\alpha(\varphi)$ of the theory $\Phi(T)$.

Example 32 Consider the subentailment map $(\Phi, \alpha) : ent(Eqtl) \hookrightarrow ent(Fol^=)$ from the entailment system of equational logic to the entailment system of first order logic with equality described in Example 17. We can associate to the equational entailment system the proof calculus $\mathcal{P}_{Eqtl}$ with $P : \underline{Th_0} \longrightarrow \underline{Cat}$ the functor sending each equational theory $T = (\Sigma, \Gamma)$ to the category $P(T)$ whose objects are Σ-terms, and whose morphisms are chains of elementary steps of equational deduction using equations in Γ. Morphism composition is chain concatenation. The functor $Pr : \underline{Cat} \longrightarrow \underline{Set}$ sends each small category $\mathcal{C}$ to the set of triples $Pr(\mathcal{C}) = \{(A, f, B) \mid f : A \longrightarrow B \ in \ \mathcal{C}\}$. Therefore, the functor $proofs$ sends T to the set of triples of the form (t, p, t') with t, t' Σ-terms and p a chain of elementary steps of equational deduction using the equations in T. The projection function π forgets about the proof p and yields the equation $t = t'$. Consider now a proof calculus $\mathcal{P}_{Fol^=}$ associated to $ent(Fol^=)$. We can for example have a natural deduction proof calculus extended with rules for equality, and give to its proofs the structure of a multicategory. A map of proof calculi $\mathcal{P}_{Eqtl} \longrightarrow \mathcal{P}_{Fol^=}$ compatible with the subentailment $(\Phi, \alpha) : ent(Eqtl) \hookrightarrow ent(Fol^=)$ now consists of a systematic way of translating a chain of elementary equational deductions $p : t \to t'$ using axioms in T into a natural deduction proof $\gamma(t, p, t')$ of the theorem $\alpha(t = t') = (\forall x_1 ... \forall x_n \; t = t')$ for the theory $\Phi(T)$, i.e., in a family of functions,

$$\gamma_T : \mathit{proofs}_{\mathbf{Eqtl}}(T) \longrightarrow \mathit{proofs}_{\mathbf{Fol}^=}(\Phi(T)).$$

Of course, "systematic" means that the maps γ_T should be independent of changes in syntax, i.e., that they should form a natural transformation $\gamma : \mathit{proofs}_{\mathbf{Eqtl}} \Longrightarrow \mathit{proofs}_{\mathbf{Fol}^=}$. The fact that proofs of one theorem go to proofs of its translation by α has a concise expression in terms of natural transformations, as explained below. □

Definition 33 Given proof calculi $\mathcal{P} = (\underline{Sign}, sen, \vdash, P, Pr, \pi)$ and $\mathcal{P}' = (\underline{Sign}', sen', \vdash', P', Pr', \pi')$, a *map of proof calculi* $(\Phi, \alpha, \gamma) : \mathcal{P} \longrightarrow \mathcal{P}'$ consists of a map $(\Phi, \alpha) : ent(\mathcal{P}) \longrightarrow ent(\mathcal{P}')$ of the underlying entailment systems together with a natural transformation $\gamma : \mathit{proofs} \Longrightarrow \mathit{proofs}' \circ \Phi$ such that the following cells are identical:

$$\begin{tikzcd}
\underline{Th_0} \arrow[rr, "sen"] \arrow[rr, bend left=60, "proofs"] \arrow[rr, phantom, bend left=30, "\Downarrow\pi"] \arrow[dr, "\Phi"'] \arrow[rr, phantom, bend right=30, "\Downarrow\alpha"] & & \underline{Set} & = & \underline{Th_0} \arrow[rr, bend left=60, "proofs"] \arrow[rr, phantom, "\Downarrow\gamma"] \arrow[dr, "\Phi"'] & & \underline{Set} \\
& \underline{Th'_0} \arrow[ur, "sen'"'] & & & & \underline{Th'_0} \arrow[ur, "proofs'"] \arrow[ur, bend right=40, "sen'"'] \arrow[ur, phantom, bend right=20, "\Downarrow\pi'"] &
\end{tikzcd}$$

An *embedding*[14] of proof calculi is a map of proof calculi $(\Phi, \alpha, \gamma) : \mathcal{P} \longrightarrow \mathcal{P}'$ such that $(\Phi, \alpha) : ent(\mathcal{P}) \longrightarrow ent(\mathcal{P}')$ is a subentailment system and γ is injective. We write $(\Phi, \alpha, \gamma) : \mathcal{P} \hookrightarrow \mathcal{P}'$ to denote an embedding. □

Another good example of a map of proof calculi is the translation of proofs in the sequent calculus into natural deduction proofs. Basically, a proof in the sequent calculus can be viewed as a set of directions for constructing a natural deduction proof in normal form (see [59], Appendix A).

Given maps of proof calculi $(\Phi_1, \alpha_1, \gamma_1) : \mathcal{P} \longrightarrow \mathcal{P}'$ and $(\Phi_2, \alpha_2, \gamma_2) : \mathcal{P}' \longrightarrow \mathcal{P}''$ we can define their *composition* $(\Phi_2, \alpha_2, \gamma_2) \circ (\Phi_1, \alpha_1, \gamma_1) : \mathcal{P} \longrightarrow \mathcal{P}''$ as the triple $(\Phi_3, \alpha_3, \gamma_3)$ with (Φ_3, α_3) the composition of the underlying maps of entailment systems, and γ_3 the natural transformation obtained by pasting together the two cells,

$$\begin{tikzcd}
\underline{Th_0} \arrow[r, "\Phi_1"] \arrow[dr, "proofs"'] & \underline{Th'_0} \arrow[r, "\Phi_2"] \arrow[d, "proofs'"] \arrow[l, phantom, shift right=3, "\overset{\gamma_1}{\Longrightarrow}"'] & \underline{Th''_0} \arrow[dl, "proofs''"] \arrow[l, phantom, shift right=3, "\overset{\gamma_2}{\Longrightarrow}"'] \\
& \underline{Set} &
\end{tikzcd}$$

[14]Since the term "proof subcalculus" has already been used with a specific technical meaning, I use the term "embedding" instead.

Checking that $(\Phi_3, \alpha_3, \gamma_3)$ is itself a map of proof calculi, and that composition is associative, is an easy cell pasting exercise. Therefore, we have a category $\underline{PCalc}$ having proof calculi as its objects and having maps of proof calculi as its morphisms. There is of course a forgetful functor $ent : \underline{PCalc} \longrightarrow \underline{Ent}$ sending each proof calculus to its underlying entailment system.

Proposition 34 The functor $ent : \underline{PCalc} \longrightarrow \underline{Ent}$ has a right adjoint $(\_)^{\natural} : \underline{Ent} \longrightarrow \underline{PCalc}$.

Proof: Given an entailment system $\mathcal{E} = (\underline{Sign}, sen, \vdash)$, the proof calculus $\mathcal{E}^{\natural}$ is the 6-tuple $\mathcal{E}^{\natural} = (\underline{Sign}, sen, \vdash, thm, 1_{\underline{Set}}, j)$ with thm the functor sending each theory to the set of its theorems described in Section 2.2, and j the natural subfunctor inclusion $thm \subseteq sen$. The unit of the adjunction for a proof calculus $\mathcal{P} = (\underline{Sign}, sen, \vdash, P, Pr, \pi)$ is the map $(1_{\underline{Sign}}, 1_{sen}, \pi) : \mathcal{P} \longrightarrow (ent(\mathcal{P}))^{\natural}$, where π is now viewed as a natural transformation $\pi : proofs \Longrightarrow thm$. $\Box$

Since proof subcalculi generalize proof calculi, the above definition of a map of proof calculi generalizes to the following definition.

Definition 35 Given proof subcalculi $\mathcal{P} = (\underline{Sign}, sen, \vdash, \underline{Sign}_0, ax, concl, P, Pr, \pi)$ and $\mathcal{P}' = (\underline{Sign}', sen', \vdash', \underline{Sign}'_0, ax', concl', P', Pr', \pi')$, a *map of proof subcalculi* $(\Phi, \alpha, \gamma) : \mathcal{P} \longrightarrow \mathcal{P}'$ consists of a map $(\Phi, \alpha) : ent(\mathcal{P}) \longrightarrow ent(\mathcal{P}')$ of the underlying entailment systems such that the functor $\Phi : \underline{Th_0} \longrightarrow \underline{Th'_0}$ restricts to a functor[15] $\Phi : \underline{Th_{ax}} \longrightarrow \underline{Th'_{ax'}}$ together with a natural transformation $\gamma : proofs \Longrightarrow proofs' \circ \Phi$ such that the following cells are identical:

$$\begin{tikzcd} \underline{Th_{ax}} \arrow[rr, bend left=60, "proofs"] \arrow[rr, "sen"] \arrow[rr, phantom, bend left=30, "\Downarrow \pi"] \arrow[dr, "\Phi"'] \arrow[rr, phantom, bend right=30, "\Downarrow \alpha"] & & \underline{Set} \\ & \underline{Th'_{ax'}} \arrow[ur, "sen'"'] & \end{tikzcd} = \begin{tikzcd} \underline{Th_{ax}} \arrow[rr, bend left=60, "proofs"] \arrow[rr, phantom, "\Downarrow \gamma"] \arrow[dr, "\Phi"'] & & \underline{Set} \\ & \underline{Th'_{ax'}} \arrow[ur, "proofs'"] \arrow[ur, bend right=40, "sen'"'] \arrow[ur, phantom, bend right=20, "\Downarrow \pi'"] & \end{tikzcd}$$

An *embedding* of proof subcalculi is a map of proof subcalculi $(\Phi, \alpha, \gamma) : \mathcal{P} \longrightarrow \mathcal{P}'$ such that $(\Phi, \alpha) : ent(\mathcal{P}) \longrightarrow ent(\mathcal{P}')$ is a subentailment system and γ is injective. We write $(\Phi, \alpha, \gamma) : \mathcal{P} \hookrightarrow \mathcal{P}'$ to denote an embedding. $\Box$

Composition of maps of proof subcalculi is defined as for proof calculi, and we get a category $\underline{PSCalc}$ with objects proof subcalculi and morphisms maps of proof subcalculi. We then have a forgetful functor $ent : \underline{PSCalc} \longrightarrow \underline{Ent}$ sending each proof subcalculus to its underlying entailment system.

By replacing everywhere the category $\underline{Set}$ by the category $\underline{Space}$ and considering effective proof calculi as objects we can define in an entirely similar way the notion of an *effective* map of proof calculi. This gives us a category $\underline{EffPSCalc}$ and a forgetful functor $\mathcal{U} : \underline{EffPSCalc} \longrightarrow \underline{PSCalc}$.

[15] Intuitively, this says that Φ transforms theories using only $\underline{Sign}_0$ signatures and ax axioms into theories that only use $\underline{Sign}'_0$ signatures and ax' axioms.

4.5 Mapping Logical Systems

We can now gather everything together and obtain notions of a map of logical systems and a map of logical subsystems.

Definition 36 Given logical sytems $S = (\underline{Sign}, sen, \underline{Mod}, \vdash, \models, P, Pr, \pi)$ and $S' = (\underline{Sign}', sen', \underline{Mod}', \vdash', \models', P', Pr', \pi')$ a *map of logical systems* $(\Phi, \alpha, \beta, \gamma) : S \longrightarrow S'$ consists of a functor Φ and natural transformations α, β, γ such that:

1. $(\Phi, \alpha, \beta) : log(S) \longrightarrow log(S')$ is a map of the underlying logics, and
2. $(\Phi, \alpha, \gamma) : pcalc(S) \longrightarrow pcalc(S')$ is a map of the underlying proof calculi.

This defines a category $\underline{LogSys}$ whose objects are logical systems and whose morphisms are maps of logical systems, and we have forgetful functors $log : \underline{LogSys} \longrightarrow \underline{Log}$ and $pcalc : \underline{LogSys} \longrightarrow \underline{PCalc}$.

An *embedding* of logical systems is a map $(\Phi, \alpha, \beta, \gamma) : S \longrightarrow S'$ of logical systems such that (Φ, α, β) is a sublogic and (Φ, α, γ) is an embedding of proof calculi. We write $(\Phi, \alpha, \beta, \gamma) : S \hookrightarrow S'$ to denote an embedding of logical systems. □

The following result is entirely analogous to Proposition 34 and is left as an exercise.

Proposition 37 The functor $ent : \underline{LogSys} \longrightarrow \underline{Log}$ has a right adjoint $(\_)^{\natural} : \underline{Log} \longrightarrow \underline{LogSys}$. □

Maps of logical subsystems are defined in an entirely analogous manner.

Definition 38 Given logical subsystems $S = (\underline{Sign}, sen, \underline{Mod}, \vdash, \models, \underline{Sign}_0, ax, concl, P, Pr, \pi)$ and $S' = (\underline{Sign}', sen', \underline{Mod}', \vdash', \models', \underline{Sign}_0', ax', concl', P', Pr', \pi')$ a *map of logical subsystems* $(\Phi, \alpha, \beta, \gamma) : S \longrightarrow S'$ consists of a functor Φ and natural transformations α, β, γ such that:

1. $(\Phi, \alpha, \beta) : log(S) \longrightarrow log(S')$ is a map of the underlying logics, and
2. $(\Phi, \alpha, \gamma) : pscalc(S) \longrightarrow pscalc(S')$ is a map of the underlying proof subcalculi.

This defines a category $\underline{LogSSys}$ whose objects are logical subsystems and whose morphisms are maps of logical subsystems, and we have forgetful functors $log : \underline{LogSSys} \longrightarrow \underline{Log}$ and $pscalc : \underline{LogSSys} \longrightarrow \underline{PSCalc}$.

An *embedding* of logical subsystems is a map $(\Phi, \alpha, \beta, \gamma) : S \longrightarrow S'$ of logical subsystems such that (Φ, α, β) is a sublogic and (Φ, α, γ) is an embedding of proof subcalculi. We write $(\Phi, \alpha, \beta, \gamma) : S \hookrightarrow S'$ to denote an embedding of logical subsystems. □

We can summarize the relationships between the different categories of entailment systems, institutions, logics, proof (sub)calculi and logical (sub)systems by the following commutative diagram, where for simplicity only the forgetful functors are included. The several adjoints already described as well as the category of effective proof subcalculi are omitted, but they should be kept in mind to obtain a more complete summary.

$$\begin{tikzcd}
& \underline{\mathit{LogSys}} \arrow[rr, hook] \arrow[drr, "\mathit{pcalc}"] \arrow[ddr, "\mathit{log}"'] & & \underline{\mathit{LogSSys}} \arrow[ddl, "\mathit{log}"] \arrow[drr, "\mathit{pscalc}"] & & \\
& & & \underline{\mathit{PCalc}} \arrow[rr, hook] \arrow[ddr, "\mathit{ent}"] & & \underline{\mathit{PSCalc}} \arrow[ddl, "\mathit{ent}"] \\
& & \underline{\mathit{Log}} \arrow[dl, "\mathit{inst}"'] \arrow[drr, "\mathit{ent}"] & & & \\
& \underline{\mathit{Inst}} & & & \underline{\mathit{Ent}} &
\end{tikzcd}$$

5 Categorical Logics

Categorical logics give us great model-theoretic flexibility, since their models are not restricted to the traditional set-theoretic structures with functions, predicates, etc., that are assumed as basic even in axiomatic approaches such as abstract model theory. In fact, the question "What is a model?" is far from settled for the many higher order logics of interest to logicians and computer scientists. Some of the proposed models have a somewhat *ad hoc* character, and may fail to reflect adequately the basic intuitions. Nevertheless, models are essential to semantic understanding. Category theory, and in particular the categorical approach to logic originating in the wide and seminal work of F.W. Lawvere, has much to offer in this regard. This is probably quite widely recognized and in some cases, such as the relationship between the typed λ-calculus and cartesian closed categories, or between intuitionistic set theory and topos theory, well understood. However, the potential of the categorical point of view has yet to be fully exploited.

There is no better way to begin our discussion of categorical logics than by quoting some words from a fundamental paper published by F.W. Lawvere in 1969 [46]. The paper begins with the following words:

> "That pursuit of exact knowledge which we call mathematics seems to involve in an essential way two dual aspects, which we may call the Formal and the Conceptual. For example, we manipulate algebraically a polynomial equation and visualize geometrically the corresponding curve. Or we concentrate in one moment on the deduction of theorems from the axioms of group theory, and in the next consider the classes of actual groups to which the theorems refer. Thus the Conceptual is in a certain sense the subject matter of the Formal."

and ends with the following paragraph:

> "Finally, in Foundations there is the familiar Galois connection between sets of axioms and classes of models, for a fixed set of relation variables R_i. Globalizing to an adjoint pair allows making precise the semantical effect, not only of increasing the axioms, but also of omitting some relation

symbols or reinterpreting them, in a unified way. And if we deal with *categories* of models, allows the latter to determine their own full sets of natural relation variables, thus giving definability theory a new significance outside the realm of axiomatic classes. To do this for a given species — equational, elementary, higher-order, etc.— of, say, I-sorted theories, one defines an adjoint situation

$$\begin{tikzcd} \mathit{Theories}^{op} \arrow[r, shift left, "\mathit{semantics}"] & (\mathit{Cat}, [\mathit{Sets}^I]) \arrow[l, shift left, "\mathit{structure}"] \end{tikzcd}$$

in which the right hand side denotes a category whose morphisms are commutative triangles

$$\begin{tikzcd} C \arrow[rr] \arrow[dr] & & C' \arrow[dl] \\ & \mathit{Sets}^I & \end{tikzcd}$$

of functors with C and C' more or less arbitrary categories. The invariant notion of theory here appropriate has, in all cases considered by the author, been expressed most naturally by identifying a theory T itself with a category of a certain sort, in which case the semantics (category of Models) of T is a certain subcategory of the category of functors $T \longrightarrow \mathit{Sets}$. There is then a further adjoint situation

$$\begin{tikzcd} \mathit{Formal} \arrow[r, shift left] & \mathit{Theories} \arrow[l, shift left] \end{tikzcd}$$

describing the presentation of the invariant theories by means of the formalized languages appropriate to the species. Composing this with above, and tentatively identifying the Conceptual with categories of the general sort $(\mathit{Cat}, [\mathit{Sets}^I])$, we arrive at a family of adjoint situations

$$\begin{tikzcd} \mathit{Formal}^{op} \arrow[r, shift left] & \mathit{Conceptual} \arrow[l, shift left] \end{tikzcd}$$

(one for each species of theory) which one may reasonably hope consitute the fragments of a precise description of the duality with which we began our discussion."

The use of the category Sets^I is more an illustration for the case of classical set-theoretic models than a necessary requirement. The version axiomatized below has a category $\underline{\mathcal{I}}$ of categories with structure instead of Sets^I. The relationship with the axiomatic definition of logic presented in Section 2 is as follows:

- The category that Lawvere calls "*Formal*" coincides with the category $\underline{Th_0}$ for a logic $\mathcal{L}$ in our sense.

- A *theory* in Lawvere's sense is a category $\mathcal{C}$ with a certain structure. That structure is meant to capture the essential aspects of a logic $\mathcal{L}$, so that the category $\mathcal{C}$ can be understood as an abstract "theory" that is independent of both a choice of syntax for $\mathcal{L}$ and a particular presentation of the axioms. In fact, such categories can be viewed both as abstract "theories," and as "generic" models. For example, a typed lambda calculus theory (Σ, E) generates a free cartesian closed category $\mathcal{F}(\Sigma, E)$, which is the abstract theory in Lawvere's sense, and also the generic "term" model of the theory (Σ, E).

- To avoid confusion between concrete and abstract theories, I identify Lawvere's category *Theories* with a category $\underline{\mathcal{T}}$ whose objects are categories with some additional structure, and whose morphisms are functors preserving that structure. For example, in the case of the typed λ-calculus, $\underline{\mathcal{T}} = \underline{\mathcal{CCCat}}$, the category of cartesian closed categories, with morphisms functors that strictly preserve the cartesian closed structure [43].

Example 39 (General Equational Logic) Equational logic was the first instance of a categorical logic considered by Lawvere in his doctoral dissertation [45]. Lawvere restricted his analysis to classical set-theoretic models. Given an equational theory (Σ, E), he exhibited a category with finite products $\mathcal{F}(\Sigma, E)$ such that Σ-algebras A that satisfy the equations E can be put into 1-1 correspondence with functors $\tilde{A} : \mathcal{F}(\Sigma, E) \rightarrow \underline{Set}$ that strictly preserve products; i.e., chosen products in $\mathcal{F}(\Sigma, E)$ are mapped to cartesian products in $\underline{Set}$.

The category $\mathcal{F}(\Sigma, E)$ is easy to describe. Its objects are the natural numbers. A morphism $[t] : n \longrightarrow 1$ is the equivalence class modulo the equations E of a Σ-term t whose variables are among $x_1, ..., x_n$. A morphism $n \longrightarrow m$ is an m-tuple of morphisms $n \longrightarrow 1$. Morphism composition is term substitution. For example, $[x_2 + x_1] \circ ([x_7 * x_3], [x_4 + x_5]) = [(x_4 + x_5) + (x_7 * x_3)]$. It is then easy to see that the object n is the n^{th} product of the object 1 with projections $[x_1], ..., [x_n]$; and, more generally, that the product of the objects n and m is $n + m$. The functor $\tilde{A}$ associated to the algebra A sends the morphism $[t] : n \longrightarrow 1$ to the derived operation $A^n \longrightarrow A$ associated to the term t. Under this correspondence between algebras and functors, an equation $t = t'$ is satisfied by a (Σ, E)-algebra A iff $\tilde{A}([t]) = \tilde{A}([t'])$. The analogous case of many-sorted equational logic was studied by Bénabou in his thesis [4]. The category $\mathcal{F}(\Sigma, E)$ is constructed as in the unsorted case, but now it has as its set of objects the free monoid S^* generated by the set S of sorts.

In the many-sorted case, we can view the construction of $\mathcal{F}(\Sigma, E)$ as a functor $\mathcal{F} : \underline{Th_0} \longrightarrow \underline{\times Cat}$ where the theories in $\underline{Th_0}$ are many-sorted equational theories, $\underline{\times Cat}$ is a category whose objects are small[16] categories with chosen finite products and whose morphisms are functors that strictly preserve the chosen finite products. The functor $\mathcal{F}$ is the left adjoint of a functor $\mathcal{U} : \underline{\times Cat} \longrightarrow \underline{Th_0}$ that associates to each category with (chosen) finite products $\mathcal{C}$ the many-sorted equational theory $\mathcal{U}(\mathcal{C})$ whose set of sorts is the set $|\mathcal{C}|$ of objects of $\mathcal{C}$, whose ranked set of operations has as constants of sort A the morphisms $a : 1 \longrightarrow A$, with 1 the chosen final object, as unary operations of type $A \longrightarrow B$ the morphisms $f : A \longrightarrow B$ in $\mathcal{C}$, and as n-ary operations of type

[16]I will not worry much about foundations; later examples will also involve "large" categories. All concerns can be resolved using universes.

$A_1 \dots A_n \longrightarrow C$, for $n > 1$, symbols $f_{A_1 \dots A_n}$, one for each $f : B = A_1 \times \dots \times A_n \longrightarrow C$ in $\mathcal{C}$ as well as new operations $\nu_{A_1 \dots A_n} : A_1 \dots A_n \longrightarrow B$ for each $B = A_1 \times \dots \times A_n$. The equations of $\mathcal{U}(\mathcal{C})$ include the equations $\nu_{A_1 \dots A_n}(\pi_{A_1}(x), \dots, \pi_{A_n}(x)) = x$, for $\pi_{A_i} : A_1 \times \dots \times A_n \longrightarrow A_i$ the chosen i^{th} projection in $\mathcal{C}$, the equations $f_{A_1 \dots A_n}(x_1, \dots, x_n) = f(\nu_{A_1 \dots A_n}(x_1, \dots, x_n))$, and all other equations satisfied when interpreting the $\nu_{A_1 \dots A_n}$'s as identitities and the f's and $f_{A_1 \dots A_n}$'s by their corresponding morphisms in $\mathcal{C}$. For example, if $h : C \longrightarrow D$ with $D = A \times B$ is the unique morphism in $\mathcal{C}$ induced by morphisms $f : C \longrightarrow A$, and $g : C \longrightarrow B$, then we have an equation $\nu_{A,B}(f(x), g(x)) = h(x)$.

The great conceptual advantage of viewing a T-algebra as a product preserving functor $\mathcal{F}(T) \longrightarrow \underline{Set}$ is that the concept generalizes immediately to that of a T-algebra in *any* category with finite products. Thus, for T_{Grp} the theory of groups, a topological group can be regarded as a product preserving functor $\mathcal{F}(T_{Grp}) \longrightarrow \underline{Top}$ landing in the category $\underline{Top}$ of topological spaces, and a sheaf of groups on a topological space X can be viewed as a product preserving functor $\mathcal{F}(T_{Grp}) \longrightarrow \underline{Sheaves(X)}$ landing in the category $\underline{Sheaves(X)}$ of sheaves on X. Therefore, we can in general define a *group* in a category with finite products $\mathcal{C}$ as a product preserving functor $\mathcal{F}(T_{Grp}) \longrightarrow \mathcal{C}$. Since $\mathcal{F}(T_{Grp})$ is also a category with finite products, we can consider the group $1_{\mathcal{F}(T_{Grp})} : \mathcal{F}(T_{Grp}) \longrightarrow \mathcal{F}(T_{Grp})$, which can be understood as the *generic* group. By construction, this group satisfies the equation $t = t'$ iff $[t] = [t']$, iff the equation $t = t'$ is a theorem of group theory. By using product preserving functors, we can even relate groups in different categories. For example, we can relate a sheaf of goups $\mathcal{G} : \mathcal{F}(T_{Grp}) \longrightarrow \underline{Sheaves(X)}$ to its group of global sections $\Gamma(\mathcal{G})$ by composing with the global sections functor $\Gamma(\_) : \underline{Sheaves(X)} \longrightarrow \underline{Set}$, which can therefore be understood as a homomorphism[17] between those two groups. Thus, we can structure all the possible groups in all possible categories and the product preserving functors that relate them as the slice category $\mathcal{F}(T_{Grp})/\underline{\times Cat}$. □

Definition 40 A logic $\mathcal{L}$ is called a *categorical logic* on $\underline{\mathcal{T}}$ if there is a category $\underline{\mathcal{T}}$ with pushouts and with a faithful functor $\underline{\mathcal{T}} \longrightarrow \underline{Cat}$ such that:

1. There are functors $\mathcal{U} : \underline{\mathcal{T}} \longrightarrow \underline{Th_0}$ and $\mathcal{F} : \underline{Th_0} \longrightarrow \underline{\mathcal{T}}$ with $\mathcal{F}$ left adjoint to $\mathcal{U}$.
2. The functor $\underline{Mod} : \underline{Th_0}^{op} \longrightarrow \underline{Cat}$ is naturally isomorphic[18] to the functor
$$\underline{Th_0}^{op} \xrightarrow{\mathcal{F}^{op}} \underline{\mathcal{T}}^{op} \xrightarrow{\_/\underline{\mathcal{T}}} \underline{Cat},$$
where the functor $\_/\underline{\mathcal{T}}$ sends an object[19] $\mathcal{C} \in \underline{\mathcal{T}}$ to the slice category $\mathcal{C}/\underline{\mathcal{T}}$.
3. For any theory $T = (\Sigma, \Gamma)$ and sentence $\varphi \in sen(\Sigma)$ we have:
$$\Gamma \vdash_\Sigma \varphi \Leftrightarrow 1_{\mathcal{F}(T)} \models_\Sigma \varphi.$$

□

[17] A more general notion of homomorphism that specializes to the usual one when all the groups are in the category $\underline{Set}$ is discussed in Remark (2) at the end of this section.

[18] To simplify the discussion, in what follows I ignore this isomorphism and identify $\underline{Mod}(T)$ with $\mathcal{F}(T)/\underline{\mathcal{T}}$.

[19] We think of $\mathcal{C}$ as a category with a certain structure, and of $\underline{\mathcal{T}}$ as the category of all categories with that type of structure.

Example 41 The general equational logic example where $\underline{\mathcal{I}} = \underline{\times Cat}$ has already been discussed in detail. Here are a few other additional examples:

1. $\mathcal{L}$ the logic of the typed lambda calculus, where $\underline{\mathcal{I}} = \underline{CCCat}$ is the category of cartesian closed categories; see [43] and Section 6.2.
2. $\mathcal{L}$ higher order intuitionistic logic, where $\underline{\mathcal{I}} = \underline{Toposes}$ is the category of elementary Lawvere-Tierney toposes; see [6].
3. $\mathcal{L}$ the logic of Martin-Lõf type theory with equality types, where $\underline{\mathcal{I}} = \underline{LCCCat}$ is the category of locally cartesian closed categories; see [65] and Section 6.2.
4. $\mathcal{L}$ the logic of Martin-Lõf type theory without equality types, where $\underline{\mathcal{I}}$ is either Cartmell's category of contextual categories, or the category $\underline{RCCCat}$ of relatively cartesian closed categories; see [10,41].
5. $\mathcal{L}$ the logic of the Girard-Reynolds polymorphic lambda calculus [21,60], where $\underline{\mathcal{I}} = \underline{PLCat}$ is Seely's category of PL-categories; see [63] and Section 6.2.
6. $\mathcal{L}$ the logic of the Girard-Reynolds polymorphic lambda calculus [21,60], where $\underline{\mathcal{I}} = \underline{RCCCat}$ is the category of relatively cartesian closed categories; see [52] and Section 6.2.
7. $\mathcal{L}$ Girard's linear logic [22], where $\underline{\mathcal{I}}$ is Seely's category of linear categories; see [64,56], and the related [14].

□

Categorical logics have very nice model-theoretic properties indeed, as expressed in the following theorem.

Theorem 42 Any categorical logic $\mathcal{L}$ is complete, liberal, exact, and admits initial models.

Proof: Completeness is clear from condition (3), since we have

$$\Gamma \models_\Sigma \varphi \Longrightarrow 1_{\mathcal{F}(\Sigma,\Gamma)} \models_\Sigma \varphi \Longrightarrow \Gamma \vdash_\Sigma \varphi.$$

For any $C \in \underline{\mathcal{I}}$, the map $1_C : C \rightarrow C$ is obviously an initial object in the slice category $C/\underline{\mathcal{I}}$. Therefore, $1_{\mathcal{F}(T)}$ is an initial object in $\mathcal{F}(T)/\underline{\mathcal{I}}$ and as a consequence $\underline{Mod}(T)$ has an initial model.

Liberality is a direct consequence of the following well known lemma.

Lemma 43 Let $\mathcal{A}$ be a category with pushouts, and $f : C \rightarrow C'$ a morphism in $\mathcal{A}$. Then, the "composition along f" functor $f/\mathcal{A} : C'/\mathcal{A} \longrightarrow C/\mathcal{A}$ mapping each $h : C' \rightarrow D$ to $h \circ f : C \rightarrow D$ has a left adjoint given by "pushout along f" that maps each $g : C \rightarrow E$ to the map $C' \rightarrow E'$ in the pushout diagram

$$\begin{tikzcd}
E \arrow[r, "f'"] & E' \\
C \arrow[u, "g"] \arrow[r, "f"'] & C' \arrow[u, "g'"']
\end{tikzcd}$$

□

For exactness, notice that, since $\mathcal{F} : \underline{Th_0} \longrightarrow \underline{\mathcal{T}}$ is a left adjoint, by duality $\mathcal{F}^{op} : \underline{Th_0}^{op} \longrightarrow \underline{\mathcal{T}}^{op}$ is a right adjoint. Since right adjoints preserve limits (see [47], Theorem V.5.1) and $\underline{Mod} \cong (\_/\underline{\mathcal{T}}) \circ \mathcal{F}^{op}$, we only have to show that $\_/\underline{\mathcal{T}}$ preserves finite limits. This follows from the easy lemma below.

Lemma 44 For any category $\mathcal{A}$, the functor $\_/\mathcal{A} : \mathcal{A}^{op} \longrightarrow \underline{Cat}$ preserves pullbacks.
□

□

In the passage of Lawvere's paper [46] cited above, Lawvere mentions a structure-semantics adjointness result. This result appeared in his thesis [45] for the case of algebras on the category of sets and has since then been generalized in many directions. However, I am not aware of other formulations with the degree of generality of Theorem 45 below.

Notice that the functor $\_/\underline{\mathcal{T}} : \underline{\mathcal{T}}^{op} \longrightarrow \underline{Cat}$ factors as:

$$\underline{\mathcal{T}}^{op} \xrightarrow{sem} \underline{Cat}//\underline{\mathcal{T}} \longrightarrow \underline{Cat},$$

where:

- $\underline{Cat}//\underline{\mathcal{T}}$ is the full subcategory of the slice category $\underline{Cat}/\underline{\mathcal{T}}$ given by those functors $\underline{\mathcal{A}} \longrightarrow \underline{\mathcal{T}}$ such that $\underline{\mathcal{A}}$ has an initial object $I_{\underline{\mathcal{A}}}$ that we assume chosen once and for all;
- the functor sem sends C to the projection functor $C/\underline{\mathcal{T}} \longrightarrow \underline{\mathcal{T}} : (C \longrightarrow D) \mapsto D$, and
- the functor $\underline{Cat}//\underline{\mathcal{T}} \longrightarrow \underline{Cat}$ is the projection functor $(\underline{\mathcal{A}} \longrightarrow \underline{\mathcal{T}}) \mapsto \underline{\mathcal{A}}$.

Theorem 45 For $\mathcal{L}$ a categorical logic on $\underline{\mathcal{T}}$, the functor $sem : \underline{\mathcal{T}}^{op} \longrightarrow \underline{Cat}//\underline{\mathcal{T}}$ is full and faithful, and has a left adjoint $str : \underline{Cat}//\underline{\mathcal{T}} \longrightarrow \underline{\mathcal{T}}^{op}$.

Proof: The functor sem is clearly faithful since, given $f, f' : C \to C'$ in $\underline{\mathcal{T}}$ with $f \neq f'$, we have $(f/\underline{\mathcal{T}})(1_{C'}) = f \neq f' = (f'/\underline{\mathcal{T}})(1_{C'})$.

To see that it is full, let $H : C'/\underline{\mathcal{T}} \longrightarrow C/\underline{\mathcal{T}}$ be a functor such that

$$\begin{tikzcd} C'/\underline{\mathcal{T}} \arrow[rr, "H"] \arrow[dr] & & C/\underline{\mathcal{T}} \arrow[dl] \\ & \underline{\mathcal{T}} & \end{tikzcd}$$

commutes. Then H has to send the object $1_{C'}$ to a map $f : C \to C'$ in $\underline{\mathcal{T}}$. We claim that $H = f/\underline{\mathcal{T}}$. Indeed, since H commutes the triangle, and there is a morphism $g : 1_{C'} \to g$ in $C'/\underline{\mathcal{T}}$ for any $g : C' \to D$, we must have a morphism $H(g) = g : f \to H(g)$ in $C/\underline{\mathcal{T}}$ and therefore $H(g) = g \circ f$, so that H and $f/\underline{\mathcal{T}}$ coincide on the objects. H and $f/\underline{\mathcal{T}}$ coincide trivially on the morphisms, since for any $h : g \to g'$ in $C'/\underline{\mathcal{T}}$ we must have $H(h) = h = (f/\underline{\mathcal{T}})(h)$, because both H and $f/\underline{\mathcal{T}}$ commute the triangle.

The left adjoint $str : \underline{Cat}//\underline{T} \longrightarrow \underline{T}^{op}$ sends an object $A : \underline{A} \rightarrow \underline{T}$ to the object $A(I_{\underline{A}}) \in \underline{T}$, and a morphism $H : A \rightarrow B$ to the morphism $B(h) : B(I_{\underline{B}}) \rightarrow A(I_{\underline{A}})$ in $\underline{T}$, for $h : I_{\underline{B}} \rightarrow H(I_{\underline{A}})$ the unique morphism in $\underline{B}$; functoriality then follows from the initiality of each $I_{\mathcal{X}}$ in its category $\mathcal{X}$.

The functor $A : \underline{A} \longrightarrow \underline{T}$ factors through $A(I_{\underline{A}})/\underline{T}$ as

$$\underline{A} \xrightarrow{\eta_A} A(I_{\underline{A}})/\underline{T} \longrightarrow \underline{T}$$

with $\eta_A(X) = A(h) : A(I_{\underline{A}}) \rightarrow A(X)$, for $h : I_{\underline{A}} \rightarrow X$ the unique morphism in $\underline{A}$, and $\eta_A(g) = A(g)$ for $g : X \rightarrow Y$. This yields our desired map $\eta_A : A \longrightarrow sem(str(A))$.

Let now $D : A \longrightarrow sem(C)$ be a morphism in $\underline{Cat}//\underline{T}$ and assume that there is a morphism $\overline{D} : sem(str(A)) \longrightarrow sem(C)$ such that $\overline{D} \circ \eta_A = D$. Then, since we have shown that sem is full and faithful, we must have $\overline{D} = f/\underline{T} : A(I_{\underline{A}})/\underline{T} \longrightarrow C/\underline{T}$ for a unique $f : C \rightarrow A(I_{\underline{A}})$. But the equation $\overline{D} \circ \eta_A = D$ forces

$$\overline{D}(\eta_A(I_{\underline{A}})) = \overline{D}(1_{A(I_{\underline{A}})}) = (f/\underline{T})(1_{A(I_{\underline{A}})}) = f = D(I_{\underline{A}})$$

and makes $\overline{D}$, if it exists, unique, namely, $\overline{D} = D(I_{\underline{A}})/\underline{T}$. Since $D(I_{\underline{A}})/\underline{T}$ is the identity on morphisms, and D sends $g : X \rightarrow Y$ in $\underline{A}$ to $A(g) : D(X) \rightarrow D(Y)$ with $D(Y) = A(g) \circ D(X)$, to show that indeed $D(I_{\underline{A}})/\underline{T} \circ \eta_A = D$ it is enough to check it on the objects. But, for any $X \in \underline{A}$, we have

$$D(X) = A(h) \circ D(I_{\underline{A}}) = \eta_A(X) \circ D(I_{\underline{A}}) = (D(I_{\underline{A}})/\underline{T})(\eta_A(X)),$$

for $h : I_{\underline{A}} \rightarrow X$ the unique morphism in $\underline{A}$. □

Corollary 46 For any C in $\underline{T}$ there is a natural isomorphism $C \cong str(sem(C))$.

Proof: This follows directly from sem full and faithful right adjoint; see [47], Theorem IV.3.1. □

The structure-semantics adjointness theorem makes clear why Lawvere calls $\underline{T}$ the category of (abstract) theories. We can think of $\mathcal{F}(T)$ as the abstract, presentation independent, theory specified by the presentation T. Indeed, we can establish an equivalence relation among theories $T, T' \in \underline{Th_0}$ by defining $T \equiv T'$ iff there is an isomorphism:

$$\begin{tikzcd} \mathcal{F}(T)/\underline{T} \arrow[rr, "\cong"] \arrow[dr] & & \mathcal{F}(T')/\underline{T} \arrow[dl] \\ & \underline{T} & \end{tikzcd}$$

that is, $T \equiv T'$ iff the corresponding categories of models are isomorphic in a way that is consistent with their projection functors to $\underline{T}$. We then have,

Corollary 47 $T \equiv T'$ iff $\mathcal{F}(T) \cong \mathcal{F}(T')$.

Proof: The "if" part is clear. For the "only if" part, let $T \equiv T'$. Then $sem(\mathcal{F}(T)) \cong sem(\mathcal{F}(T'))$, and applying the functor str we get

$$\mathcal{F}(T) \cong str(sem(\mathcal{F}(T))) \cong str(sem(\mathcal{F}(T'))) \cong \mathcal{F}(T'),$$

as desired. □

Remarks:

1. In some instances, the class of models is restricted by restricting the class of categories $\mathcal{C}$ on which a model $M : \mathcal{F}(T) \longrightarrow \mathcal{C}$ of a theory T can land. For example, in the original treatment of equational logic given by Lawvere [45], the category $\mathcal{C}$ must be the category $\underline{Set}$. More generally, one could restrict the class of categories by requiring that they belong to the image of a functor $V : \underline{\mathcal{W}} \longrightarrow \underline{\mathcal{T}}$. For example, all topos models of the polymorphic lambda calculus are obtained by restricting the corresponding relatively cartesian closed categories to be toposes, i.e., to belong to the image of the forgetful functor $V : \underline{Toposes} \longrightarrow \underline{RCCCat}$ [52]. Therefore, given a functor $V : \underline{\mathcal{W}} \longrightarrow \underline{\mathcal{T}}$ and a categorical logic $\mathcal{L}$ on $\underline{\mathcal{T}}$, we can define the *V-restriction* $\mathcal{L}|_V$ of $\mathcal{L}$ to V as the logic with same entailment system as $\mathcal{L}$ and such that, for T a theory, $\underline{Mod}(T)$ is the "comma category" [47] $\mathcal{F}(T)/V$ whose objects are pairs $(M : \mathcal{F}(T) \longrightarrow V(\mathcal{D}), \mathcal{D})$, with $M : \mathcal{F}(T) \longrightarrow V(\mathcal{D})$ in $\underline{\mathcal{T}}$ and $\mathcal{D} \in \underline{\mathcal{W}}$, and with morphisms $H : (M : \mathcal{F}(T) \longrightarrow V(\mathcal{D}), \mathcal{D}) \longrightarrow (M' : \mathcal{F}(T) \longrightarrow V(\mathcal{D}'), \mathcal{D}')$ morphisms $H : \mathcal{D} \longrightarrow \mathcal{D}'$ in $\underline{\mathcal{W}}$ such that $V(H) \circ M = M'$. Satisfaction is defined as before, i.e., $(M, \mathcal{D}) \models \varphi$ in $\mathcal{L}|_V$ iff $M \models \varphi$ in $\mathcal{L}$. This notion of V-restriction includes the case when $\mathcal{C}$ is constrained to be just one category: in that case, we take as our V the functor from the one morphism category 1 to $\underline{\mathcal{T}}$ that picks up the category $\mathcal{C}$. For any "restriction functor" V there is an associated map of logics $\mathcal{L} \longrightarrow \mathcal{L}|_V$, called its *restriction* map. The case when $V : \underline{\mathcal{W}} \longrightarrow \underline{\mathcal{T}}$ has a left adjoint K is particularly interesting, since then, the comma category $\mathcal{F}(T)/V$ is isomorphic to the slice category $K(\mathcal{F}(T))/\underline{\mathcal{W}}$. Therefore, if the unit map $\eta_{\mathcal{F}(T)} : \mathcal{F}(T) \longrightarrow V(K(\mathcal{F}(T)))$ is such that $\Gamma \vdash \varphi \Leftrightarrow \eta_{\mathcal{F}(T)} \models \varphi$, then it follows easily that the logic $\mathcal{L}|_V$ is in fact a categorical logic on $\underline{\mathcal{W}}$.

2. The definition of categorical logic given above is satisfactory and general for the models. Such models are functors of the form $M : \mathcal{F}(T) \longrightarrow \mathcal{C}$, for T the theory in question, that satisfy the additional properties of morphisms in $\underline{\mathcal{T}}$. However, the notion is too restrictive for homomorphisms. The only homomorphisms permitted between two models $M : \mathcal{F}(T) \longrightarrow \mathcal{C}$ and $M' : \mathcal{F}(T) \longrightarrow \mathcal{C}'$ are functors $H : \mathcal{C} \longrightarrow \mathcal{C}'$ in $\underline{\mathcal{T}}$ such that $M' = H \circ M$. Consider the equational logic case already discussed in Example 39, where ordinary Σ-algebras A satisfying equations E were placed in a 1-1 correspondence with product-preserving functors $\tilde{A} : \mathcal{F}(\Sigma, E) \longrightarrow \underline{Set}$. Under such correspondence, Σ-homomorphisms $f : A \longrightarrow B$ can be put into 1-1 correspondence with natural transformations $\tilde{f} : \tilde{A} \Longrightarrow \tilde{B}$. Therefore, in order to give a full account of homomorphisms we should allow for natural transformations in our definition. The point is that $\underline{\mathcal{T}}$ should be not just a category, but a 2-category [47], and the forgetful functor $\underline{\mathcal{T}} \longrightarrow \underline{Cat}$ should be a 2-functor. This leads to the definition of a *2-categorical logic*. The details of this definition will be given elsewhere.

6 Axiomatizing Logic Programming

What does programming in a logic mean? We can begin to answer this question by stating informally some of the requirements that a logic programming language should satisfy. I call the view represented below the "weak" view.

> **Weak Logic Programming.** A program P in a logic programming language is a theory in a logic $\mathcal{L}$. After entering the program P into the machine, the user can ask questions about his/her program. Such questions, called *queries*, belong to a specified class of sentences in the language of P. When the user submits a query φ, if it is the case that φ is a provable consequence of the axioms in P, then the machine will return a set of *answers* justifying the truth of φ. We can view each of these answers as different *proofs* of the truth of φ; such "proofs" may reasonably omit a good part of the information that a completely detailed proof would provide. If the query φ is not provable from P, two things can happen: either the machine stops after a finite amount of time with the answer "failure," or otherwise the machine loops forever. Therefore, two things are made equivalent: computation in the machine, and deduction in the logic.

One should of course add that in some pragmatic sense the implementation in the machine should be reasonably *efficient* so that for a broad enough class of applications the language can in fact be used in practice; otherwise such a system should be better described as a *theorem prover*. We could summarize the weak view with the slogan

$$\text{Computation} = \text{Deduction}.$$

Although this view is probably the most commonly held, I do not take it as primary. The problem with it is that it makes no reference to the *models* that the theory is a linguistic device for. A theory may in principle have many models. However, when solving a particular problem, such as computing a numerical function or sorting a list of names, we usually have a specific model in mind, such as the integers, the real numbers, or the set of all sequences of expressions of a certain kind. Such a model is then the *intended* or *standard* model of the theory, and its conceptual importance is primary; the theory serves only a secondary role as a linguistic device for describing the model. In the logic programming literature, the standard model is referred to as the "closed world" that the program describes. In a wide variety of cases this standard model can be characterized as an *initial* model.

Let us denote by I_P the model intended by our program P. In the context of such a model, the meaning of a query φ acquires a new significance. Our primary interest is not in *truths* that are generally valid for all models. Rather, our interest is in the *facts* that are true about our model. In other words, we are primarily interested in the *satisfaction* of the query φ by the model I_P and only secondarily in the *provability* of φ from the axioms in P. The theory P is a linguistic device through which such satisfaction may be verified, since if the query is provable, it must be true in all models and therefore it should be a true fact about I_P. The most satisfactory way of exploiting provability as a method of settling facts about our model is to restrict our attention to queries for which, *conversely*, if the query is true in the intended model I_P, then it is provable from P. Otherwise, in cases when the query cannot be proved, we would be left with the doubt as to whether or not it is true in our model. As we shall see, this is a widely exploited property that I call "query completeness." It leads us to the following stronger requirements for a logic programming language,

Strong Logic Programming. A program P in a logic programming language is a theory in a logic $\mathcal{L}$. The ***mathematical semantics*** of the program P is a model I_P of the theory P that is ***standard*** in an adequate sense. After entering the program P into the machine, the user can ask questions about what properties hold in his/her model. Such questions, called ***queries***, belong to a specified class of sentences in the language of P and have the property that for sentences φ in that class the standard model I_P satisfies φ if and only if φ is provable from the axioms of the theory P. When the user submits a query φ, if it is the case that φ is a provable consequence of the axioms in P, then the machine will return a set of ***answers*** justifying the truth of φ. We can view each of these answers as different ***proofs*** of the the truth of φ; in other words, the ***operational semantics*** of the language is given by some ***proof theory***. If the query φ is not provable from P two things can happen: either the machine stops after a finite amount of time with the answer "failure," or otherwise the machine loops forever. Therefore, three things are made equivalent: computation in the machine, deduction in the logic, and satisfaction in the standard model.

Of course, the efficiency requirement applies exactly as before, and provides the pragmatic boundary between theorem proving and logic programming. We can summarize the strong view of logic programming under the slogan

Computation = Deduction = Satisfaction in the standard model.

I have already mentioned that the weak view of logic programming, being exclusively proof-theoretic in nature, is unsatisfactory. Nevertheless, weak logic programming seems to have the advantage of having a broader range of applicability, so that it could cover certain examples of logic programming languages for which strong logic programming might prove too restrictive. However, we have already seen in Proposition 9 that, thanks to the generality of the axioms for a logic, we can always associate a model theory to an entailment system so that the entailment system becomes a complete logic with initial models. This shows that there is no need for carrying along two different notions. Surprisingly enough, we can actually understand the weak view of logic programming not as a broader notion, but rather as a *special case* of the strong notion, one for which the models are proof-theoretic structures. This suggests making strong logic programming our basic notion. This is a richer, conceptually and semantically more satisfactory notion, yet in the sense just explained it is the notion that is most broadly applicable.

Definition 48 axiomatizes the strong logic programming view. This definition is a further step in a series of previous attempts by J.A. Goguen and the author to articulate a broad view of logic programming open to many logics and languages. The paper [32] presented this view and used it as a natural way to unify two logic programming language paradigms, the functional and the relational, by unifying their logics. It also argued that every program should have an initial model as its mathematical semantics, and showed that this was the case for first order functional programming, first order relational programming with Horn clauses, and their unification. This view was made formal in a paper by J.A. Goguen [24] using institutions. Goguen proposed that logic programming languages should have an underlying institution so that the statements of the language were sentences in that institution, the operational semantics was given by an efficient form of deduction in that institution, and the mathematical semantics was given by a class of models, preferably initial; the paper [28] by Goguen and Burstall also proposed this formalization. The definition below is very much in the same spirit, but it combines the proof-theoretic and model-theoretic aspects of the issue using the concepts developed in this paper to suggest two new conditions. One is a query completeness requirement with the explicit demand that what is provable should coincide with what is true in the initial model; the other is a formal definition of an operational semantics as an effective proof subcalculus. Also, the use of initial models for the mathematical semantics is here made mandatory.

Definition 48 A *logic programming language* $\mathcal{LP}$ is a 4-tuple $\mathcal{LP} = (\mathcal{L}, \underline{Sign}_0, stat, quer)$ with:

1. $\mathcal{L} = (\underline{Sign}, sen, \underline{Mod}, \vdash, \models)$ a logic.
2. $\underline{Sign}_0$ a subcategory of $\underline{Sign}$.
3. $stat : \underline{Sign} \longrightarrow \underline{Set}$ a subfunctor of the functor obtained by composing sen with the finite powerset functor, i.e., there is a natural inclusion $stat(\Sigma) \subseteq \mathcal{P}_{fin}(sen(\Sigma))$ for each $\Sigma \in \underline{Sign}$. Each $\Gamma \in stat(\Sigma)$ is called a set of Σ-*statements* in $\mathcal{LP}$. This defines a subcategory $\underline{Th}_{stat}$ of $\underline{Th}_0$ whose objects are theories $P = (\Sigma, \Gamma)$ with $\Sigma \in \underline{Sign}_0$ and $\Gamma \in stat(\Sigma)$, and with morphisms axiom-preserving theory morphisms H such that $H \in \underline{Sign}_0$. Each such theory $P \in \underline{Th}_{stat}$ is called a *program* in $\mathcal{LP}$.
4. $quer : \underline{Sign} \longrightarrow \underline{Set}$ a subfunctor of the sen functor. The sentences $\varphi \in quer(\Sigma)$ are called the Σ-*queries* of $\mathcal{LP}$.

In addition, the following properties are satisfied:

Mathematical Semantics: Each program $P \in \underline{Th}_{stat}$ has an initial model I_P. The denotation function

$$P \mapsto I_P$$

is called the *mathematical semantics* of $\mathcal{LP}$.

Query Completeness: For each program $P = (\Sigma, \Gamma)$ and query $\varphi \in quer(\Sigma)$ we have

$$\vdash_P \varphi \Leftrightarrow I_P \models_\Sigma \varphi.$$

Operational Semantics: There is an effective proof subcalculus of the form $\mathcal{O} = (\underline{Sign}, sen, \vdash, \underline{Sign}_0, sen_0, stat, quer, P, Pr, \pi)$, i.e., having $ent(\mathcal{L})$ as its underlying entailment system, $\underline{Sign}_0$ as its category of admissible signatures, $stat$ as its axioms, and $quer$ as its conclusions.

The effective proof subcalculus $\mathcal{O}$ is not assumed to be unique. Any such $\mathcal{O}$ is called *an operational semantics* for the logic programming language $\mathcal{LP}$. □

Notice that, as pointed out at the end of Section 3.2, given an operational semantics $\mathcal{O}$ for a logic programming language $\mathcal{LP}$, every program P has an associated partial recursive search function

$$search_P : quer(P) \times \mathbb{N} \longrightarrow \mathcal{P}_{fin}(proofs(P)),$$

so that we can ask for as many answers to a query φ as we desire, and then we get back the answers if they exist, or otherwise either information about failure in finite time or no answer at all. One way in which the operational semantics of the logic programming language can change is by changing the mode of computation; for example, in a debugging mode answers should be much more informative than in a standard mode. The axioms for an effective proof subcalculus are very flexible; they allow expressing different notions of "proof" suitable for different purposes as different subcalculi.

This finishes our axiomatization of logic programming languages. However, the above definition has the drawback of not taking into account efficiency considerations. In practice, we would not be willing to use a logic programming language if answers to queries were to take an inordinate amount of time compared with the time that it would take to compute the solution to the problem using a more conventional language. We might be willing to accept the system implementing the language as a *theorem prover*, but not as a programming language. Therefore, some pragmatic line must be drawn between theorem provers and programming languages. It may be impossible to settle this issue once and for all, for the following reasons:

Emergence of increasingly more efficient operational semantics: Linear resolution made it possible to develop interpreters for Horn clause logic, and term rewriting allowed equational logic programming interpreters. Present compilation techniques for Horn clauses and for functional languages permit developing compilers that make the efficiency of these languages entirely acceptable compared with more conventional languages run on the same sequential machines.

New models of parallel computation and new architectures: These can drastically alter the mathematical complexity of many problems and make possible computations that were not feasible with previous technology. Logic programming languages, thanks to their declarative character, can, in fact, play a leading role in the discovery of such new models and architectures.

Advances in hardware technology: For the moment, these show a dramatic increase in computing speed and a decrease in device size, although the laws of physics will eventually pose a hard boundary to such advances.

However, there are intrinsic complexity theory bounds that no technological advance can reverse. Therefore, one of the most important tasks in logic programming is to find efficient proof subcalculi for those entailment systems that have them.

In practice, we can observe a *migration* process. First, certain logics exist; then, some theorem provers are developed to mechanize their deduction; and, finally, some of these theorem prover techniques are found to be efficient and give birth to programming language interpreters. Each new language in turn suggests new models of computation, new compilers, and new architectures. In first order logic programming this is clearly the trend, and in type theory a similar trend is apparent for higher order logics. Of course, historical developments are not logical necessities, and in the future we may find more and more language designers in the role of producers rather than consumers of new logics.

6.1 First Order Logic Programming

The logic programming ideas historically originated from the tradition of first order resolution theorem proving [42,72] and were first embodied in the Prolog language [11]. The Prolog culture has been so successful that for many researchers the part —i.e., Horn clause relational programming in its different variants, or perhaps first order logic programming for the truly ambitious— seems to become identified with the whole. This of course may be styfling. A different theorem proving tradition, namely equational theorem proving, has existed alongside and provided term rewriting techniques that were recognized by several researchers in the late seventies as a very good basis for designing and giving semantics to functional programming languages. Pioneering work in this direction includes that of J.A. Goguen, who created the OBJ language [23,25], and M.J. O'Donnell's thesis [55]. In the 1980's it became gradually apparent that these two styles of first order logic programming, the relational based on Horn clauses and the functional based on equations, should be unified and several proposals emerged. The Eqlog language [33] was the first proposal suggesting that this unification could best be achieved by unifying both logics into Horn clause logic with equality, and giving an initial model semantics for the resulting programs.

What follows is a discussion of a variety of first order logic programming styles. The emphasis is on the particular choices of statements, queries and proofs. I mention some languages only to give a few examples. There are of course many other languages that could be mentioned, but this is not a survey. To simplify the exposition, I present the ideas in an unsorted first order logic notation; however, all that I say generalizes to many-sorted and order-sorted first order logic.

Horn Clause Logic Programming

In this case, the signatures are finite first order signatures, and the sets of statements are finite sets of Horn clauses, i.e., of sentences of the form

$$\forall \vec{x}\; A \Leftarrow B_1, ..., B_n$$

where $A, B_1, \ldots, B_n$ are *atomic formulas* that do not involve an equality predicate. The queries are existential sentences of the form

$$\exists \vec{x}\; C_1, ..., C_n$$

with $C_1, ..., C_n$ atomic formulas not involving equality. The initial model of a program $P = (\Sigma, \Gamma)$ is its *Herbrand model* $T_{\Sigma,\Gamma}$, whose functional part consists of the term algebra T_F on the function symbols F of Σ, and for each n-ary predicate symbol p and $\vec{t} \in T_F^n$ we have,

$$\vec{t} \in p_{T_{\Sigma,\Gamma}} \text{ iff } \Gamma \vdash_\Sigma p(\vec{t}).$$

Query completeness is a direct consequence of Herbrand's theorem. The standard operational semantics is Horn clause resolution [42]. A *proof* of a query $\exists\bar{x}\ C_1, ..., C_n$ for a program P is a substitution θ such that $\vdash_P\ \theta(C_1), ..., \theta(C_n)$. Although Prolog [11] is the most popular Horn clause logic programming language, its extralogical features, the incompleteness of its search strategy and its nonstandard unification make it fall short of the logic programming ideal.

Equational Logic Programming

In this case, signatures are finite functional signatures, and the sets of statements are finite sets of Church-Rosser and terminating equations[20]. The mathematical semantics of a program $P = (F, E)$ is given by the *initial algebra* $T_{F,E}$ in the class of all models of the theory (F, E). For any set E of F-equations, the initial algebra $T_{F,E}$ has a very simple construction as the quotient algebra of the term algebra T_F by the congruence relation $\equiv_E$ defined by

$$t \equiv_E t' \text{ iff } E \vdash_F t = t'.$$

See the original ADJ paper [37], or the survey [53] for a detailed proof of the initiality of $T_F/\equiv_E$. The operational semantics is term rewriting, i.e., equational deduction using the equations only from left to right. Since the equations are Church-Rosser and terminating, each term t rewrites after a finite number of steps to a unique canonical form $can_E(t)$ that cannot be further simplified by the equations. The term $can_E(t)$ is the unique canonical representative of the $\equiv_E$-equivalence class $[t]$. Therefore, an isomorphic (but computationally more intuitive) representation of the initial algebra $T_{F,E}$ can be given as the set $Can_{F,E}$ whose elements are ground terms of the form $can_E(t)$; this set has an obvious F-algebra structure making it isomorphic to $T_F/\equiv_E$, namely for $f \in F_n$ and $t_1, ..., t_n \in Can_{F,E}$ we define $f_{Can_{F,E}}(t_1, ..., t_n) = can_E(f(t_1, ..., t_n))$. Two basic computations that can be performed by term rewriting are: reduction of a ground term t to canonical form, and deciding when two ground terms t and t' are made equal by the equations. They give rise to two types of queries:

1. $\exists x\ t = x$

2. $t = t'$

where the sentence $\exists x\ t = x$ is always true (since we could choose x to be t itself) but we are interested in the most informative proof possible, namely one where x is chosen to be $can_E(t)$. Therefore, for queries φ of type (1) query completeness is trivial; for queries of type (2) it follows trivially from the definition of $\equiv_E$.

An example of a logic programming language of this type based on untyped equational logic is the language of Hoffmann and O'Donnell [39,54]. The OBJ0 language was also untyped [25], but subsequent versions have all been typed. OBJ2 and OBJ3 [19,29] are based on order-sorted equational logic [35], where types can be related by a partial order subtype relation, e.g., $Nat < Int$, and operations can have several

[20]This condition can be relaxed, e.g., by dropping the termination property. O'Donnell [54] does not require termination, but imposes sufficient conditions to ensure the Church-Rosser property. In any event, Church-Rosser and terminating equations are powerful enough to specify *all* total computable functions [5].

typings, e.g., natural, integer, rational and complex addition, all with the same restriction to subtypes; this makes OBJ programs very expressive. The expressiveness of OBJ programs is also increased by the use of conditional equations, so that the whole discussion above should be understood as taking place in the context of conditional equational logic.

One of the great advantages of logic programming languages is that they are declarative and make no commitments to a particular execution sequence. Therefore, they can be used to design and program entirely new parallel architectures. In the case of equational logic programming, one such architecture is the Rewrite Rule Machine that Goguen, Leinwand, Winkler, Aida and I are building at SRI [36]. Its model of computation is a new operational semantics for equational logic programming based on concurrent term rewriting [30].

Horn Clause Logic Programing with Equality

Horn clause logic programming is *relational*, whereas equational logic programming is *functional*. Each approach has its own, somewhat complementary, strengths. For a problem where searching is crucial, the relational approach is ideal. However, many computations are functional in nature and do not require any search or backtracking; for those, term rewriting is best. In [32] Goguen and I suggested that the functional and relational approaches to logic programming could be combined by combining the corresponding logics. The combination is of course Horn clause logic *with* equality. In this logic programming style, the signatures are finite first order signatures, and the sets of statements are finite collections of Horn clauses that now may involve the equality predicate and such that the clauses for equality are Church-Rosser and terminating in a suitable sense[21]. The mathematical semantics is an initial model semantics that generalizes Herbrand models and initial algebras. Given a first order signature $\Sigma = (F, P)$ and a set Γ of Horn clauses —possibly involving equalities— the initial model $T_{\Sigma,\Gamma}$ has a functional part consisting of the quotient algebra of the term algebra T_F under the congruence relation $\equiv_\Gamma$ defined by

$$t \equiv_\Gamma t' \text{ iff } \Gamma \vdash_\Sigma t = t'$$

and a relational part given by

$$([t_1], ..., [t_n]) \in p_{T_{\Sigma,\Gamma}} \text{ iff } \Gamma \vdash_\Sigma p(t_1, ..., t_n)$$

for each $p \in P_n$ in P. See [34] for a detailed proof of the initiality of $T_{\Sigma,\Gamma}$ in the general case of order-sorted logic, that contains unsorted logic as a sublogic. Queries are existential sentences of the form

$$\exists \bar{x}\, C_1, ..., C_n$$

with $C_1, ..., C_n$ atomic formulas but now some of them can be equations. Query completeness is a direct consequence of Herbrand's theorem; see [34] for a proof in the general case of order-sorted logic. Solving queries is done in a fashion entirely similar to ordinary Horn clause resolution; the only difference is that standard unification is

[21]Since we have Horn clauses, the equations may be conditional and even have predicates other than equality in their conditions.

replaced by unification modulo the equations of the program. By the Church-Rosser and terminating assumptions, this can be done by some complete strategy for narrowing [40]. As before, the answers to queries are substitutions that make the instance of the query provable. The Eqlog language [32] is an example of a language in this style of logic programming; its logic is order-sorted Horn clause logic with equality. Other approaches encode predicates as functions [18,15] or functions as predicates [71].

Logic Programming in other Fragments of First Order Logic

Although Horn clause logic with equality is in a sense the end of the road for fragments of first order logic admitting initial models [48], we can still view the kind of weak logic programming that is possible in any fragment of first order logic as an instance of strong logic programming for a logic $\mathcal{L}'$ that has the same entailment system as first order logic, but has a different underlying institution admitting initial models. There are several such possible institutions. One is given by Theorem 9, which in a sense is the ideal model theory for people with a strong proof-theoretic bias; another such institution could have hyperdoctrine models such as "logical categories" [73]. As a model theory, such choices seem preferable to standard first order logic where one would have to worry about many different models, none of them initial.

6.2 Higher Order Logic Programming

Although type theory and logic programming share a vital connection with logic, they have developed in relative isolation from each other. Much can be gained at both the conceptual and practical levels from an attempt to understand the relationships between these two fields. Conceptually, logic programming can be saved from becoming parochial and losing important new opportunities, and type theory may gain new logic and model-theoretic insights. At the practical level, what can be gained is a much better understanding of how to design powerful new languages that integrate such features as generic modules, higher order functions, logical variables and subtypes, and yet have a clear and rigorous semantics based on logic. Categorical logic plays a key role in relating type theory and logic programming, so that functional languages based on type theory can be understood as logic programming languages in the strong sense of our axiomatization.

One of the greatest strengths of categorical logics is that they unify proof theory and model theory in a particularly illuminating way. Given a theory T, the associated free category $\mathcal{F}(T)$ is at the same time an abstract theory, providing a notion of equivalence of proofs, and the initial model of the theory. The logic programming axioms underscore the importance of these free or "term" models, because in them mathematical and operational semantics are interlocked as two aspects of the same reality. Being initial, these models are in a sense the most general, and being generated by the rules of the logic, they wear their operational semantics on their sleeves.

Type theories can be used to define programming languages with powerful type mechanisms. They can also be used as formal frameworks to reason about programs or to automatically generate correct programs from their specifications. Although, of course, it is one of the great advantages of type theory that formal reasoning about a program written in it can be carried out in the type theory itself, for the purposes of this paper, I concentrate on the first use. I consider three well-known examples of type theory: the typed lambda calculus, the Girard-Reynolds polymorphic lambda calculus, and Martin-Lōf type theory. All of these calculi are higher order equational logics, and all have a notion of *reduction* entirely similar to term rewriting in first order equational logic. Therefore, in all three cases programs are finitary equational theories whose equations are Church-Rosser and terminating in the given calculus, and operational semantics is given by reduction to canonical form. Queries are similar to those of the first order equational case, i.e., they are sentences of the form:

1. $\exists x\, t = x$
2. $t = t'$

with t and t' terms in the appropriate syntax[22].

The Typed Lambda Calculus

The typed lambda calculus is a categorical logic on $\underline{CCCat}$, the category of cartesian closed categories. The mathematical semantics of a program $P = (\Sigma, E)$ is given by its initial model $1_{\mathcal{F}(\Sigma,E)}$, where $\mathcal{F}(\Sigma, E)$ is the free cartesian closed category generated by $P = (\Sigma, E)$ (see [43] I.10–11 for a detailed description of such theories, called there "typed lambda calculi" and the $\mathcal{F}(\Sigma, E)$ construction; note that they assume a weak natural numbers object in their theories, but this is not an essential requirement). Query completeness is trivially satisfied. A particularly elegant operational semantics is provided by Curien's *categorical combinators* [13].

The ML language [38] is closely related to the typed lambda calculus, but it is instead based on the *polymorphic* lambda calculus; expressions in general do not have one type, but a family of types that are the instances of a unique type expression involving type variables. Categorical combinators can be used to provide a simple and efficient ML implementation [12].

The Girard-Reynolds Second Order Polymorphic Lambda Calculus

This calculus, proposed by Girard [21] and discovered independently by Reynolds [60], is more powerful than the usual polymorphic lambda calculus; it allows universal quantification of type variables inside type formulas. There are two possible categorical semantics. One, based on a type of hyperdoctrines called PL-categories, was given by Seely [63]; the other, called the "universe model semantics," is based on relatively cartesian closed categories and was proposed in [52]. These two categorical semantics are related by a map of logics that is the identity on the underlying entailment system and that for each theory T provides a forgetful functor from its universe models to its PL-category models. The PL-category models provide only *names* for the types, but not the types themselves; the universe models, however, provide type extensions as objects of a category. Again, query completeness is trivial and the operational semantics is given by reduction of expressions to normal form.

[22]In some cases, for example in Martin-Lōf type theory, we can have similar queries for *type expressions*, for which there is also a normal form.

From the point of view of applications, there are compelling reasons to extend the second order polymorphic lambda calculus in various ways. Although all the provably terminating functions can be expressed in the basic calculus, not all the algorithms for computing a function of this kind are expressible, and therefore it is more expressive to have full recursion. Also, to do programming-in-the-large, it is very useful to be able to compute with modules as values, i.e., to assume that Type:Type. Extensions of the universe model semantics that provide a categorical semantics for the second order polymorphic lambda calculus with full recursion and/or with Type:Type are given in [52]; for Type:Type, a categorical semantics was given in [70]; proof rules for extensions of this kind are given in [8]. The language Pebble [7] has full recursion, Type:Type as well as other features; the Quest language [9] is also in this category, and in addition provides subtypes.

Martin-Löf Type Theory

Martin-Löf type theory [49,50] provides powerful type constructions with both universal and existential quantification, equality types, etc. Normalization of expressions is Church-Rosser and terminating both for terms and for type expressions and provides the operational semantics. The work of Seely [65] has shown how Martin-Löf type theory can be viewed as a categorical logic on <u>*LCCCat*</u>, the category of locally cartesian closed categories. If equality types are dropped, the categorical semantics can be broadened in several closely related ways such as contextual categories [10], or relatively cartesian closed categories [41].

7 Concluding Remarks

The main focus of this paper has been on basic concepts and definitions. Once the basic framework is set up, general results about logics satisfying some additional conditions should be investigated. Results of this nature are obtained in the context of traditional set-theoretic structures by the methods of abstract model theory [2], and several such results have already been obtained for general institutions by Tarlecki [67,68] and by Sannella and Tarlecki [61]. Some of the additional conditions that one may want to impose have to do with properties of the category of signatures and the functor of sentences that can have a natural formulation in terms of the "charters" and "parchments" of Goguen and Burstall [28].

The satisfaction relation between sentences and models can be seen as a characteristic function taking the value *true* or *false*. B. Mayoh [51] suggested interesting applications in which one would like to broaden the notion of "truth value" and proposed a generalization of institutions called *galleries*. Goguen and Burstall [28] gave a nice categorical formulation of institutions as a pair of functors together with an extranatural transformation or "wedge" [47] involving the truth value category **2** with objects *true* and *false* and only one nonidentity morphism from *false* to *true*; they then suggested a notion of a *generalized institution* that can be obtained replacing **2** by a category $\mathcal{V}$ of truth values in their categorical formulation. They argued that Mayoh's galleries could, after some modifications, be seen as generalized institutions. A substantial portion of the present theory could have been developed in the more general setup of a truth value category other than **2**, and the implications of that possibility are an interesting topic of future research. However, having the notion of a proof calculus available may decrease the need for the extra generality. What is needed is a study of examples to see how naturally they can be expressed in the different frameworks. The ideas of Poigné [58], who has proposed another way of generalizing institutions should also be taken into account.

The study of mappings between the different logical structures should be further developed. Those mappings consist of a functor relating the two categories of theories and of one or more natural transformations; for each of those transformations one could in principle choose between a "forward" and a "backward" direction. However, not all possible combinations may behave well or have interesting examples. I have presented particular choices that seemed natural, had interesting examples and permitted defining a notion of logical substructure. However, such choices should not exclude other possibilities that may be equally useful. More experience with examples is needed to ascertain what choices should be favored; computer science can provide a rich source of examples and applications.

Computer science applications have provided the original stimulus for the development of the theory; with the basic concepts now in place, one of the main tasks ahead is to bring the abstract concepts to bear on specific problem areas within computer science. For logic programming, Section 6 has given just the beginnings of an application, but several other important issues, such as compilation, or the use of mappings between logics to design new programming languages with more powerful features have not been discussed. Type theory and concurrency are also areas where many applications are possible; the paper [52] gives a particular type theory application, and the joint paper [56] with N. Martí-Oliet gives an application to concurrency. Applications to other computer science areas such as artificial intelligence and automated deduction also seem very natural. I hope that other researchers will find the methods useful and will undertake many of those applications themselves.

References

[1] L. Bachmair, N. Dershowitz, and J. Hsiang. Orderings for equational proofs. In *Proceedings of the Symposium on Logic in Computer Science*, pages 346–357, IEEE, June 1986.

[2] J. Barwise and S. Feferman (eds.). *Model-Theoretic Logics*. Springer-Verlag, 1985.

[3] K. J. Barwise. Axioms for abstract model theory. *Ann. Math. Logic*, 7:221–265, 1974.

[4] Jean Bénabou. Structures algébriques dans les catégories. *Cahiers de Topologie et Géometrie Différentielle*, 10:1–126, 1968.

[5] Jan Bergstra and John Tucker. Characterization of computable data types by means of a finite equational specification method. In J. W. de Bakker and J. van Leeuwen, editors, *Automata, Languages and Programming, Seventh Colloquium*, pages 76–90, Springer-Verlag, 1980. Lecture Notes in Computer Science, Volume 81.

[6] A. Boileau and A. Joyal. La logique des topos. *J. Symbol. Logic*, 46(1):6–16, 1981.

[7] R. Burstall and B. Lampson. A kernel language for abstract data types and modules. In G. Kahn, D.B. MacQueen, and G.D. Plotkin, editors, *Semantics of Data Types*, pages 1–50, Springer Lecture Notes in Computer Science 173, 1984.

[8] L. Cardelli. *A polymorphic λ-calculus with Type:Type*. Technical Report, DEC System Research Center, Palo Alto, Ca, 1985.

[9] L. Cardelli. *A Quest Preview.* Technical Report, DEC System Research Center, Palo Alto, Ca, 1988.

[10] J. Cartmell. Generalised algebraic theories and contextual categories. *Annals Pure Appl. Logic*, 32:209–243, 1986.

[11] A. Colmerauer, H. Kanoui, and M. van Caneghem. *Etude et Réalisation d'un Système Prolog.* Technical Report, Groupe d'Intelligence Artificielle, U.E.R. de Luminy, Université d'Aix-Marseille II, 1979.

[12] G. Cousineau, P.-L. Curien, and M. Mauny. The categorical abstract machine. *Science of Computer Programming*, 8:173–202, 1987.

[13] Pierre-Louis Curien. *Categorical Combinators, Sequential Algorithms and Functional Programming.* Pitman, London, 1986.

[14] Valeria C.V. de Paiva. *The Dialectica Categories.* PhD thesis, Mathematics Department, University of Cambridge, 1988.

[15] Nachum Dershowitz and David A. Plaisted. Equational programming. In J. Richards, editor, *Machine Intelligence 11: The logic and acquisition of knowledge*, pages 21–56, Oxford University Press, 1988.

[16] H.-D. Ebbinghaus. Extended logics: the general framework. In J. Barwise and S. Feferman, editors, *Model-Theoretic Logics*, pages 25–76, Springer Verlag, 1985.

[17] J. Fiadeiro and A. Sernades. *Structuring Theories on Consequence.* Technical Report , INESC/IST, Lisbon, 1988?

[18] Laurent Fribourg. Oriented equational clauses as a programming language. *Journal of Logic Programming*, 1(2):179–210, 1984.

[19] Kokichi Futatsugi, Joseph Goguen, Jean-Pierre Jouannaud, and José Meseguer. Principles of OBJ2. In Brian Reid, editor, *Proceedings of 12th ACM Symposium on Principles of Programming Languages*, pages 52–66, ACM, 1985.

[20] Jean-Yves Girard. Towards a geometry of interaction. In J.W. Gray and A. Scedrov, editors, *Proc. AMS Summer Research Conference on Categories in Computer Science and Logic, Boulder, Colorado, June 1987*, American Mathematical Society, 1988.

[21] J.Y. Girard. *Interprétation Fonctionelle et Élimination des Coupures dans l'Arithmétique d'ordre Supérieure.* PhD thesis, Univ. Paris VII, 1972.

[22] J.Y. Girard. Linear Logic. *Theoretical Computer Science*, 50:1–102, 1987.

[23] Joseph Goguen. Abstract errors for abstract data types. In Peter Neuhold, editor, *Proceedings of First IFIP Working Conference on Formal Description of Programming Concepts*, pages 21.1–21.32, MIT, 1977. Also published in *Formal Description of Programming Concepts*, Peter Neuhold, Ed., North-Holland, pages 491-522, 1979.

[24] Joseph Goguen. One, none, a hundred thousand specification languages. In H.-J. Kugler, editor, *Information Processing '86*, pages 995–1003, Elsevier, 1986. Proceedings of 1986 IFIP Congress.

[25] Joseph Goguen. Some design principles and theory for OBJ-0, a language for expressing and executing algebraic specifications of programs. In Edward Blum, Manfred Paul, and Satsoru Takasu, editors, *Proceedings, Mathematical Studies of Information Processing*, pages 425–473, Springer-Verlag, 1979. Lecture Notes in Computer Science, Volume 75; Proceedings of a Workshop held August 1978.

[26] Joseph Goguen and Rod Burstall. *Institutions: Abstract Model Theory for Computer Science.* Technical Report CSLI-85-30, Center for the Study of Language and Information, Stanford University, 1985. Also submitted for publication.

[27] Joseph Goguen and Rod Burstall. Introducing institutions. In Edmund Clarke and Dexter Kozen, editors, *Logics of Programs*, pages 221–256, Springer-Verlag, 1984. Lecture Notes in Computer Science, Volume 164.

[28] Joseph Goguen and Rod Burstall. A study in the foundations of programming methodology: specifications, institutions, charters and parchments. In David Pitt, Samson Abramsky, Axel Poigné, and David Rydeheard, editors, *Proceedings, Conference on Category Theory and Computer Programming*, pages 313–333, Springer-Verlag, 1986. Lecture Notes in Computer Science, Volume 240; also, Report Number CSLI-86-54, Center for the Study of Language and Information, Stanford University, June 1986.

[29] Joseph Goguen, Claude Kirchner, Hélène Kirchner, Aristide Mégrelis, and José Meseguer. An introduction to obj3. In Jean-Pierre Jouannaud and Stephane Kaplan, editors, *Proceedings, Conference on Conditional Term Rewriting, Orsay, France, July 8-10, 1987*, pages 258–263, Springer-Verlag, Lecture Notes in Computer Science No. 308, 1988.

[30] Joseph Goguen, Claude Kirchner, and José Meseguer. Concurrent term rewriting as a model of computation. In Robert Keller and Joseph Fasel, editors, *Proceedings, Graph Reduction Workshop*, pages 53–93, Springer-Verlag, 1987. Lecture Notes in Computer Science, Volume 279.

[31] Joseph Goguen and José Meseguer. Completeness of many-sorted equational logic. *Houston Journal of Mathematics*, 11(3):307–334, 1985. Preliminary versions have appeared in: *SIGPLAN Notices*, July 1981, Volume 16, Number 7,

pages 24-37; SRI Computer Science Lab Technical Report CSL-135, May 1982; and Report CSLI-84-15, Center for the Study of Language and Information, Stanford University, September 1984.

[32] Joseph Goguen and José Meseguer. Eqlog: equality, types, and generic modules for logic programming. In Douglas DeGroot and Gary Lindstrom, editors, *Logic Programming: Functions, Relations and Equations*, pages 295–363, Prentice-Hall, 1986. An earlier version appears in *Journal of Logic Programming*, Volume 1, Number 2, pages 179-210, September 1984.

[33] Joseph Goguen and José Meseguer. Equality, types, modules and generics for logic programming. In S.-A. Tärnlund, editor, *Proc. 2nd Intl. Logic Programming Conf., Uppsala, July 2-6, 1984*, pages 115–125, Uppsala University, 1984.

[34] Joseph Goguen and José Meseguer. Models and equality for logical programming. In Hartmut Ehrig, Giorgio Levi, Robert Kowalski, and Ugo Montanari, editors, *Proceedings, 1987 TAPSOFT*, pages 1–22, Springer-Verlag, 1987. Lecture Notes in Computer Science, Volume 250; also, Technical Report, Center for the Study of Language and Information, Stanford University.

[35] Joseph Goguen and José Meseguer. *Order-Sorted Algebra I: Partial and Overloaded Operations, Errors and Inheritance.* Technical Report to appear, SRI International, Computer Science Lab, 1989. Given as lecture at Seminar on Types, Carnegie-Mellon University, June 1983.

[36] Joseph Goguen, José Meseguer, Sany Leinwand, Timothy Winkler, and Hitoshi Aida. *The Rewrite Rule Machine.* Technical Report to appear, SRI International, Computer Science Lab, 1989.

[37] Joseph Goguen, James Thatcher, and Eric Wagner. *An Initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data Types.* Technical Report RC 6487, IBM T. J. Watson Research Center, October 1976. Appears in *Current Trends in Programming Methodology, IV*, Raymond Yeh, Ed., Prentice-Hall, 1978, pages 80-149.

[38] Robert Harper, David MacQueen, and Robin Milner. *Standard ML.* Technical Report ECS-LFCS-86-2, Dept. of Computer Science, University of Edinburgh, 1986.

[39] Christoph M. Hoffmann and Michael O'Donnell. Programming with equations. *Transactions on Programming Languages and Systems*, 1(4):83–112, 1982.

[40] Jean-Marie Hullot. Canonical forms and unification. In Wolfgang Bibel and Robert Kowalski, editors, *Proceedings, Fifth Conference on Automated Deduction*, pages 318–334, Springer-Verlag, 1980. Lecture Notes in Computer Science, Volume 87.

[41] J.M.E. Hyland and A. Pitts. The theory of constructions: categorical semantics and topos-theoretic models. In J.W. Gray and A. Scedrov, editors, *Proc. AMS Summer Research Conference on Categories in Computer Science and Logic, Boulder, Colorado, June 1987*, American Mathematical Society, 1988.

[42] Robert Kowalski. *Logic for Problem Solving.* Technical Report DCL Memo 75, Department of Artificial Intelligence, University of Edinburgh, 1974. Also, a book in the Artificial Intelligence Series, North-Holland Press, 1979.

[43] J. Lambek and P.J. Scott. *Introduction to Higher Order Categorical Logic.* Cambridge Univ. Press, 1986.

[44] Joachim Lambek. Deductive systems and categories ii. In *Category Theory, Homology Theory and their Applications I,* Springer Lecture Notes in Mathematics No. 86, 1969.

[45] F. William Lawvere. Functorial semantics of algebraic theories. *Proceedings, National Academy of Sciences,* 50, 1963. Summary of Ph.D. Thesis, Columbia University.

[46] F.W. Lawvere. Adjointness in foundations. *Dialectica,* 23(3/4):281–296, 1969.

[47] Saunders MacLane. *Categories for the working mathematician.* Springer, 1971.

[48] J.A. Makowski. *Why Horn Formulas Matter in Computer Science: Initial Structures and Generic Examples.* Technical Report 329, C.S. Dept, Technion, July 1984.

[49] P. Martin-Lōf. An Intuitionistic Theory of Types: Predicative Part. In H.E. Rose and J.C. Shepherdson, editors, *Logic Colloq. '73,* Noth-Holland, 1973.

[50] P. Martin-Lōf. *Intuitionistic Type Theory.* Bibliopolis, 1984.

[51] Brian H. Mayoh. *Galleries and Institutions.* Technical Report DAIMI PB-191, Computer Science Dept., Aarhus University, 1985.

[52] J. Meseguer. Relating Models of Polymorphism. In *Proc. POPL'89,* pages 228–241, ACM, 1988.

[53] José Meseguer and Joseph Goguen. Initiality, induction and computability. In Maurice Nivat and John Reynolds, editors, *Algebraic Methods in Semantics,* pages 459–541, Cambridge University Press, 1985.

[54] M. O'Donnell. *Equational Logic as a Programming Language.* MIT Press, 1985.

[55] Michael J. O'Donnell. *Computing in Systems Described by Equations.* Springer-Verlag Lecture Notes in Computer Science 58, 1977.

[56] Narciso Martí-Oliet and José Meseguer. *From Petri Nets to Linear Logic.* Technical Report SRI-CSL-89-4, C.S. Lab., SRI International, March 1989.

[57] Gordon D. Plotkin. *A Structural Approach to Operational Semantics.* Technical Report DAIMI FN-19, Computer Science Dept., Aarhus University, 1981.

[58] Axel Poigné. Foundations are rich institutions, but institutions are poor foundations. 1986. Imperial College.

[59] Dag Prawitz. *Natural Deduction.* Almqvist and Wiksell, Stockholm, 1965.

[60] J.C. Reynolds. Towards a Theory of Type Structure. In B. Robinet, editor, *Programming Symposium*, pages 408–425, Springer Lecture Notes in Computer Science 19, 1974.

[61] Donald Sannella and Andrzej Tarlecki. Specifications in an arbitrary institution. *Information and Computation*, 76:165–210, 1988.

[62] D. Scott. Completeness and axiomatizability in many-valued logic. In L. Henkin et al., editor, *Proc. Tarski Symp.*, pages 411–435, AMS, 1974.

[63] R.A.G. Seely. Categorical semantics for higher order polymorphic lambda calculus. *J. Symbol. Logic*, 52(4):969–989, 1987.

[64] R.A.G. Seely. Linear logic, $*$-autonomous categories and cofree coalgebras. In J.W. Gray and A. Scedrov, editors, *Proc. AMS Summer Research Conference on Categories in Computer Science and Logic, Boulder, Colorado, June 1987*, AMS, 1988.

[65] R.A.G. Seely. Locally cartesian closed categories and type theory. *Math. Proc. Camb. Phil. Soc.*, 95:33–48, 1984.

[66] Joseph R. Shoenfield. *Degrees of Unsolvability*. North-Holland, 1971.

[67] Andrzej Tarlecki. On the Existence of Free Models in Abstract Algebraic Institutions. *Theoretical Computer Science*, 37:269–304, 1985.

[68] Andrzej Tarlecki. Quasi-varieties in abstract algebraic institutions. *Journal of Computer and System Sciences*, 33:333–360, 1986.

[69] A. Tarski. On some fundamental concepts of metamathematics. In *Logic, Semantics, Metamathematics*, pages 30–37, Oxford U.P., 1956.

[70] P. Taylor. *Recursive Domains, Indexed Category Theory and Polymorphism*. PhD thesis, Mathematics Department, University of Cambridge, 1987.

[71] Maarten H. van Emden and Keitaro Yukawa. *Equational Logic Programming*. Technical Report CS-86-05, University of Waterloo, March 1986.

[72] M.H. van Emden and R.A. Kowalski. The semantics of predicate logic as a programming language. *Journal of the Association for Computing Machinery*, 23(4):733–742, 1976.

[73] H. Volger. Completeness theorem for logical categories. In F.W. Lawvere, C. Maurer, and G.C. Wraith, editors, *Model Theory and Topoi*, Springer Lecture Notes in Mathematics No. 445, 1975.